



Introduction to Robot Raconteur® using Python

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>

March 18, 2017

Contents

1	Introduction	4
1.1	Example Robot	8
1.2	Advantages over roslib	9
2	Service definitions	10
2.1	Value types	11
2.2	Object types	12
2.3	Options	15
2.4	Exceptions	15
2.5	Robot Raconteur naming	16
3	Robot Raconteur Python	16
3.1	Python ↔ Robot Raconteur data type mapping	17
3.2	Python Function References	17
4	iRobot Create Python example	17
4.1	Simple service	19
4.2	Simple client	23
4.3	iRobot Create Service	24
4.4	iRobot Create Client	26
5	Webcam Example	27
5.1	Webcam Service	27
5.2	Webcam Client	29
5.3	Webcam Client (streaming)	29
5.4	Webcam Client (memory)	30
6	Service auto-discovery	30
7	Authentication	31
8	Exclusive object locks	33
8.1	User	33
8.2	Client	33
8.3	Monitor	34
9	Time-critical software with Wire member	34
10	Forward compatibility with the “implements” statement	35
11	Asynchronous programming	35
12	Gather/Scatter operations	37
13	Debugging Python with Eclipse PyDev	37

14	PipeBroadcaster and WireBroadcaster	37
15	Conclusion	39
A	URL Format	40
B	Port Sharer	42
C	Robot Raconteur Reference	42
C.1	RobotRaconteurNode	42
C.2	EventHook	55
C.3	ServiceInfo2	55
C.4	NodeInfo2	56
C.5	Pipe	56
C.6	PipeEndpoint	57
C.7	Callback	60
C.8	Wire	60
C.9	WireConnection	61
C.10	TimeSpec	63
C.11	ArrayMemory	65
C.12	MultiDimArrayMemory	66
C.13	ServerEndpoint	67
C.14	ServerContext	68
C.15	AuthenticatedUser	69
C.16	NodeID	70
C.17	RobotRaconteurException	71
C.18	RobotRaconteurRemoteException	71
C.19	Transport	72
C.20	LocalTransport	72
C.21	TcpTransport	74
C.22	CloudTransport	78
C.23	HardwareTransport	79
C.24	PipeBroadcaster	79
C.25	WireBroadcaster	80
D	Example Software	82
D.1	iRobotCreateService.py	82
D.2	iRobotCreateClient.py	88
D.3	SimpleWebcamService.py	89
D.4	SimpleWebcamClient.py	94
D.5	SimpleWebcamClient_streaming.py	95
D.6	SimpleWebcamClient_memory.py	97
D.7	iRobotCreateAsyncClient.py	98
D.8	iRobotCreateService_Broadcaster.py	100
D.9	SimpleWebcamService_Broadcaster.py	105

1 Introduction

This document provides an introduction and overview of Robot Raconteur® and serves as the documentation for using Python with Robot Raconteur®.

Robot Raconteur® is a royalty-free communication library designed to ease the integration of complex automation systems that are composed of disparate components that run within different processes on a computer, are distributed over a network, or are embedded devices. These components are often produced by different vendors with completely different interfaces that may not run on the same platforms and are frequently mutually exclusive in terms of the API provided for the user of the component. Add in that most modern systems are distributed over a network and the result is a long, often frustrating development cycle to produce a front-end that is capable of controlling all of the elements in a high-level user friendly manner. For modern laboratory or prototype systems this usually means producing a MATLAB, Python, or LabView front end that can be scripted. After the prototype is completed a high-level interface may be developed in a language like C++. Robot Raconteur is designed specifically to ease this design process and adds a number of additional capabilities that would otherwise be time consuming to implement.

Robot Raconteur provides a language, platform, and technology neutral augmented object-oriented communication system that provides the ability for a *client* to rapidly access functionality exposed by a *service* either within the same computer, over a network, or over a connection technology such as USB. Robot Raconteur is built around the idea of a *Service* exposing *Objects* to a client. (This tutorial assumes that you are familiar with basic object-oriented programming used in Python, C#, or Java. If you are not please review before continuing.) This is accomplished by registering a *root object* as a service within a Robot Raconteur *Node* that acts as the server. Object *Members* consist of the contents of the object, and are typically *functions*, *properties*, and *events*. (In C++, Python, and Java these are made by using helper classes like Boost or JavaBeans.) Robot Raconteur uses an *augmented object-oriented* model that has a number of member types: properties, functions, events, objrefs, pipes, callbacks, wires, and memories. The specific function of these members will be discussed later. These members are mirrored to *Object References* (sometimes called “Proxy Objects”) in a client Robot Raconteur *Node*. These references mirror the members and allow the client to access the members of the service objects through a *Transport Connection* between the client and service node. Figure 1 shows a diagram of this configuration. Multiple clients can access the same service simultaneously as shown in Figure 2.

A node can expose more than one service. Each service is registered with a unique name that is used as part of the URL to connect to the service.

The Robot Raconteur library contains almost all of the functionality to implement the communication between the client and service. It has been designed to require a minimal amount of “boilerplate” coding instead uses dynamic meta-programming when possible or code-generation when necessary to produce the “thunk” code that implements the conversion between the client

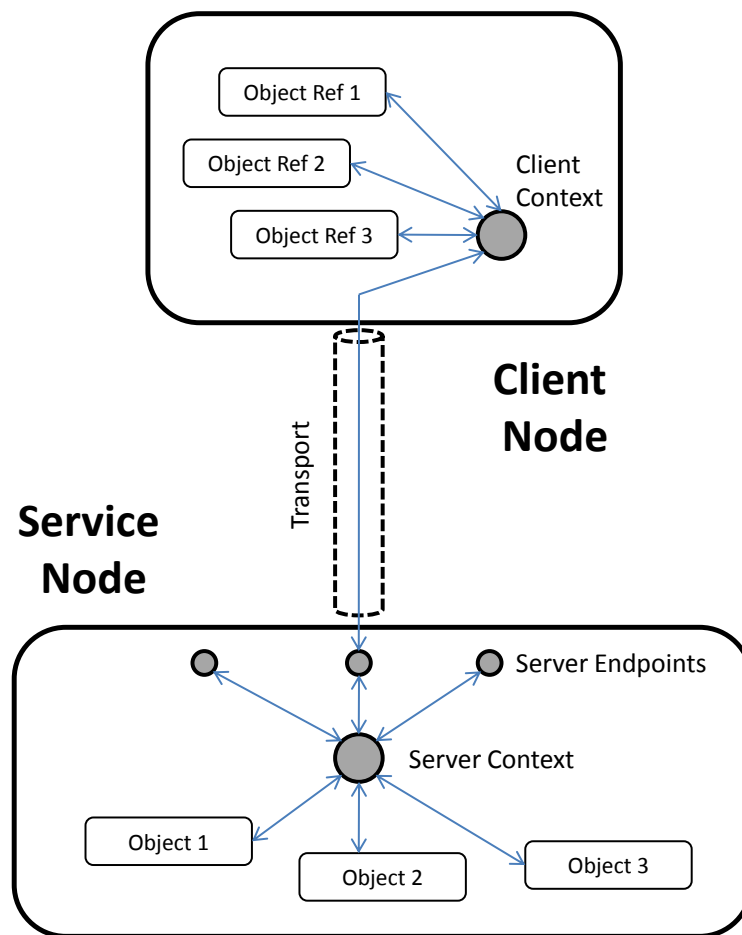


Figure 1: Configuration of Client-Service communication

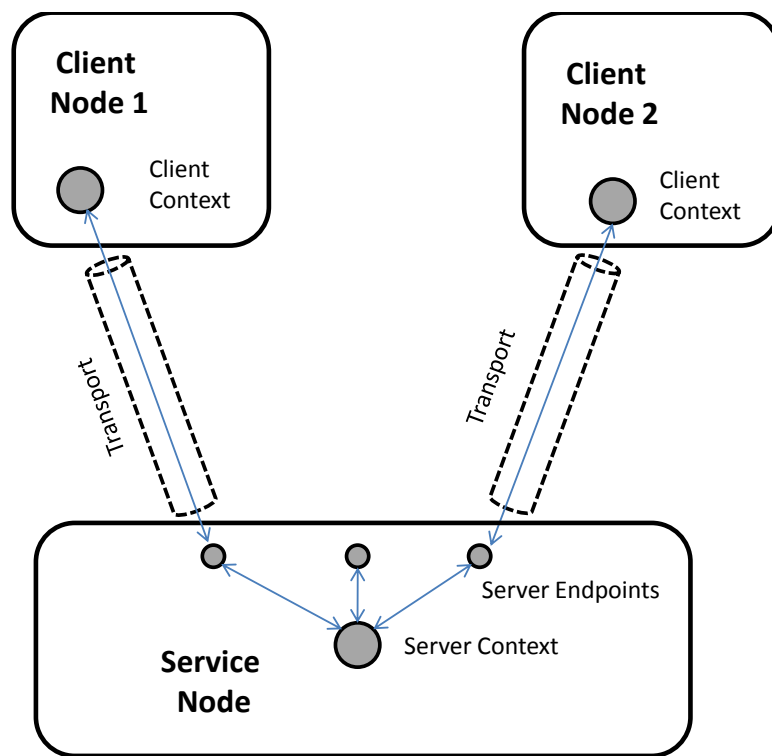


Figure 2: Configuration of Client-Service communication with multiple clients

member reference and the real member in the service. To the user the network layer is *almost* invisible. Robot Raconteur uses plain-text files called *Service Definition* files to define the objects and composite data types (structures) that are used in a service. Example 1 shows an example service definition, and Section 2 goes into great detail how these files are used. A service definition is a very simple way to define the *interface* to the service objects. The service definition provides a “lowest-common denominator” to all the languages that Robot Raconteur supports. These service definitions are used as the input to code generation or dynamic programming (such as in Python) and can result in tens of thousands of lines of codes in some situations that would otherwise need to be written manually. A very unique feature of Robot Raconteur is that it sends these service definition files at runtime when a client connects. This means that a dynamic language like Python or MATLAB does not need any a priori information about a service; these languages can simply connect and provide a fully functional interface dynamically. This is *extremely* powerful for prototyping and is the initial motivation for the development of Robot Raconteur.

Beyond the client-service communication, Robot Raconteur has a number of highly useful support features. Robot Raconteur has auto-discovery, meaning that nodes can find each other based on the type of the root object service among other criteria. Authentication is available to control access to the services on a Node. Multi-hop routing is implemented meaning that nodes can be used as routers between a client and service node. This is mainly implemented for access control and transition between different transport technologies. Finally, Robot Raconteur provides exclusive object locks that allow clients to request exclusive use of objects within a service to prevent access collisions (which was a common occurrence in certain past applications).

The core Robot Raconteur library is written in C++ using the Boost[1] libraries and is capable of running on a number of platforms. It has minimal dependencies making it highly portable. Currently RR has been compiled and tested on Windows (x86 and x64), Linux (x86, x86_64, ARM hard-float, ARM soft-float, PowerPC, MIPS), Mac OSX, Android (ARM, x86), and iOS. Language bindings are used to allow access to the library from other languages. Additional supported languages include C#, Java, Python, and MATLAB. LabView can be used through the C# library.

Two additional implementations are available for use in a web browser written in JavaScript and for use in an ASP.NET server written in pure C#. These implementations take advantage of the support for WebSockets[2] over TCP to allow for compatibility with existing web infrastructure. The HTML5/Javascript implementation has been tested with Chrome, Firefox, Internet Explorer, Edge, and Safari. The pure C# implementation (Called RobotRaconteur.NET/CLI) has been tested with IIS and allows RR to run inside a web server.

Robot Raconteur communication is based on a simple message passing system with a well defined format that is simple enough that a fully functional (yet simple) service service has been implemented on an Arduino UNO (2 kB RAM, 32 kB flash) *with full auto-discovery* using a customized version of the core Robot Raconteur functionality. The supported languages and platforms will continue to expand over time so check back frequently.

The rest of this document will provide a tutorial format to introduce the plethora of features available. It serves as the primary overview of Robot Raconteur and as the documentation for the Python library. Other languages will have shorter documents that describe the specific use of Robot Raconteur in that language. While Robot Raconteur is feature-rich and has a learning



Figure 3: Photo of the example robot

curve, it is not necessary to use all the features available in all cases. Once Robot Raconteur is learned, new services can be developed *very* quickly compared to competing technologies, and frequently services have been developed for application in less than an hour that would have otherwise taken days.

Robot Raconteur provides network security through TLS encryption and certificate based authentication. Details on TLS and certificates are covered in the document *Robot Raconteur® Security using TLS and Certificates*.

1.1 Example Robot

This tutorial utilizes a iRobot Create that has been augmented with two webcams, a power converter, and a Raspberry Pi 3 ARM computer. These robots will be available upon request. Figure 3 shows a picture of the robot. Services will be developed in this tutorial to drive the robot and access the webcams. Clients will use the services to control the robot and read the sensors.

1.2 Advantages over roslib

A frequent question about Robot Raconteur is how it is different than ROS. These are just some of the numerous advantages:

- ROS topics provide a stream of single structures, and ROS services provide a single function with a structure argument and a structure return. RR provides a fully object-oriented interface
- Robot Raconteur objects provide a number of “members” that go far beyond ROS and provide powerful two-way communication
- Robot Raconteur allows for sophisticated compound data-types. ROS only has messages which are basic structures.
- Robot Raconteur has wildcard data types.
- Robot Raconteur does not use a “master” node. Connections are made directly between nodes. ROS depends on XML-RPC for master-slave communication which requires XML parsing that limits its use on very low memory devices.
- Robot Raconteur provides detailed information about the service interface on connection by providing “service definition” files to the client. This allows for dynamic generation of interface code by MATLAB and Python at runtime. ROS does not have this capability and it needs much more work to connect to a service.
- Robot Raconteur has automatic node discovery on both IPv4 and IPv6 using UDP broadcast methods.
- Robot Raconteur has exclusive username, session, and mutex locks on objects within a service.
- Robot Raconteur has full IPv6 support
- Robot Raconteur has authentication support
- Robot Raconteur uses an advanced “reactor with thread pool” design that is estimated to have ten times lower latency than ROS
- Robot Raconteur has advanced asynchronous capabilities that allow for massively parallel gather/scatter operations. See Section 11 for more details.
- Robot Raconteur has TLS security and node certificate distribution
- Robot Raconteur has WebSocket support
- Robot Raconteur can run in any modern Web Browser using JavaScript
- Robot Raconteur can run in an IIS ASP.NET Web Server

- Robot Raconteur can run on a USB or PCIe device

2 Service definitions

Example 1 shows the code contained in the “experimental.create.robdef” files. It is a *service definition*. Service definition files are plain text files that describe the *object types* and *value types* (data types). Object types are *references*, meaning that on the client they are simply an advanced reference to the service. Value types are the actual data that are transmitted between client and service. They are always passed by *value*, meaning that they are local to the client or service.

Example 1 Service definition file “experimental.create.robdef”

```
#Service to provide sample interface to the iRobot Create
service experimental.create

option version 0.5

struct SensorPacket
    field uint8 ID
    field uint8 [] Data
end struct

object Create
    option constant int16 DRIVE.STRAIGHT 32767
    option constant int16 SPIN.CLOCKWISE -1
    option constant int16 SPIN.COUNTERCLOCKWISE 1

    function void Drive(int16 velocity , int16 radius)

    function void StartStreaming()
    function void StopStreaming()

    property int32 DistanceTraveled
    property int32 AngleTraveled
    property uint8 Bumpers

    event Bump()

    wire SensorPacket packets

    callback uint8 [] play_callback(int32 DistanceTraveled , int32 AngleTraveled)

end object
```

The first line in the service definition contains the keyword `service` followed by the name of the service type. Version 0.5 introduces the concept of “namespaces” which follow similar rules to Java package names. For experimental software, the name should be prefixed with “experimental”, for example “experimental.create”. For hobbyists and standalone software, the name should be prefixed with “community” and your username, for example “community.myusername.create”, where “myusername” is replaced with your `robotraconteur.com` username. If a domain name for an organization is available it can be used in the same way as Java packages, for example

“com.wasontech.examples.create”. Note that namespaces that use a domain will require signing certificates in the future. Unless you have valid ownership of a domain, “experimental” or “community” should be used.

Next in the service there should be “option version” and the minimum version of Robot Raconteur required to access the service. For now this should be “0.5”. Example 1 does not show it, but there can also be one or more “import” to reference structures and objects in other service definitions. The rest of service definition defines the *structures* and *objects* of the service definition. (Lines starting with “#” are comments.)

2.1 Value types

Value types are the data that are passed between the client and service. Value types can be *primitives*, *structures*, *maps*, or *multidimensional arrays*.

Primitives

Primitives consist of scalar numbers, single dimensional number arrays, and strings. Table 1 contains the primitives that are available for use. Primitive numerical types can be turned into arrays by appending brackets “[]” to the end, for example `int32[]` is an array of 32 bit signed integers. If a fixed size array is desired, a number can be included between the brackets for the desired array size, for example `int32[8]` has a fixed length of 8 integers. If an array is desired that has a maximum size, a “-” sign can be included in the brackets, for example `int32[100-]` can have up to 100 integers. Strings are always arrays so the brackets are not valid. The `void` type is only used for functions that do not have a return value.

Structures

Structures are collections of value types; structures can contain primitives, other structures, maps, or multidimensional arrays. Example 1 shows the definition of the structure `SensorPacket`. A structure is started with the keyword `struct` followed by the structure name. It is ended with the `end struct` keywords. The entries in the structure are defined with the keyword `field` followed by the type, and finally the name of the field. If a structure from a different service definition is used, first the referenced service definition is imported at the top of the service definition and the structure is referenced by the external service definition “dot” the name of the structure.

Maps

Maps can either be keyed by `int32` or `string`. In other languages they would be called “Dictionary”, “Hashtable”, or “Map”. The data is a value type (but not another map). They are created with curly braces. For example, `string{int32}` would be a map of strings keyed by an integer. `string{string}` would be a map of strings keyed by another string. `SensorPacket{string}` and `int32[] {int32}` are also valid examples. `string{int32}{int32}` is *not* valid. There can only be one dimension of keying.

Lists

Lists follow similar rules to maps. They are created with curly braces. For example,

Table 1: Robot Raconteur Primitives

Type	Bytes/Element	Description
void	0	Void
double	8	Double precision floating point
single	4	Single precision floating point
int8	1	Signed 8-bit integer
uint8	1	Unsigned 8-bit integer
int16	2	Signed 16-bit integer
uint16	2	Unsigned 16-bit integer
int32	4	Signed 32-bit integer
uint32	4	Unsigned 32-bit integer
int64	8	Signed 64-bit integer
uint64	8	Unsigned 64-bit integer
string	1	UTF-8 string

`string{list}` would be a list of strings. `SensorPacket{list}` and `int32[] {list}` are also valid examples.

`string{list}{list}` is *not* valid. There can only be one dimension of lists.

Multidimensional Arrays

The multidimensional arrays allow for the transmission of real or complex matrices of any dimension. They are defined by putting a “*” inside the brackets of an array. For example, `double[*]` defines a multidimensional array of doubles. Multidimensional arrays can also have fixed dimensions. For example `double[3,3]` defines a 3x3 matrix. The dimensions are in matrix (column-major) order.

varvalue

In certain situations it may be desirable to put in a “wildcard” value type. The `varvalue` type allows this. Use with caution!

Note: All value types except for scalar numbers and strings can be “null” (NULL, None, etc. depending on language).

2.2 Object types

Objects begin with the keyword `object` followed by the name of the object, and closed with the keywords `end object`. Objects have *members* that implement functionality. Within Robot Racon-

teur there are eight types of members: Properties, Functions, Events, ObjRefs, Pipes, Callbacks, Wires, and Memories . They are defined between `object` and `end object`.

Properties (Keyword: `property`)

Properties are similar to class variables (field). They can be written to (set) or read from (get). A property can take on any value type. A property is defined within an object with the keyword `property` followed by the value type of the property, and finally the name of the property. (All member names must be unique). An example:

```
property double myvar
```

Functions (Keyword: `function`)

Functions take zero or more value type parameters, and return a single value type. The parameters of the functions must all have unique names. The return value of the function may be `void` if there is no return. A function is defined by the keyword `function` followed by the return type, followed by the name of the function. The parameters follow as a comma separated list of parameter type and parameter name. The parameter list is enclosed with parenthesis. An example:

```
function double addTwoNumbers(int32 a, double b)
```

Events (Keyword: `event`)

Events provide a way for the service to notify clients that an event has occurred. When an event is fired, every client reference receives the event. How the event is handled is language-specific. An event is defined similar to a function, however there is no return. The parameters are passed to the client. There is no return. An example:

```
event somethingHappened(string what, double when)
```

Object References (Keyword: `objref`)

A service consists of any number of objects. The *root object* is the object first referenced when connection to a service. The other object references are obtained through the `objref` members. These members return a reference to the specified object. An `objref` is defined by the keyword `objref` followed by the object type followed by the `objref` member name. The object type can be `varobject` to return any valid object type (Use with caution!). The `objref` can also be indexed by number (`[]`, `{int32}`) or by string (`{string}`). This returns a different reference based on the index. It does not return a set of references. An example:

```
objref mysubobj anotherobj{string}
```

If an object from a different service definition is used, first the referenced service definition is imported at the top of the service definition and the object is referenced by the external service definition “dot” the name of the object.

Pipes (Keyword: `pipe`)

Pipes provide full-duplex first-in, first-out (FIFO) connections between the client and service. Pipes are unique to each client, and are indexed so that the same member can handle multiple connections. The pipe member allows for the creation of “PipeEndpoint” pairs. One

endpoint is on the client side, and the other is on the server side. For each connected pipe endpoint pair, packets that are sent by the client appear at service end, and packets that are sent by the service end up on the client side. Packets can be retrieved in order from the receive queue in the “PipeEndpoint”. The type of the packets is defined by the member definition. An endpoint can request a Packet Acknowledgment to be sent once the packet is received by setting “RequestPacketAck” to true. “SendPacket” is used to send packets, and “ReceivePacket” is used to receive the next packet in the queue. “Available” can be used to determine if more packets are available to receive. Pipe endpoint pairs are created with the “Connect” function on the client. Either the client or the service can close the endpoint pair using the “Close” function. A pipe is specified by the keyword `pipe` followed by the packet type, followed by the member name of the pipe. An example:

```
pipe double[] sensordata
```

Callbacks (Keyword: `callback`)

Callbacks are essentially “reverse functions”, meaning that they allow a service to call a function on a client. Because a service can have multiple clients connected, the service must specify which client to call. The syntax is equivalent to the “function”, just replace “function” with “callback”. An example:

```
callback double addTwoNumbersOnClient(int32 a, double b)
```

Wires (Keyword: `wire`)

Wires are very similar to pipes, however rather than providing a stream of packets the wire is used when only the “most recent” value is of interest. It is similar in concept to a “port” in Simulink. Wires may be transmitted over lossy channels or channels with latency where packets may not arrive or may arrive out of order. In these situations the lost or out of order packet will be ignored and only the newest value will be used. Each packet has a timestamp of when it is sent (from the sender’s clock). Wires are full duplex like pipes meaning it has two-way communication, but unlike pipes they are not indexed so there is only one connection per client object reference. The wire allows for a “WireConnection” pair to be created with one “WireConnection” on the client and the other “WireConnection” on the service. Unlike pipes, each wire member can only create one connection pair per client, per service object instance. The “WireConnection” is used by setting the “OutValue” to the current value. This sends the new value to the opposite “WireConnection”, which updates its “InValue”. The same can be reversed. For instance, setting the “OutValue” on the service changes the “InValue” on the client, and setting the “OutValue” on the client changes the “InValue” on the service. It is also possible to receive the “LastValueReceivedTime” and “LastValueSentTime” to read the timestamps on the values. Note that “LastValueReceivedTime” is in the *sender’s* clock, not the local clock and is generated when it is first transmitted. Either the client or the service can close the “WireConnection” pair using the “Close” function.

The wire provides the basis for future real-time communication. (See also Section 9.) An example wire member definition:

```
wire double[2] currentposition
```

Memories (Keyword: `memory`)

Memories represent a random-access segment of numeric primitive arrays or numeric primitive multi-dim arrays. The memory member is available for two reasons: it will break down large read and writes into smaller calls to prevent buffer overruns (most transports limit message sizes to 10 MB) and the memory also provides the basis for future shared-memory segments. An example:

```
memory double[] datahistory
```

2.3 Options

The service definition specification allows for “options”. They are started with the keyword `option` followed by the category. The rest is specific to that option category. Options can contain “hints” to different languages on how to handle the service definition. Options can exist anywhere in the service definition. If they occur within an object or structure they affect that structure or object only.

The only currently supported option category is *constants*. Constants are defined with the keywords `option constant` followed by the constant type, the name, and the value. Constants can only be primitives. Strings are defined with double quotes. Number arrays are defined with brackets with commas between the value (C# style). Numbers can be decimal or in “0x” style hexadecimal format. Examples are:

```
option constant uint32 myconst 0xFB
option constant double[] myarray {10.3, 584.9, 594}
option constant string mystring "Hello world!"
```

2.4 Exceptions

Robot Raconteur will transparently pass exceptions thrown by the receiver to the caller for transactions such as functions, properties, callbacks, and memory reads/writes. Normally these exceptions are of the type `RobotRaconteurRemoteException` which is a universal container for all types of exceptions. In some cases it is useful to have named exceptions that can be passed from receiver to caller and keep their class type. These custom exceptions inherit from `RobotRaconteurRemoteException`. Service definitions can define these exceptions. Exceptions are defined by starting the line with `exception` followed by the name of the exception. For example, the following line will define the exception “MyException” which can then be used in any of the supported languages:

```
exception MyException
```

2.5 Robot Raconteur naming

When naming things like service definitions, objects, structures, and members, certain rules must be followed. The name must consist of letters, numbers, and underscores (_). Names must start with a letter and may not start with any uppercase/lowercase combination of “RobotRaconteur”, “RR”, “get_”, or “set_”. Service names may not end with “_signed”. This is reserved for future use.

3 Robot Raconteur Python

Note: It is best to use the 32-bit version of Python for these examples on Windows.

This document uses Python to demonstrate how Robot Raconteur works, and also serves as the reference for the Python Robot Raconteur library. The examples require that the Robot Raconteur Python library be installed. On Windows, simply download the correct build version and architecture and run the setup program. It will detect your Python installation and place the files in the correct place. On Linux and Mac OSX, download the appropriate tar file and run the following commands in a terminal (replace with the correct tar file for your platform and latest Robot Raconteur version):

```
cd /
```

```
sudo tar xvzf /Downloads/RobotRaconteur-0.5-testing-  
Python.macosx-10.9-intel-py2.7-2014-09-02.tar.gz
```

The examples also require the installation of pySerial (<http://pyserial.sourceforge.net/>). The iRobot examples should work on all platforms using a serial connection or BlueTooth adapter (if installed on the robot). The examples all use NumPy which needs to be installed before running example code (<http://www.scipy.org/Download>). The webcam examples require OpenCV (<http://opencv.org>).

OpenCV can be installed using macports or apt-get, and Windows binaries can be found at <http://www.lfd.uci.edu/~gohlke/pythonlibs/#opencv> . pyGame is a package that is used to access the joystick, and can be found in macports or apt-get, Windows binaries can be found at <http://www.pygame.org/download.shtml> .

NOTE: On the Raspberry Pi (or Ubuntu), you can run the following commands to install NumPy, pySerial, Python OpenCV, pyGame, and access the serial port and webcams:

```
sudo apt-get install python-serial python-numpy python-opencv python-pygame
```

```
sudo usermod -a -G dialout pi
```

```
sudo usermod -a -G video pi
```

Change “pi” to your username in the usermod command.

When using Robot Raconteur in Python, the “thunk” code require to handle different service and data types is handled dynamically so there is no need to generate out extra source code. Instead, the client will receive an object that has all the correct members automatically on connect, and a service simply needs to have the correct functions and properties. How this is accomplished will be discussed through the rest of this document. Python uses “duck typing” so it is not necessary to worry about inheritance or interfaces, the functions and properties just need to exist. A significant advantage of Python’s dynamic typing is Robot Raconteur can generate client interface objects dynamically on connect so a client does not need any prior information about the service it is connecting to.

3.1 Python ↔ Robot Raconteur data type mapping

An important aspect to working with Robot Raconteur is understanding the mapping between Robot Raconteur types and the native types in the language using Robot Raconteur. For Python these are a little more complicated because Python does not have as strong a typing system as other languages. Robot Raconteur can use NumPy (<http://www.scipy.org>) to handle arrays and matrices. While its use is not required it is highly recommended and all examples will use NumPy.

Table 2 shows the mapping between Robot Raconteur and Python data types. For simple arrays, Robot Raconteur expects *column* NumPy arrays of the correct type. Multi-dim arrays are normal NumPy arrays of the correct type. Structures are initialized using a special command in RobotRaconteurNode called `CreateStructure` and will be demonstrated later. Maps are simply `dict` in Python.

3.2 Python Function References

Robot Raconteur frequently uses function references (called function handles or function pointers) to implement callbacks for events and other situations where the library needs to notify the software. In Python, this is accomplished using function references (also called function objects depending on the author). Consider a simple module “MyModule” shown in Example 2.

This example demonstrates that a function reference can be easily made by referencing the function without the argument parenthesis. This method works for module and class functions.

4 iRobot Create Python example

Currently Robot Raconteur is not natively supported by commercial hardware so it is necessary to “wrap” the provided APIs with a Robot Raconteur service. For this example, we are going to wrap the serial Create Open Interface (OI) with a service. The sample code is by no means exhaustive of all the capabilities the robot has to offer; it is intended to be instructive on the use of Robot

Table 2: Robot Raconteur ↔ Python Type Map

Robot Raconteur Type	Python Type	Notes
double, single	float	
int8, uint8, int16, uint16, int32, uint32, int64, uint64	int or long	Map depends on sys.maxint size
double[]	numpy.array	numpy.array([1, 2, ...], dtype='f8')
single[]	numpy.array	numpy.array([1, 2, ...], dtype='f4')
int8[]	numpy.array	numpy.array([1, 2, ...], dtype='i1')
uint8[]	numpy.array	numpy.array([1, 2, ...], dtype='u1')
int16[]	numpy.array	numpy.array([1, 2, ...], dtype='i2')
uint16[]	numpy.array	numpy.array([1, 2, ...], dtype='u2')
int32[]	numpy.array	numpy.array([1, 2, ...], dtype='i4')
uint32[]	numpy.array	numpy.array([1, 2, ...], dtype='u4')
int64[]	numpy.array	numpy.array([1, 2, ...], dtype='i8')
uint64[]	numpy.array	numpy.array([1, 2, ...], dtype='u8')
Multi-dim arrays	numpy.array	Type maps same as array, more dimensions
string	string or unicode	unicode always returned
Map (int32 key)	dict	All keys must be int
Map (string key)	dict	All keys must be string or unicode
List	list	Standard list of expected type
structure	<i>varies</i>	See text for more info
varvalue	RobotRaconteurVarValue	See text for more info

Example 2 Function reference example

```

class myobj(object):
    def hello1(name):
        print "Hello_" + name

def hello2(name):
    print "Hello_" + name

o=myobj()
ref1=o.hello1
ref2=hello2

ref1("John")
ref2("John")

```

Table 3: Members of Create object

Member	Description
<code>function void Drive(int16 velocity, int16 radius)</code>	Drives the create at <code>velocity</code> (mm/s) with <code>radius</code> (mm)
<code>function void StartStreaming()</code>	Starts the sensor packet streaming (Bumpers (17), Distance Traveled (19), Angle Traveled (20))
<code>function void StopStreaming()</code>	Stops the sensor packet streaming
<code>property int32 DistanceTraveled</code>	Total distance traveled (doesn't seem to be accurate...)
<code>property int32 AngleTraveled</code>	Total angle traveled (doesn't seem to be accurate...)
<code>property uint8 Bumpers</code>	Returns the byte with flags about the state of the bumper and wheel drops (See OI manual sensor packet id 7)
<code>event Bump()</code>	Event fired when the bumper goes from no contact to contact
<code>pipe SensorPacket packets</code>	Provides a stream of the raw sensor information as it is received. The ID is always 19. The rest of the packet is the sensor data followed by checksum. The "nBytes" field is not included.
<code>callback uint8[] play_callback(int32 DistanceTraveled, int32 AngleTraveled)</code>	A callback that is called when the "Play" button is pressed and returns notes to play on the Create.

Raconteur. The user is encouraged to fill out the functionality by adding more members to the service definition!

4.1 Simple service

The first step in using Robot Raconteur is to develop an object that implements the service definition. Example 3 shows a non-Robot Raconteur program that contains a class `Create_impl` that implements the service definition in Example 1. Table 3 lists the members and the functionality that will be implemented.

Example 3 shows the members implemented. Properties and functions are simply properties and functions in Python, events are implemented through the `EventHook` class that must be present as a variable in the class. The `Wire` and `Callback` objects are implemented as properties and

Example 3 Initial iRobot Create service without Robot Raconteur code

```
import serial
import struct
import time
import RobotRaconteur as RR
#Convenience shorthand to the default node.
#RRN is equivalent to RR.RobotRaconteurNode.s
RRN=RR.RobotRaconteurNode.s
import thread
import threading

serial_port_name="/dev/ttyUSB0"

class Create_impl(object):
    def __init__(self):
        self.Bump=RR.EventHook()
        self._lock=threading.RLock()
        self._packets=None
        self._play_callback=None

    def Drive(self, velocity, radius):
        with self._lock:
            dat=struct.pack(">B2h",137,velocity,radius)
            self._serial.write(dat)

    def StartStreaming(self):
        pass

    def StopStreaming(self):
        pass

    @property
    def DistanceTraveled(self):
        return 0;

    @property
    def AngleTraveled(self):
        return 0;

    @property
    def Bumpers(self):
        return 0;

    @property
    def packets(self):
        return self._packets
    @packets.setter
    def packets(self, value):
        self._packets=value

    @property
    def play_callback(self):
        return self._play_callback;
    @play_callback.setter
    def play_callback(self, value):
        self._play_callback=value

    def Init(self, port):
        self._serial=serial.Serial(port="/dev/ttyUSB0",baudrate=57600)
        dat=struct.pack(">2B",128,131)
        self._serial.write(dat)

    def Shutdown(self):
        self._serial.close()

def main():

    #Initialize the object in the service
    obj=Create_impl()
    obj.Init(serial_port_name)

    #Drive a bit to show that it works
    obj.Drive(200,1000)
    time.sleep(1)
    obj.Drive(0,0)

    #Shutdown
    obj.Shutdown()

if __name__ == '__main__':
    main()
```

initialized to `None` and will be set by the Robot Raconteur node when the object is exposed as a service. The main function in this example will drive the robot a few feet to demonstrate that the service works. Replace `"/dev/ttyUSB0"` with the appropriate device (COM1, COM2, etc on Windows). The class shown above is mostly a skeleton class that needs to be filled in further to have functionality beyond simply driving.

Notice that the function `drive` has a `with self._lock` block protecting the code within the function. Robot Raconteur is multi-threaded, meaning that all members including functions can be called *concurrently*, meaning that if there is an operation or data structure that can be corrupted by simultaneous access, it is necessary to use a *thread lock*, also-known-as a *mutex*. In the `__init__` for class `Create_impl`, the `self._lock` variable is set to a new instance of `threading.RLock()`. When used with the `with` statement, it will lock itself so only one block can execute at a time with one thread. If all the functions in the class use the same “with lock”, only one thread at a time will be able to access the class. If you are not familiar with multi-threading, it is best to have one global lock for all your functions to prevent collisions.

Now that there is a basic object implemented, it is time to expose it as a Robot Raconteur service. Example 4 shows a replacement for the `main` function that instead of simply driving the robot, exposes the service. Because Python is dynamically typed it is possible to include the service definition as a string constant within the code. The string `create_servicedef` contains the service definition.

Next, it is necessary to create and register a “Transport”. Transports provide the connection between nodes. First we initialize a `LocalTransport` which provides communication within the same computer. First a `LocalTransport` is created and a server is started with a “NodeName” using the function `StartServerAsNodeName`. The “NodeName” in this case is the same as the service name, “experimental.create.Create”.

```
t1=RR.LocalTransport()
t1.StartServerAsNodeName("experimental.create.Create")
```

`StartServerAsNodeName` does a few things that are very important for a service. A “NodeName” can only be used on the same computer one at a time. Each computer generates a unique 128-bit “NodeID” for each “NodeName” registered with a `LocalTransport`. A “NodeID” is a UUID that is globally unique to the service node and is used to identify the node. Because it is a UUID it is (practically) guaranteed to globally uniquely identify the node. By using “StartServerAsNodeName” the “NodeID” is automatically maintained, unique, and static to the Computer/NodeName combination. If a second node attempts to start a local server with the same name an exception will be thrown. If more than one service of the same name needs to be started it is suggested that a “dot” and a number be appended, for instance “experimental.create.Create.1” would be the second instance of the create server node.

The `LocalTransport` that has been registered can be used to communicate within the same computer, but is not able to communicate over the network. For this, the `TcpTransport` is used. `TcpTransport` uses standard TCP/IP sockets for communication. The `TcpTransport` is created and the function `StartServer` is called with a port number. In this case the port number is randomly selected. The transport is then registered with the node for use. The Port Sharer is a small

Example 4 Replacement code for the create server

```
create.servicedef= """
#Service to provide sample interface to the iRobot Create
service experimental.create

option version 0.5

struct SensorPacket
    field uint8 ID
    field uint8[] Data
end struct

object Create
    option constant int16 DRIVE.STRAIGHT 32767
    option constant int16 SPIN.CLOCKWISE -1
    option constant int16 SPIN.COUNTERCLOCKWISE 1

    function void Drive(int16 velocity, int16 radius)

    function void StartStreaming()
    function void StopStreaming()

    property int32 DistanceTraveled
    property int32 AngleTraveled
    property uint8 Bumpers

    event Bump()

    wire SensorPacket packets

    callback uint8[] play_callback(int32 DistanceTraveled, int32 AngleTraveled)
end object
"""

def main():
    #Create Local transport, start server as name, and register it
    t1=RR.LocalTransport()
    t1.StartServerAsNodeName("experimental.create.Create")
    RRN.RegisterTransport(t1)

    #Create Tcp transport, register it, and start the server
    t2=RR.TcpTransport()
    RRN.RegisterTransport(t2)
    t2.StartServer(2354) #random port, any unused port is fine

    #Initialize the object in the service
    obj=Create.impl()
    obj.Init(serial_port.name)

    #Register the service type and the service
    RRN.RegisterServiceType(create_servicedef)
    RRN.RegisterService("Create", "experimental.create.Create", obj)

    #Wait for the user to stop the server
    raw_input('Server started, _press_enter_to_quit...')

    #Shutdown
    obj.Shutdown()

    #You MUST shutdown or risk segfault...
    RRN.Shutdown()

if __name__ == '__main__':
    main()
```

service that allows multiple nodes to listen on port 48653, the standard Robot Raconteur port. This has many advantages and is discussed in detail in Appendix B. To use the Port Sharer, use the command `StartServerUsingPortSharer` instead of `StartServer`.

Once the identification and transports have been initialized, the object is registered for use. The first parameter in `RRN.RegisterService` is the name of the service, the second parameter is the fully qualified Robot Raconteur type of the object, and the last parameter is the object to expose as a service. (Note that a node can have multiple services registered as long as they have different names).

After initialization, the program waits for the user to press “Enter” to stop the server. The Create object is shutdown, and then the Robot Raconteur node is shutdown. This shutdown must be called! The service is now available for use by a client!

4.2 Simple client

While there are several steps to starting a service, connecting as a client is very simple. Example 5 shows a full example that accomplishes the same driving motion as Example 3 but over the network.

Example 5 Simple create client

```
from RobotRaconteur.Client import *

#Connect to the service
obj=RRN.ConnectService("rr+tcp://101.2.2.2?service=Create")

#Drive a bit
obj.Drive(200,1000)
time.sleep(1)
obj.Drive(0,0)
```

The example registers uses the `RobotRaconteur.Client` convenience module to configure for the most common client operations. This module creates a variable “RR” that contains the Robot Raconteur module, and “RRN” that is the default node. It also registers the transports `TcpTransport`, `LocalTransport`, `HardwareTransport`, and `CloudTransport`.

Robot Raconteur uses URLs to connect to services. The most common URLs are for local and TCP cases.

The url format for the `LocalTransport` is:

```
rr+local:///?nodename=TargetNodeName&service=ServiceName
```

and the url format for the `TcpTransport` is:

```
rr+tcp://server:port?service=ServiceName
```

The standard URL format is used, and the target service is passed as part of the “query” portion

of the URL. Often it is necessary to specify the node to connect. For instance, the local transport requires the “nodename” to be specified because there can be multiple nodes running locally. When using the port sharer, it is also necessary to specify the target node (See Appendix B). The target node can be identified by NodeName, by NodeID, or by both. The NodeID should be the UUID of the node without curly braces. This is due to the limitations of URL syntax.

For instance, these are all valid URLs for the local transport to connect to the CreateService (replace the UUID with the one generated for your service):

```
rr+local:///?nodename=experimental.create.Create&service=Create
```

```
rr+local:///?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&service=Create
```

```
rr+local:///?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&nodename=experimental.create.Create&service=Create
```

The following are valid URLs to connect to the CreateServer using tcp:

```
rr+tcp://localhost:2354/?service=Create
```

```
rr+tcp://localhost:2354/?nodename=experimental.create.Create&service=Create
```

```
rr+tcp://localhost:2354/?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&service=Create
```

```
rr+tcp://localhost:2354/?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&nodename=experimental.create.Create&service=Create
```

Replace “localhost” with the IP address or hostname of a foreign computer if accessing over a network.

Note that for the TCP connections, the “rr+tcp” can be connected to “rrs+tcp” to enable TLS to encrypt the communication. See the *Robot Raconteur® Security using TLS and Certificates* manual for details on using TLS.

See Appendix A for details on how to use URLs for more advanced cases.

A reference to the service object is returned, and it can now be used to access the members. In this example, the robot is driven a bit to demonstrate how to use a function. Finally, the `Shutdown` command is called on the node to shut down Robot Raconteur. This *must* be called before exiting the program.

4.3 iRobot Create Service

The initial service shown in Example 4 only fills in the `Drive` member. Appendix D.1 shows a complete service that fills in all of the members. This is not intended to be exhaustive for the full features of the iRobot Create; it is instead intended to be used to demonstrate features of Robot Raconteur. Because of the length of the code it is printed in the appendix and will be referred to throughout this section.

The functions `StartStreaming` and `StopStreaming` start and stop a thread that receives data from

the serial port and transmits the data to the `Bump` event, the `packets` pipe, or the `play_callback` where appropriate. The `StartStreaming` and `StopStreaming` functions also send commands to the Create robot to start or stop sending the data. The function `_recv_thread` implements the ability to receive and parse the packets. This function is dedicated to handling the serial data from the robot and calls the `_fire_Bump` function to fire the `Bump` event, the `_SendSensorPacket` function to set the new value of the `packets` wire, or the `_play` function to handle when the Play button is pressed on the robot. It also keeps a running tally of distance and angle traveled in the `_DistanceTraveled` and `_AngleTraveled` variables. The rest of this section will discuss the implementation of the different members. It stores the `Bump` data in the `_Bumpers` variable.

The `Bumpers`, `DistanceTraveled`, and `AngleTraveled` properties are implemented as standard Python properties using the `@Property` decorator. Because these are read only, the setters throw an exception. Properties transparently transmit exceptions back to the client. Functions also transparently transmit exceptions to the client. All Robot Raconteur calls should be surrounded with `try/except` blocks that catch `Exception` meaning it will catch and process any thrown exception.

Events in Python are implemented using the `EventHook()` class. The `__init__` function of `Create_impl` sets:

```
self.Bump==RR.EventHook()
```

This line creates the `EventHook` object that is used to connect events. The `fire_Bump` function then fires this event. The Robot Raconteur node will transmit this event to all connected clients. Note that the `fire` command of `EventHook` may contain parameters if the event has parameters.

The `packets` wire is implemented by the `packets` property in `Create_impl`. A *Wire Server* is passed to the object when it is registered as a Robot Raconteur service object. The setter of the `packets` property stores the wire server in the `_packets` variable, and then sets the `WireConnectCallback` to `self._packet_wire_connected`. (See section C.8.) `_packet_wire_connected` will now be called when a client connects to the wire. This function will receive a `WireConnection` object as the parameter stored in `wire`. Because there can be multiple clients connected to the same service, it is possible to have multiple `WireConnections` for the same wire. The `_packet_wire_connected` function stores the `WireConnection` in a dictionary by the `Endpoint`, which is the Robot Raconteur endpoint of the current client. This uniquely identifies the client for the `WireConnection`. (See section C.8.)

The `_SendSensorPackets` function is called by the serial receive thread when there is a new data packet. The `_SendSensorPackets` loops through all the connect `WireConnections` for `packets` and sets the `OutValue` to the current packet. The packet data is stored in a `experimental.create.SensorPacket` structure that is defined in the service definition. The `RRN.NewStructure` command is used to initialize a new Robot Raconteur structure in Python. If there is an error, assume that the wire has been closed and delete it from the dictionary. Note that wires are *full duplex*, meaning that there is an `InValue` and `OutValue` available at both ends of the wire to set and read a value. There is also the `LastValueReceivedTime` and `LastValueSentTime` to determine the last time that values were updated.

The `play_callback` member is implemented by the `play_callback` property using a Python prop-

erty. The property is set when the object is registered like wires; the setter stores it in the `_play_callback` variable. The `_play` function demonstrates how to use the callback. The `StartStreaming` command contains the following line:

```
self._ep=RR.ServerEndpoint.GetCurrentEndpoint()
```

This line is used to determine the “endpoint” of the current client that is calling the function. The endpoint is used to uniquely identify the client. When a callback is used, it is necessary to specify which client to call because there may be multiple connected clients. The client is identified using the endpoint. The `_play` function contains the following lines, which executes the callback on the client:

```
cb_func=self.play_callback.GetClientFunction(self._ep)

notes=cb_func(self.DistanceTraveled, self._AngleTraveled)
```

The first line retrieves the a function handle to call the client based on the stored endpoint. The second line executes this function, which is actually implemented by calling the client with the supplied parameters and then returning the result. Note that exceptions are also transmitted transparently by callbacks from the client to the service. (See section C.7.)

The `main` function is nearly identical to the previous example except for the addition of the following line:

```
t.EnableNodeAnnounce()
```

This line initializes the auto-discovery system to send out beacon packets so that client nodes can find the service. This process is discussed in Section 6.

4.4 iRobot Create Client

A client that utilizes the full iRobot Create Service is shown in Appendix D.2. The client is similar to the previous example client, however it adds functionality using the `Bump`, `packets`, and `play_callback` member. The line:

```
c.Bump += Bumped
```

adds the function `Bumped` as a handler when the event is fired. The line:

```
wire=c.packets.Connect()
```

connects to the `packets` wire and returns a `WireConnection` object that is stored in the `wire` variable. This `WireConnection` has the same functionality as the one provided to the service object in the previous section. In this example, the `WireValueChange` event is used. The line:

```
wire.WireValueChanged+=wire_changed
```

adds the `wire_changed` function as a handler and is called when the service provides a new value

for the wire. This event is also available on the service however in this application it is not needed. The final step in the configuration is to set the function `play_callback` as the callback function for the `play_callback` member through the following line:

```
c.play_callback.Function=play_callback
```

This function will now be called by the service when the service calls this client's callback.

After the setup the robot is driven a bit and then pauses to allow the user to try out the functionality, and then Robot Raconteur is shutdown.

5 Webcam Example

5.1 Webcam Service

The example robot also has webcams that can be accessed using the Python OpenCV libraries. Appendix D.3 contains the listing of a program that exposes the webcams as a Robot Raconteur service. The example is intended to demonstrate the usage of the “objref”, “pipe”, and “memory” members that were not used in the iRobot Create examples.

The service definition for the `experimental.createwebcam` shown in Example 6 contains two objects:

`WebcamHost` and `Webcam`. The `Webcam` object type represents a single camera, and the `WebcamHost` object allows for the client to determine the number of webcams and retrieve the `Webcam` objects through an “objref” member.

The class `WebcamHost_impl` implements the `WebcamHost` object type. The function `WebcamNames` returns a map of the indexes and names of the cameras, and is an example of the `string{int32}` Robot Raconteur type. The function `get_Webcams` implements the `Webcams` objref. Note that the objref is implemented by prepending “get_” to the name of the objref member. The index may come as a `string` even though an `int32` is expected, so convert the type to `int` before using. When returning an object from an objref, it is necessary to return the fully qualified Robot Raconteur type of the object as a second parameter.

Note: objects can only be registered as a service object ONCE. Objects cannot be returned by two separate objrefs. Objrefs must form a “tree” structure, where the child branches are the return objects from objrefs.

The `Webcam_impl` object implements the webcam functionality. The `CaptureFrame` function returns a single frame to the client. The `StartStreaming` and `StopStreaming` functions begin or stop a thread implemented by the `frame_threadfunc` function that sends streaming frames to the connected clients through the `FrameStream` pipe.

Pipes are very similar to wires, and are implemented using Python properties in a similar way. The `FrameStream` property setter sets the `PipeConnectCallback` function to

Example 6 Service definition file “experimental.createwebcam.robdef”

```
#Service to provide sample interface to webcams
service experimental.createwebcam

option version 0.5

struct WebcamImage
    field int32 width
    field int32 height
    field int32 step
    field uint8[] data
end struct

struct WebcamImage_size
    field int32 width
    field int32 height
    field int32 step
end struct

object Webcam
    property string Name
    function WebcamImage CaptureFrame()

    function void StartStreaming()
    function void StopStreaming()
    pipe WebcamImage FrameStream

    function WebcamImage_size CaptureFrameToBuffer()
    memory uint8[] buffer
    memory uint8[*] multidimbuffer

end object

object WebcamHost
    property string{int32} WebcamNames
    objref Webcam{int32} Webcams
end object
```

`self.FrameStream_pipeconnect`. The `FrameStream_pipeconnect` function adds the passed `PipeEndpoint` to the dictionary of connected `PipeEndpoints`. While a wire can only have one `WireConnection` client/server pair per client, pipes can have “indexed” `PipeEndpoints` meaning a single client can have multiple `PipeEndpoint` client/server pairs per client. They are “indexed”, meaning a `PipeEndpoint` is defined by the Robot Raconteur client endpoint (not to be confused with the `PipeEndpoint`) and the index of the `PipeEndpoint`. (See Section C.5 and Section C.6.)

The function `frame_threadfunc` demonstrates looping through the connect `PipeEndpoints` and transmitting a thread on each connected endpoint. If there is an error, it is assumed the `PipeEndpoint` has been closed and the endpoint is deleted.

The final members of interest in the `WebcamService` are the two memories, `buffer` and `multidimbuffer`. These two members demonstrate how to use two flavors of memories that are either single dimensional or multi-dimensional. Memories are useful when data greater than about 10 MB needs to be transmitted between client and server, when there is a random-access block of memory, or in the future for shared memory applications. The function `CaptureFrameToBuffer` captures the data and saves it to the buffers. Note that multi-dimensional arrays in Python are simply multi-dimensional NumPy arrays. Some processing is done to place the data in “Matlab” style image formats. A structure of type “`experimental.createwebcam.WebcamImage_size`” is returned to tell the client how big the image is.

The two memories are implemented using properties. The `buffer` member returns an `ArrayMemory` object, and the `multidimbuffer` returns a `MultiDimArrayMemory` object. Both contain their respective array and multi-dimensional array. In this example a new memory object is returned every time. This is not generally the best way to use the memory; instead, a persistent memory object should be used with a persistent memory block. (See sections C.11 and C.12.)

5.2 Webcam Client

Appendix D.4 lists a program that will read the webcams and display the images. The initialization, connection, and shutdown are similar to the iRobot Create example. The main difference is the use of the “objrefs”, which are used to get references to the webcams `c1` and `c2`:

```
c1=c_host.get_Webcams(0)
```

```
c2=c_host.get_Webcams(1)
```

The rest of the program deals with OpenCV related functions to show the images.

5.3 Webcam Client (streaming)

Appendix D.5 lists a program that provides a “live” view of the camera, although depending on the speed of the computer it may be fairly slow because Python is an interpreted language. The program connects and retrieves the webcam object reference `c` the same way as the previous

example, and the connects to the pipe `FrameStream`. The pipe index is given as the argument, and -1 means *any index*.

```
p=c.FrameStream.Connect(-1)
```

Next, a callback is added so that the function `new_frame` will be called when a new pipe packet arrives.

```
p.PacketReceivedEvent+=new_frame
```

This function will be called from a different thread by Robot Raconteur when a new frame has arrived. In the `new_frame` function, the variable `current_frame` is updated with the new value. The `Available` property in the `PipeEndpoint` provides the number of packets waiting, and the `ReceivePacket` retrieves the next packet. Packets always arrive in order.

The rest of the program handles showing the images as they arrive and shutting down, including closing the pipe.

```
p.Close()
```

5.4 Webcam Client (memory)

Appendix D.6 demonstrates the use of the memories. The memories have functions `Read` and `Write` that allow for a segment of the memory to be read or written into or from a buffer. The memory position, buffer, buffer position, and count are passed. For multi-dimensional arrays, the memory position, buffer position, and count are lists. The `ArrayMemory` has the special property "Length" for the length of the array, and the `MultiDimArrayMemory` has the special properties "Dims", "DimCount", and "Complex".

6 Service auto-discovery

A powerful feature of Robot Raconteur is the ability to detect services automatically. Each transport has a method to broadcast what services are available. For the `TcpChannel` this is accomplished through broadcast UDP packets. Other transports will use the method most appropriate for the transport technology.

The first step in the auto-discovery process for TCP is for the node containing the service to broadcast an announcement that the node is available. This broadcast is sent every five seconds using UDP multicast. The packet contains the `NodeName`, `NodeID`, and a URL to connect to the "Service Index", which is a special service that lists the services registered in the node. The client will interrogate the service nodes it has discovered to determine the available services. This is shown in the diagram by the client requesting the available services, and the service node returns the available services. The "Service Index" is registered automatically by the node and does not require any extra work by the user.

Using auto-discovery requires a few more function calls as were shown in the examples. The first step is to enable the auto-discovery announce for the service and the auto-discovery listening for the client. This is done through the transport. For the client, use:

```
t=RR.TcpTransport()
t.EnableNodeDiscoveryListening()
```

For the service, use:

```
t=RR.TcpTransport()
t.EnableNodeAnnounce()
```

To find a service, use the command:

```
res=RRN.FindServiceByType("experimental.create.Create",
    ["rr+local","rr+tcp","rrs+tcp"])
```

where “experimental.create.Create” is replaced with the fully qualified type being searched for and the second parameter is a list of the transport types to search. `res` is a list of `ServiceInfo2` structures that contains the `NodeID`, `NodeName`, `Name`, `RootObjectType`, `RootObjectImplements`, `ConnectionURL` (list), and the `Attributes`. The attributes entry is type `varvalue{string}` but should only contain type `string` and numeric entries. This is used to help identify the correct service to connect to. Service attributes are set through the `ServerContext` object that is returned when a service is registered. A short example:

```
context=RRN.RegisterService
    ("Create","experimental.create.Create",obj)

attributes={"RobotName" : RR.RobotRaconteurVarValue("Create1","string")}

context.SetServiceAttributes(attributes)
```

7 Authentication

Robot Raconteur provides a built-in authentication system. During connection, a client can specify a “username” of type `string`, and “credentials” of type “`varvaluestring`”. Normally the credentials contains a simple string entry for the password, but some authentication methods may require more complex credentials like a key/password pair. A connection example:

```
credentials={"password", RR.RobotRaconteurVarValue("mypassword","string")}

obj=RRN.ConnectService(
    "rr+tcp://localhost:2354?service=Create","myusername",credentials)
```

Of course the current example service does not have authentication enabled. The first step is to

create a `UserAuthenticator`. The `UserAuthenticator` will receive the authentication requests and compare the username and credentials. If they are correct, the user is authenticated. The only authenticator currently available is the `PasswordFileUserAuthenticator`. This authenticator uses a plain-text string that contains the user definitions, one per line. There are three entries separated by spaces: the username, the MD5 hash of the password, and a comma separated list of credentials (no spaces between). The two credentials of interest are “objectlock” and “objectlockoverride”. The meaning of these is discussed in Section 8. Example 7 shows the contents of a simple password file. The MD5 hash for the password can be generated using “RobotRaconteurGen”.

```
RobotRaconteurGen --md5passwordhash mypassword
```

Example 7 Example password file for `PasswordFileUserAuthenticator`

```
myusername 34819d7beeabb9260a5c854bc85b3e44 objectlock
anotherusername 1910ea24600608b01b5efd7d4ea6a840 objectlock
superuser f1bc69265be1165f64296bcb7ca180d5 objectlock,objectlockoverride
```

The `PasswordFileUserAuthenticator` can now be initialized:

```
with open('passwords.txt') as content_file:
```

```
    content=content_file.read()
```

```
p=RR.PasswordFileUserAuthenticator(content)
```

The next step is to create a `ServiceSecurityPolicy` that describes the security requirements. The policy contains the authenticator and a dictionary of policies. Currently only “requirevaliduser” and “allowobjectlock” are valid, and both should be set to “true”.

```
policies={"requirevaliduser" : "true", "allowobjectlock" : "true"}
```

```
s=RR.ServiceSecurityPolicy(p,policies)
```

Finally, the service can be registered using the policy.

```
RRN.RegisterService("Create","experimental.create.Create",obj)
```

When the service is running, it may be useful to determine if there is a currently authenticated user. This is accomplished through the `ServerEndpoint` class static method. The current `AuthenticatedUser` can be retrieved:

```
user=RR.ServerEndpoint.GetCurrentAuthenticatedUser()
```

Note that this call will raise an exception if no user is currently authenticated. The `AuthenticatedUser` contains the fields `Username`, `LoginTime`, `LastAccessTime`, and `Privileges` fields to help determine the user currently accessing the service. This function will work during all transactional calls to the service.

The authenticated user will be logged out when the client connection is closed or after a timeout

of typically 15 minutes.

8 Exclusive object locks

During the first applications of the experimental version of Robot Raconteur there was a frequent problem of multiple users trying to access a device remotely at the same time and causing confusing collisions. It became rapidly apparent that some form of locking needed to be available. Robot Raconteur has three types of locks "User", "Client", and "Monitor".

8.1 User

The "User" lock provides a lock on an object within a service that is exclusive to a specific username. The user must be authenticated and have the "objectlock" privilege. The same user can authenticate multiple times from any location using this lock. The lock works on the selected object, and all objects below the current object in the "objref" tree. (This means all objects that are referenced by the locked object's objrefs.) To lock an object, use:

```
RRN.RequestObjectLock(obj, RR.RobotRaconteurObjectLockFlags_USER_LOCK)
```

obj must be a Robot Raconteur client object reference. It does not have to be the root object. This function will raise an exception if the object is already locked by another user. To release the lock, use:

```
RRN.ReleaseObjectLock(obj)
```

If the user has the privilege "objectlockoverride" the user can release all locks even if the user did not initiate the lock.

Note that the lock will prevent transactional operations from occurring, but will not stop wire connections, pipe endpoints, and events from functioning normally. If exclusive wire and pipe connections are required conflicting wire and pipe connections will need to be closed by the service object.

8.2 Client

The "Client" lock is identical to the "User" lock but only allows one unique connection. This means that the user cannot access the same service object from a different connection even with the same username. To request a client lock:

```
RRN.RequestObjectLock(obj, RR.RobotRaconteurObjectLockFlags_CLIENT_LOCK)
```

To release a client lock:

```
RRN.ReleaseObjectLock(obj)
```

8.3 Monitor

The “Monitor” lock provides a global thread monitor (more often called “mutex”) lock on a single object. This means that globally only the current thread with the lock can access the object. The “Monitor” lock is intended for short operations that cannot be interrupted and will timeout if 15 seconds elapses between operations on the client. It does not inherit to other objects like “User” and “Client” locks. Unlike “User” and “Client”, the object must implement the monitor locking functionality explicitly. (Note that the “RLock” type in Python does not support locking with timeout which makes things a little less clear.) An example object that implements the required functionality:

```
class MyMonitorLockableObject:

    def __init__(self):

        self._lock=threading.RLock()

    def RobotRaconteurMonitorEnter(self):

        self._lock.acquire()

    def RobotRaconteurMonitorEnter(self,timeout):

        self._lock.acquire()

    def RobotRaconteurMonitorExit(self):

        self._lock.release()
```

To request a monitor lock on the client side, use:

```
RRN.MonitorEnter(obj)
```

To release the lock, use:

```
RRN.MonitorExit(obj)
```

9 Time-critical software with Wire member

The “Wire” member is a unique feature of Robot Raconteur that is designed to transmit a constantly changing value. This is intended to emulate a physical wire carrying an analog value, but is capable of carrying any valid Robot Raconteur data type. As discussed in Section 2.2, the “wire” is full-duplex meaning that it can send data in both directions, and it only provides the latest value. When the `OutValue` of one `WireConnection` is changed, a packet is generated that contains the

data and a timestamp of type `TimeSpec`. This packet is transmitted through the channel and received by the other `WireConnection` in the pair. If the timestamp is newer, the `InValue` is updated. The timestamps can also be read through `LastValueReceivedTime` and `LastValueSentTime`.

Unlike previous versions of Robot Raconteur, from version 0.5 onward the `Wire` member is **non-blocking** meaning that the `OutValue` is set, the new value will be placed in the send queue and control will return immediately to the caller. If the an older value exists it will be discarded and replaced with the new value.

The `LastValueReceivedTime` property can be used to detect how old the `InValue` data is. The `TimeSpec` returned is in the **remote** node's clock. This means that it cannot be compared directly to the local node clock. Clock synchronization is not directly supported by Robot Raconteur. Different transports may provide this functionality.

Robot Raconteur will in the future add real-time “sideband” transports for `Wire` connections. This will be implemented using shared memory, USB, real-time Ethernet, UDP, or any other available transport capable of deterministic communication. The exact methods for real-time wires will be discussed in a future application note when it is available.

10 Forward compatibility with the “implements” statement

The “implements” statement is a feature that is intended to help with future compatibility and variation of service object types. The concept behind “implements” is to state that one object type “covers” another object type. This means that it contains all the members with matching names, types, and parameters. Unlike other languages like Python there is no implicit inheritance of members; each member must be specified in the object that contains the implements statement. The idea behind the implements statement is to allow a client to use a service object that has more features than it understands. For example, consider a simple service “Light” that contains an object with one member, “Power” which can either be 0 or 1. Now consider another service “AdvancedLight” with an object that contains two members, “Power” and “Dim” but implements “Light”. A client that understands “Light” can still use “AdvancedLight” because it implements the simpler “Light”. This is less of an issue in Python because there is no explicit typing, but in other languages this can become very important. The implements statement must form a clear hierarchy; there cannot be circular implements. Implements can be used with the “import” statement the same way that structures and objrefs can work with the “import” statement.

11 Asynchronous programming

Most of the functions in Robot Raconteur are “blocking” functions, meaning that they will block the current executing thread until the result of the operation is completed. An example is `RRN.ConnectService()`. This function will begin the connection process and block the current thread until the connection is complete. This process can take anywhere from a few hundred

milliseconds to several seconds. If the client is only accessing one device this is normally not a problem, but if the client needs to connect to a hundred devices this can become a severe problem as having a large number of threads becomes very computationally expensive and difficult to coordinate. The solution to this problem is to use “asynchronous functions”. These functions begin the operation but return immediately. When the operation is completed, a supplied handler function is called by the thread pool. See Example 8.

Example 8 Example asynchronous invocation

Consider the synchronous connect function used in the previous examples:

```
{c=RRN.ConnectService('rr+tcp://localhost:2354?service=Create')}
```

The asynchronous equivalent would be:

```
def connect_handler(c,err):
    if (err is not None):
        # If "err" is not None it means that an exception occurred.
        # "err" contains the exception object
        print "An_error_occured!" + str(err)
        return
    print "Got_the_connection!"
    # Now "c" is ready for use

# Start the connect process with a 5 second timeout
c=RRN.AsyncConnectService('rr+tcp://localhost:2354?service=Create',None,None,None,connect_handler
,5)
# Do other tasks while connection is being created
```

The form of “Async” functions is normally the same as synchronous functions but with “Async” prepended and two extra parameters: the handler function and the timeout. The handler function will take zero, one, or two arguments depending on the return arguments. The handler function can be any free module function or a bound instance method. The form of the handlers for each function are described in Section C. The last argument is the timeout, which is in seconds. (Note: other language bindings use milliseconds for the timeout). The default for most is `RR.TIMEOUT_INFINITE` which means the function will never timeout. This is not recommended as it can cause a deadlock. Always specify a timeout.

For object references created by the `RRN.ConnectService()` function, functions, properties, and objrefs are available in asynchronous form. In general these functions operate the same as their standard synchronous counterparts but are prepended by “async.” and have two extra parameters, “handler” and “timeout”. The property forms use getter and setter functions of the form `async_get_*` and `async_set_*`. If the function produces a return value, the handler will have the form `handler(ret, err):.` For void functions, the handler will have the form `handler(err):.`

See Section D.7 for an example of how these functions work in practice.

12 Gather/Scatter operations

In many situations it is necessary to query data from dozens or hundreds of sensors. Thanks to the asynchronous functionality discussed in Section 11 this is possible and is limited only by memory and network bandwidth. An example of a practical application of gather/scatter operations is that of an advanced lighting system that needs to rapidly query large numbers of sensors, make control decisions, and then distribute updated commands to a large number of lighting fixtures. Consider a list of connections that have already been connected and are stored in a variable `c_list`. Each connection is to a service that has the function `ReadSensor()` that returns some important data. Example 9 will query each sensor concurrently and call the handler when all the sensors have been queried.

13 Debugging Python with Eclipse PyDev

Robot Raconteur uses a thread pool with a default 20 threads that responds to network activity, timers, asynchronous returns and events. When the Robot Raconteur library calls back to user code it is normally executed by a thread pool thread. Because of this multi-threaded behavior, any debugger used must support multithreading. Currently the best Python debugger with multithreading support is Eclipse with the PyDev plugin. As of version 0.5 Robot Raconteur fully supports this debugger. You may see a warning about `sys.settrace` being called by `RobotRaconteurPythonUtil`. This warning can be safely ignored.

14 PipeBroadcaster and WireBroadcaster

Pipes and wires are frequently used as “broadcasters”, meaning that the same data is sent to each active client endpoint/connection that the clients have created. The examples demonstrate a primitive way to use pipes and wires as broadcasters, however these are overly simplified and don’t necessarily take into account the true machinery necessary to implement a broadcaster. To solve this, `PipeBroadcaster` and `WireBroadcaster` have been added. See Appendix C.24 and Appendix C.25 for more details.

The `WireBroadcaster` is initialized by passing the wire that is set to the service object during initialization. It is then simply used by setting the `OutValue` property. The supplied `OutValue` will be transmitted to all connected clients. See Appendix D.8 for an example.

The `PipeBroadcaster` is initialized by passing the pipe that is set to the service object during initialization, and a parameter called “backlog”. Backlog is used as a flow-control measure to avoid clogging the transports with packets. If “backlog” is set to -1, no packets will be dropped. If “backlog” is greater than zero, packets will be dropped on transmit if more than the specified number of packets is “in flight” on the way to the client. The packets that are transmitted will still arrive in order. The reason for this capability is to allow for functionality by transmitting video

Example 9 Example of a gather/scatter operation

```
global_err=[]
global_data=[]
ev=threading.Event()

def read_finished(data, err):
    # "data" now contains a list of data
    # "err" contains a list with each element containing "None" or the exception that occurred for that read

    # Store data in global variables
    global global_err, global_data
    global_err=err
    global_data=data

    # Notify "main()" that the read is complete
    ev.set()

# c_list contains a list of connections created with RobotRaconteur.s.ConnectService
def start_read(c_list, handler, timeout):
    N=len(c_list)
    keys=[]
    keys_lock=threading.Lock()
    ret=[None]*N
    err=[None]*N

    def h(key, d, erri):
        done=False
        with keys_lock:
            if (erri is not None):
                err[key]=erri
            else:
                ret[key]=d
                keys.remove(key)

            if (len(keys)==0): done=True

        if (done):
            handler(ret, err)

    with keys_lock:
        for i in xrange(N):
            try:
                c_list[i].async_ReadSensor(funcutils.partial(h, i), timeout)
                keys.append(i)
            except Exception as erri:
                err[i]=erri

        if (len(keys)==0):
            raise Exception("Could not read any sensors")

def main()

    # Create all the c_list connections here

    # Start the read with a 100 ms timeout
    start_read(c_list, read_finished, 0.1)

    # Wait for completion
    ev.wait()

    # Do something with the results
    print global_data
    print global_err
```

frames where it is acceptable if some frames are dropped, but they still are a “streaming” data type. A typical value is 3 for “backlog”. Once created `SendPacket` and `AsyncSendPacket` can be used to send packets. Note that the handler for `AsyncSendPacket` has the form `handler()` :, meaning it has no arguments.

15 Conclusion

This documents serves as the introduction and primary reference for the use of Robot Raconteur. It also serves as the reference for the Python language bindings. For other languages, supplemental documents are provided that explain how to use Robot Raconteur in specific languages.

Table 4: Supported URL Schemes

Scheme	Transport	Description
rr+local	LocalTransport	Local connection within the same computer
rr+tcp	TcpTransport	TCP connection without encryption
rrs+tcp	TcpTransport	TCP connection with TLS encryption
rr+ws	TcpTransport	WebSocket connection without encryption
rrs+ws	TcpTransport	WebSocket connection with Robot Raconteur encryption
rr+wss	TcpTransport	WebSocket connection with HTTPS encryption
rrs+wss	TcpTransport	WebSocket connection with HTTPS encryption and Robot Raconteur encryption
rr+usb	HardwareTransport	Connection to local USB device
rr+pci	HardwareTransport	Connection to local PCI or PCIe device
rr+industrial	HardwareTransport	Connection to device on local fieldbus
rr+cloud	CloudTransport	Connection to node using Robot Raconteur cloud
rr	CloudTransport	Same as rr+cloud

A URL Format

Robot Raconteur uses URLs to specify how and where to connect to a service. The URLs follow this basic format:

scheme://host:port/path/to/endpoint/?nodeid=NodeId&nodename=NodeName&service=ServiceName

The italic letters are replaced with their actual values. This is the full format. The host, port, path-to-file, NodeID, and NodeName are all optional depending on the transport. For instance, a full URL to the create robot would be:

```
rr+tcp://localhost:2354/?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c
&nodename=experimental.create.Create&service=Create
```

Note that due to the limits on the URL format, the braces should not be include in the NodeID. The “scheme” specifies what transport method should be used to connect to the client. Schemes start with “rr+” for unsecured transport, and “rrs+” for secure transports using TLS. See the *Robot Raconteur® Security using TLS and Certificates* manual for details on using TLS. Table 4 lists the available schemes and which transports they use.

Local URLs

Local connections are made using the LocalTransport. The URL must include a NodeName or NodeID to specify the desired node. The hostname, port, and path-to-file must be omitted. Examples:


```
rr+local:///?nodename=experimental.create.Create&service=Create
```

```
rr+local:///?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&service=Create
```

```
rr+local:///?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&nodename=experimental.create.Create&service=Cr
```

TCP URLs

TCP connections are made using the TCP transport. The URL must include the hostname to connect to. The hostname can be either “localhost”, a hostname (example.com), IPv4 address with four numbers separated by dots (101.2.2.2), or an IPv6 address in brackets ([fe80::d022:37ee:feee:aab2]). The port must be included if it is not 48653. If it is 48653, the port number can be omitted. If the port sharing service is being used, a NodeName or NodeID must be specified. Otherwise they can be omitted. Examples:

```
rr+tcp://localhost:2354/?service=Create
```

```
rr+tcp://example.com:2354/?service=Create
```

```
rr+tcp://101.2.2.2:2354/?service=Create
```

```
rr+tcp://[fe80::d022:37ee:feee:aab2]/?service=Create
```

```
rr+tcp://localhost:2354/?service=Create
```

```
rr+tcp://localhost:2354/?nodename=experimental.create.Create&service=Create
```

```
rr+tcp://localhost:2354/?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&service=Create
```

```
rr+tcp://localhost:2354/?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&nodename=experimental.create.Creat
```

```
rr+tcp://101.2.2.2/?nodeid=6f6706c9-91cc-d448-ae8c-c5a2acac198c&service=Create
```

Using “rrs+tcp” instead of “rr+tcp” will enable encrypted communication using TLS.

WebSocket URLs

The WebSocket URL is used to connect to a HTTP server that is running a Robot Raconteur node. This can be accomplished using IIS with RobotRaconteur.NET/CLI. There are four supported schemes, rr+ws, rrs+ws, rr+wss, and rrs+wss, each using a different level of encryption (See Table 4). The URL used is the same format as any other WebSocket or HTTP link, except the change in scheme. The query is used to specify which service to connect. Example:

```
rr+wss://example.com/api/robotraconteur?service=Create
```

Hardware Devices

Hardware is supported through the use of the *Robot Raconteur Hardware Service*, a small service that runs in the background and manages hardware connections. Supported schemes are rr+usb, rr+pci, and rr+industrial. All URLs to hardware must contain a NodeName or a NodeID.

```
rr+usb:///?nodename=myusbdevice&service=DAQ
```

```
rr+pci:///?nodename=mypcidevice&service=DAQ
```

```
rr+industrial:///?nodename=mycandevic&service=DAQ
```

Cloud Connection

The Robot Raconteur® Cloud Transport uses the Robot Raconteur cloud servers to create connection between nodes over the Internet when a direct connection may not be possible. To create a connection, use the username as the hostname. Specify the NodeId or NodeName of the target node. This transport requires that the Cloud Client is running and connected on the same computer. The Cloud Client will become available in late 2016.

```
rr://myusername?nodename=experimental.create.Create&service=Create
```

B Port Sharer

The Port Sharer is a small service that is installed and runs in the background. It listens on port 48653, the official Robot Raconteur port. A node can use the service by calling `StartServerUsingPortSharer` on a `TcpTransport` instance. The node will then register with the service based on its `NodeName` and `NodeId`. The `NodeId` is normally automatically assigned when `StartServerAsNodeName` is called on the `LocalTransport`. When a client connects, it must include either the “nodename” or “nodeid” in the connection URL, as described in Appendix A. Because the `NodeName` is not guaranteed to be unique, it is recommended that the `NodeId` be used when possible.

The Port Sharer can be downloaded from the website located at <http://robotraconteur.com/download>.

Installation:

Windows: Double click on the msi installer and follow the instructions. You will need administrator access.

Mac OSX: Double click on the dmg file, and then double click on the pkg file. Follow the instructions. You will need administrator access.

Linux: The port sharer is distributed as a deb file. It is compatible with Debian, Ubuntu, Raspian, Linux Mint, and any other Debian based distribution. The deb file is for all architectures. To install, run:

```
sudo dpkg -i robotraconteurportsharer_1.0_all.deb
sudo apt-get -f install
```

C Robot Raconteur Reference

C.1 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. Note that in many cases a reference will be

made called “RRN” to `s` as a shorthand. Most programs using Robot Raconteur will use the following two lines to import the module as `RR` and then create the convenience variable (or shorthand) `RRN`.

```
import RobotRaconteur as RR
RRN=RR.RobotRaconteurNode.s
```

In the following member descriptions **RobotRaconteurNode.s** and **RRN** are interchangeable.

RobotRaconteurNode.s.RobotRaconteurVersion → string

Returns the version of the Robot Raconteur library.

RobotRaconteurNode.s.NodeName ↔ string OR unicode

The name of the node. This is used to help find the correct node for a service. It is guaranteed to be unique unless using `LocalTransport.StartServerAsNodeName()`. The name must be set before any other operations on the node. If it is not set it remains blank.

RobotRaconteurNode.s.NodeID ↔ 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.

RobotRaconteurNode.s.RegisterTransport(transport) → None

Registers a transport with the node.

Parameters:

- *transport* (Transport) - The transport to be registered

Return Value:

None

RobotRaconteurNode.s.ConnectService(url, username=None, credentials=None, servicelistener=None)
→ object

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

Parameters:

- *url* (string, unicode, or list) - The url to connect to. A list can be specified if there are multiple possible routes to the service.
- *username* (string OR unicode) - (optional) The username to use with authentication.
- *credentials* (dict) - (optional) The credentials to use with authentication.
- *servicelistener* (function) - (optional) A function to call when a client event is generated such as disconnect. The function should have the form `def callback(context, code, param):`.

Return Value:

(object) - The connected object. This is a Robot Raconteur object reference that provides access to the remote service object.

RobotRaconteurNode.s.DisconnectService(obj) → None

Disconnects a service.

Parameters:

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

Return Value:

None

RobotRaconteurNode.s.AsyncConnectService(url, username, credentials, servicelister, handler, timeout = `RR_TIMEOUT_INFINITE`) → None

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

Parameters:

- *url* (string, unicode, or list) - The url to connect to. A list can be specified if there are multiple possible routes to the service.
- *username* (string or unicode) - The username to use with authentication. None if authentication not used.
- *credentials* (dict) - The credentials to use with authentication. None if authentication is not used.
- *servicelister* (function) - A function to call when a client event is generated such as disconnect. The function should have the form `def callback(context, code, param):`. None if not used.
- *handler* (function) - The handler function that will be called by the thread pool upon completion. The function must have the form `handler(obj,err):`. `err` is None if no error occurred
- *timeout* (integer) - (optional) The timeout for the connect in seconds. Default is infinite.

Return Value:

None

RobotRaconteurNode.s.AsyncDisconnectService(obj, handler, timeout = `RR_TIMEOUT_INFINITE`) → None

Disconnects a service.

Parameters:

- *obj* (object) - The client object to disconnect. Must have been connected with the `ConnectService`

function.

- *handler* (function) - The handler function that will be called by the thread pool upon completion. The function must have the form `handler() : .` No error value is returned.
- *timeout* (integer) - (optional) The timeout for the disconnect in seconds. Default is infinite.

Return Value:

None

RobotRaconteurNode.s.NewStructure(*structtype*, *objectreference* = None) → structure

Returns a new Robot Raconteur structure with type *structtype*.

Parameters:

- *structtype* (string or unicode) - The fully qualified type of the structure.
- *objectreference* (object) - (optional) The client connection object reference to be used with this structure. This is necessary because each client maintains type information. A client must be provided from which type information can be queried. For services this parameter is unnecessary.

Return Value:

None

RobotRaconteurNode.s.GetExceptionType(*exceptiontype*) → exception

Returns a reference to the exception class of fully qualified type *exceptiontype*. Note that this is a class reference, not an instance.

Parameters:

- *exceptiontype* (string or unicode) - The fully qualified type of the exception.

Return Value:

(exception) - Returns a class of the requested exception type.

RobotRaconteurNode.s.GetConstants(*servicetype*) → structure

Returns a structure that contains the constants of the specified service type.

Parameters:

- *servicetype* (string or unicode) - The name of the service definition.

Return Value:

(structure) - Returns a structure containing the service definition constants.

RobotRaconteurNode.s.Shutdown() → None

Shuts down Robot Raconteur and closes all connections.

Parameters:

None

Return Value:

None

RobotRaconteurNode.s.RegisterServiceType(*servicetype*) → None

Registers a new service type with the node. This can either be a `string` containing the service definition or an object of type `ServiceDefinition`. It is only necessary to register a service definition when also registering a service. Clients will automatically retrieve the service definitions from the service. If the service definition uses the “import” statement, it must be registered after the imported service definition or an error will be generated.

Parameters:

- *servicetype* (`string`, `unicode`, or `ServiceDefinition`) - A string with the text service definition or a `ServiceDefinition` object to register with the node.

Return Value:

None

RobotRaconteurNode.s.GetServiceType(*name*) → ServiceDefinition

Returns the `ServiceDefinition` named *name*.

Parameters:

- *name* (`string` or `unicode`) - The name of the service type.

Return Value:

(`ServiceDefinition`) - The requested service definition.

RobotRaconteurNode.s.GetRegisteredServiceTypes() → list

Returns a list of the names of the registered service definitions.

Parameters:

None

Return Value:

(`list`) - A list of the names of the registered service definitions.

RobotRaconteurNode.s.IsServiceTypeRegistered(*servicename*) → bool

Returns True if *servicename* is registered, otherwise False

Parameters:

- *name* (string or unicode) - The name of the service type.

Return Value:

(bool) - True if service type is registered, otherwise False.

RobotRaconteurNode.s.GetServiceAttributes(obj) → dict

Retrieves the attributes of a connected service. *obj* must have be a service object connected through ConnectService.

Parameters:

- *obj* (object) - The connected service object

Return Value:

(dict) - The attributes of the remote service.

RobotRaconteurNode.s.GetPulledServiceType(obj, name) → ServiceDefinition

Returns the ServiceDefinition named *name* pulled by client *obj*.

Parameters:

- *obj* (object) - The connected service object
- *name* (string or unicode) - The name of the service type to retrieve the definition

Return Value:

(ServiceDefinition) - The requested service definition.

RobotRaconteurNode.s.GetPulledServiceTypes(obj) → list

Returns a list of the names of the service definitions pulled by the client *obj*.

Parameters:

- *obj* (object) - The connected service object

Return Value:

(list) - A list of the names of the registered service definitions.

RobotRaconteurNode.sFindObjectType(obj, membername, ind=None) → unicode

Returns the fully qualified name of objref return of *obj* named *membername* with optional index *ind*.

Parameters:

- *obj* (object) - The connected service object

- *membername* (string or unicode) - The name of the objref to query
- *ind* (string or unicode) - (optional) The index converted to a string of the objref

Return Value:

(unicode) - The fully qualified name of the object type.

RobotRaconteurNode.s.RegisterService(*name*, *servicetype*, *obj*, *securitypolicy*=None) → ServerContext

Registers a service with the node. Once registered, a client can access the registered object and all objref'd objects. The *securitypolicy* object can be used to specify authentication requirements.

Parameters:

- *name* (string or unicode) - The name of the service. This must be unique within the node.
- *servicetype* (string or unicode) - The fully qualified name of the root object type specified by a registered service type.
- *obj* (object) - The root object. It must be compatible with the object type specified in the *servicetype* parameter.
- *securitypolicy* (SecurityPolicy) - (optional) The security policy for this service.

Return Value:

(ServerContext) - The server context for this service.

RobotRaconteurNode.s.CloseService(*name*) → None

Closes the service with name *name*.

Parameters:

- *name* (string or unicode) - The name of the service to close.

Return Value:

None

RobotRaconteurNode.s.RequestObjectLock(*obj*, *flags*) → None

Requests an object lock for a connected service object. The flags specify if the lock is a "User" lock or a "Client" lock.

Parameters:

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

Return Value:

None

RobotRaconteurNode.s.ReleaseObjectLock(*obj*) → None

Requests an object lock for a connected service object. The flags specify if the lock is a “User” lock or a “Client” lock.

Parameters:

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

Return Value:

None

RobotRaconteurNode.s.AsyncRequestObjectLock(*obj*, *flags*, *handler*, *timeout* = `RR_TIMEOUT_INFINITE`) → None

Requests an object lock for a connected service object. The flags specify if the lock is a “User” lock or a “Client” lock.

Parameters:

- *obj* (object) - The object to lock. This object must have been created through `ConnectService` or an `objref`.
- *flags* (integer) - The flags for the lock. Must be `RobotRaconteurObjectLockFlags_USER_LOCK` for a “User” lock, or `RobotRaconteurObjectLockFlags_CLIENT_LOCK` for a “Client” lock.
- *handler* (function) - The handler the thread pool will call when the lock is completed. Must be of the form `handler(res, err) :. err` will be `None` if the lock is successful.
- *timeout* (integer) - (optional) The timeout in seconds. By default the timeout is infinite.

Return Value:

None

RobotRaconteurNode.s.AsyncReleaseObjectLock(*obj*, *handler*, *timeout* = `RR_TIMEOUT_INFINITE`) → None

Requests an object lock for a connected service object. The flags specify if the lock is a “User” lock or a “Client” lock.

Parameters:

- *obj* (object) - The object to unlock. This object must have been created through `ConnectService` or an `objref`.
- *handler* (function) - The handler the thread pool will call when the unlock is completed. Must be of the form `handler(res, err) :. err` will be `None` if the lock is successful.

- *timeout* (float) - (optional) The timeout in seconds. By default the timeout is infinite.

Return Value:

None

RobotRaconteurNode.s.MonitorEnter(*obj*, *timeout*) → None

Requests a monitor lock for a connected service object.

Parameters:

- *obj* (object) - The object to lock. This object must have been created through `ConnectService` or an objref.
- *timeout* (float) - The timeout for the lock in milliseconds. Specify -1 for no timeout.

Return Value:

None

RobotRaconteurNode.s.MonitorExit(*obj*) → None

Releases a monitor lock.

Parameters:

- *obj* (object) - The object to unlock. This object must have been locked through `MonitorEnter`.

Return Value:

None

RobotRaconteurNode.s.NowUTC() → datetime.datetime

Returns a the current system time. This function is intended to provide a high-resolution timer, but on Windows the resolution is limited to 16 ms. Future versions of Robot Raconteur may have better timing capabilities. This function will use the system clock or simulation clock if provided.

Parameters:

None

Return Value:

(datetime.datetime) - The current time in UTC

RobotRaconteurNode.s.Sleep(*duration*) → None

Sleep for *duration* in seconds. This function will use the system clock or simulation clock if provided.

Parameters:

- *duration* (double) - The duration to sleep in seconds.

Return Value:

None

RobotRaconteurNode.s.CreateAutoResetEvent() → AutoResetEvent

Returns a new AutoResetEvent. This event will use the system clock or simulation clock if provided.

Parameters:

None

Return Value:

(AutoResetEvent) - A new AutoResetEvent

RobotRaconteurNode.s.CreateRate(rate) → Rate

Returns a new Rate. This event will use the system clock or simulation clock if provided.

Parameters:

- *rate* (double) - The frequency of the rate in Hertz.

Return Value:

(Rate) - A new rate with the specified frequency

RobotRaconteurNode.s.CreateTimer(period, handler, oneshot=False) → Timer

Returns a new Timer. This event will use the system clock or simulation clock if provided.

Parameters:

- *period* (double) - The period of the timer in seconds.
- *handler* (function) - A handler for when the timer fires. It should accept one argument of type TimerEvent.
- *oneshot* (bool) - (optional) Set to True if the timer should only fire once, or False for a repeating timer.

Return Value:

(Timer) - A new timer

RobotRaconteurNode.s.SetExceptionHandler(handler) → None

Sets an exception handler to catch exceptions that occur during asynchronous operations.

Parameters:

- *handler* (function) - A function with one parameter that receives the exceptions.

Return Value:

None

RobotRaconteurNode.s.FindServiceByType(*servicetype*, *transportschemes*) → list

Finds services using auto-discovery based on the type of the root service object.

Parameters:

- *servicetype* (string or unicode) - The fully qualified type of the root object to search for.
- *transportschemes* (list) - A list of the schemes to search for string.

Return Value:

(list) - A list of ServiceInfo2 structures with the detected services.

RobotRaconteurNode.s.FindNodeByName(*nodename*, *transportschemes*) → list

Finds a node using auto-discovery based on the “NodeName”

Parameters:

- *nodename* (string or unicode) - The “NodeName” to search for.
- *transportschemes* (list) - A list of the schemes to search for string.

Return Value:

(list) - A list of NodeInfo2 structures with the detected nodes.

RobotRaconteurNode.s.FindNodeByID(*nodeid*, *transportschemes*) → list

Finds a node using auto-discovery based on the “NodeID”

Parameters:

- *NodeID* (NodeID) - The “NodeID” to search for.
- *transportschemes* (list) - A list of the schemes to search for string.

Return Value:

(list) - A list of NodeInfo2 structures with the detected nodes.

RobotRaconteurNode.s.AsyncFindServiceByType(*servicetype*, *transportschemes*, *handler*, *time-out=5*) → None

Finds services using auto-discovery based on the type of the root service object.

Parameters:

- *servicetype* (string or unicode) - The fully qualified type of the root object to search for.
- *transportschemes* (list) - A list of the schemes to search for string.
- *handler* (function) - A handler function that accepts the results. Must be of the form

handler(discoveredservices)

- *timeout* (float) - (optional) The time to search for nodes in seconds. Default is 5 seconds.

Return Value:

None

RobotRaconteurNode.s.AsyncFindNodeByName(*nodename*, *transportschemes*, *handler*, *timeout=5*) → None

Finds a node using auto-discovery based on the “NodeName”

Parameters:

- *nodename* (string or unicode) - The “NodeName” to search for.
- *transportschemes* (list) - A list of the schemes to search for string.
- *handler* (function) - A handler function that accepts the results. Must be of the form handler(discoveredservices)
- *timeout* (float) - (optional) The time to search for nodes in seconds. Default is 5 seconds.

Return Value:

None

RobotRaconteurNode.s.AsyncFindNodeByID(*nodeid*, *transportschemes*, *handler*, *timeout=5*) → None

Finds a node using auto-discovery based on the “NodeID”

Parameters:

- *NodeID* (NodeID) - The “NodeID” to search for.
- *transportschemes* (list) - A list of the schemes to search for string.
- *handler* (function) - A handler function that accepts the results. Must be of the form handler(discoveredservices)
- *timeout* (float) - (optional) The time to search for nodes in seconds. Default is 5 seconds.

Return Value:

None

RobotRaconteurNode.s.UpdateDetectedNodes() → None

Updates the detected nodes. Update must be called before GetDetectedNodes

Parameters:

None

Return Value:

None

RobotRaconteurNode.s.AsyncUpdateDetectedNodes(handler) → None

Updates the detected nodes asynchronously. Update must be called before GetDetectedNodes. handler should have no arguments.

Parameters:

None

Return Value:

None

RobotRaconteurNode.s.GetDetectedNodes() → list

Returns a list of NodeID of all the nodes that have been detected. This can be used with FindNodeByID to implement a custom search

Parameters:

None

Return Value:

(list) - A list of NodeID with the detected nodes.

RobotRaconteurNode.s.EndpointInactivityTimeout ↔ double

The length of time an endpoint will remain active without receiving a message in seconds.

RobotRaconteurNode.s.TransportInactivityTimeout ↔ double

The length of time a transport connection will remain active without receiving a message in seconds.

RobotRaconteurNode.s.TransactionTimeout ↔ double

The timeout for a transactional call in seconds. Default is 15 seconds.

RobotRaconteurNode.s.MemoryMaxTransferSize ↔ integer

During memory reads and writes, the data is transmitted in smaller pieces. This property sets the maximum size per piece. Default is 100 KB.

RobotRaconteurNode.s.ThreadPoolCount ↔ integer

Gets or sets the size of the thread pool. Default is 20. If thread exhaustion occurs increase this value.

RobotRaconteurNode.s.NodeDiscoveryMaxCacheCount ↔ integer

Gets or sets the number of discovered nodes to cache. When a node discovery packet is received, it is cached for use with auto-discovery. This cache number can be increased or decreased depending on the available memory and number of nodes on the network.

C.2 EventHook

class **EventHook**

EventHook is used to implement multiple listener events in Robot Raconteur Python. Callback functions are registered with the **EventHook** and are all notified when the event is fired.

operator +=

Adds a callback function to be notified of events.

operator -=

Removes a callback function.

fire(*) → None

Fires the event.

Parameters:

- ***** (*) - Variable arguments. Must match the expected event arguments.

Return Value:

None

C.3 ServiceInfo2

class **ServiceInfo2**

ServiceInfo2 contains the results of a search for a service using auto-detect. Typically a search will result in a list of **ServiceInfo2**. The **ConnectionURL** field is then used to connect to the service after the connect service is selected. **ConnectService** can take a list of URL and will attempt to connect using all the possibilities.

NodeName → unicode

The name of the found node.

NodeID → NodeID

The id of the found node.

Name → unicode

The name of the service.

RootObjectType → unicode

The fully qualified type of the root object in the service.

RootObjectImplements → list

List of the fully qualified types that the root object in the service implements.

ConnectionURL → list

A list of URL that can be used to connect to the service. List entries are type unicode.

Attributes → dict

A dict of Robot Raconteur type varvalue{string} that contains attributes specified by the service. This is used to help find the correct service to connect to.

C.4 NodeInfo2

class **NodeInfo2**

NodeInfo2 contains the results of a search for a node using auto-detect by “NodeName” ore “NodeID”. Typically a search will result in a list of NodeInfo2. The ConnectionURL field is then used to connect to the service after the connect service is selected. ConnectService can take a list of URL and will attempt to connect using all the possibilities.

NodeName → unicode

The name of the found node.

NodeID → NodeID

The id of the found node.

ConnectionURL → list

A partial URL to connect to the node. Append the service name to the returned URL to connect to the service. List entries are type unicode.

C.5 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.

MemberName → unicode

Returns the member name of this pipe.

Connect(*index*) → PipeEndpoint

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

Parameters:

- *index* (integer) - (optional) The index of the PipeEndpoint pair. This can be -1 to mean “any index”.

Return Value:

(PipeEndpoint) - The connected PipeEndpoint

AsyncConnect(*index*, *handler*, *timeout* = *RR.TIMEOUT_INFINITE*) → None

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

Parameters:

- *index* (integer) - (optional) The index of the PipeEndpoint pair. This can be -1 to mean “any index”.
- *handler* (function) - The handler function that will be called by the thread pool when the connection is complete. It must have the form `handler(pipeendpoint, err):`. `err` will be None if no error occurs.
- *timeout* (float) - The connect timeout in seconds. Default is infinite timeout.

Return Value:

None

PipeConnectCallback ↔ function

Specifies the callback to call on the service when a client connects a PipeEndpoint. This function must be the form of `def callback(pipeendpoint):`

C.6 PipeEndpoint

class **PipeEndpoint**

The PipeEndpoint class represents one end of a connected 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 `PipeEndpoint` connections are created by the `Pipe` members.

Endpoint → long

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.

Index → integer

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

Available → integer

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

ReceivePacket() → *

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

Parameters:

None

Return Value:

(*) - The next packet.

PeekPacket() → *

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

Parameters:

None

Return Value:

(*) - The next packet.

SendPacket(packet) → long

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

Parameters:

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

Return Value:

(long) - The packet number of the sent packet.

AsyncSendPacket(*packet*, *handler*, *timeout* = *RR_TIMEOUT_INFINITE*) → None

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

Parameters:

- *packet* (*) - The packet to send
- *handler* (function) - The handler function called by the thread pool when the packet has been sent. It must have the form `handler(packetnumber, err)`. `err` will be None if no error occurs.

Return Value:

None

Close() → None

Closes the pipe endpoint connection pair.

Parameters:

None

Return Value:

None

AsyncClose(*handler*, *timeout* = 2) → None

Closes the pipe endpoint connection pair.

Parameters:

- *handler* (function) - The handler function that will be called by the thread pool when the pipe endpoint close is complete. It must have the form `handler(err)`. `err` will be None if no error occurs.
- *timeout* (float) - The timeout in seconds. Default is 2 seconds.

Return Value:

None

PipeEndpointClosedCallback ↔ function

A callback function called when the PipeEndpoint is closed. This is used to detect when it has been closed. It must have the form `def callback(pipeendpoint):`

PacketReceivedEvent → EventHook

An event triggered when a packet is received. The event callback function should have the form

`def callback(pipeendpoint):` . See the definition of `EventHook` for more info on how it should be used. Do not set this field.

RequestPacketAck \leftrightarrow `bool`

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

PacketAckReceivedEvent \rightarrow `EventHook`

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 event callback function should have the form `def callback(pipeendpoint, packetnumber):` . The `packetnumber` will match the number returned by `SendPacket`. See the definition of `EventHook` for more info on how it should be used. Do not set this field.

C.7 Callback

class **Callback**

The `Callback` class implements the “callback” member type. This class allows a callback function to be specified on the client, and allows the service to retrieve functions that can be used to execute the specified function on the client.

Function \leftrightarrow `function`

Specifies the function that will be called for the callback. This is only available for the client.

GetClientFunction(endpoint) \rightarrow `function`

Retrieves a function that will be executed on the client selected by the *endpoint* parameter. The *endpoint* can be determined through `ServerEndpoint.GetCurrentEndpoint()`. This is only available in a service.

Parameters:

- *endpoint* (`long`) - The endpoint identifying the client to execute the function on

Return Value:

(`function`) - A function that will be executed on the client.

C.8 Wire

class **Wire**

The `Wire` class implements the “wire” member. The `Wire` object is used to create `WireConnection` ob-

jects which implement a connection between the client and the service. On the client side, the function `Connect` is used to connect the `WireConnection` to the service. On the service side, a callback function `ConnectCallback` is called when clients connects.

MemberName → unicode

Returns the member name of this wire.

Connect() → `WireConnection`

Connects and returns a `WireConnection` on the client connected to the service where another corresponding `WireConnection` is created.

Parameters:

None

Return Value:

(`WireConnection`) - The connected `WireConnection`.

AsyncConnect(handler, timeout = `RR_TIMEOUT_INFINITE`) → None

Connects and returns a `WireConnection` on the client connected to the service where another corresponding `WireConnection` is created.

Parameters:

- *handler* (function) - The handler function that will be called by the thread pool when the connection is complete. It must have the form `handler(wireconnection, err):`. `err` will be None if no error occurs.
- *timeout* (float) - The connect timeout in seconds. Default is infinite timeout.

Return Value:

None

WireConnectCallback ↔ function

Specifies the callback to call on the service when a client connects a `WireConnection`. This function must be the form of `def callback(wireconnection):`

C.9 WireConnection

class **WireConnection**

The `WireConnection` class represents one end of a wire connection which is formed by a pair of `WireConnection` objects, one in the client and one in the service. The wire connections are symmetric, meaning they are identical in both the client and service. The `InValue` on one end is set by the `OutValue` of the other end of the connection, and vice versa. The `WireConnection` connections are created by the `Wire` members. The wire is used to transmit a constantly changing value where only the latest value is of interest. If changes

arrive out of order, the out of order changes are dropped. Changes may also be dropped.

Endpoint → long

Returns the Robot Raconteur endpoint that this pipe endpoint is associated with. This is used by the service to detect which client the pipe endpoint is associated with. Each connected 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.

InValue → *

Returns the current in value of the wire connection, which is set by the matching remote wire connection's out value. This will raise an exception if the value has not been set by remote wire connection.

OutValue ↔ *

Sets the out value of this end of the wire connection. It is used to transmit a new value to the other end of the connection. The out value can also be retrieved. The type must match the wire defined in the service definition. This operation will place the new value into the send queue and return immediately.

InValueValid → bool

Returns True if the InValue has been set, otherwise False.

OutValueValid → bool

Returns True if the OutValue has been set, otherwise False.

LastValueReceivedTime → TimeSpec

Returns the last time that InValue has been received. This returns the time as a TimeSpec object. The time is in the *sender's* clock, meaning that it cannot be directly compared with the local clock. The basic Robot Raconteur library does not have a built in way to synchronize clocks, however future versions may have this functionality.

LastValueSentTime → TimeSpec

Returns the last time that OutValue was set. This time is in the local system clock.

Close() → None

Closes the wire connection pair.

Parameters:

None

Return Value:

None

AsyncClose(handler, timeout = 2) → None

Closes the wire connection pair.

Parameters:

- *handler* (function) - The handler function that will be called by the thread pool when the wire connection close is complete. It must have the form `handler(err) : err` will be `None` if no error occurs.
- *timeout* (float) - The timeout in seconds. Default is 2 seconds.

Return Value:

`None`

WireConnectionClosedCallback \leftrightarrow function

A callback function called when the `WireConnection` is closed. This is used to detect when it has been closed. It must have the form `def callback(wireconnection) :`

WireValueChanged \rightarrow EventHook

An event triggered when `InValue` has changed. The event callback function should have the form `def callback(wireconnection, value, timestamp) : .` See the definition of `EventHook` for more info on how it should be used. Do not set this field.

C.10 TimeSpec

class **TimeSpec**

Represents time in seconds and nanoseconds. The seconds is a 64-bit signed integer, and the nanoseconds are a 32-bit signed integer. For real time, the `TimeSpec` is relative to the standard Unix epoch January 1, 1970. The time may also be relative to another reference time.

seconds \leftrightarrow long

A 64-bit integer representing the seconds.

nanoseconds \leftrightarrow integer

A 32-bit integer representing the nanoseconds.

TimeSpec(seconds, nanoseconds) \rightarrow TimeSpec

Creates a new `TimeSpec`.

Parameters:

- *seconds* (long) - Seconds
- *nanoseconds* (long) - Nanoseconds

Return Value:

(TimeSpec) - The new TimeSpec.

operator ==

operator !=

operator >

operator <

operator >=

operator <=

operator -

operator +

Standard operators for use with TimeSpec.

cleanup_nanoseconds() → None

Adjusts value so that `nanoseconds` is positive.

Parameters:

None

Return Value:

None

Now() → TimeSpec

Returns a TimeSpec representing the current time relative to January 1st, 1970, 12:00 am.

Parameters:

None

Return Value:

(TimeSpec) - The current time

C.11 ArrayMemory

class **ArrayMemory**

The `ArrayMemory` is designed to represent a large array that is read in smaller pieces. It is used with the “memory” member to allow for random access to an array.

ArrayMemory(*array*) → `ArrayMemory`

Creates a new `ArrayMemory`.

Parameters:

- *array* () - The array

Return Value:

(`ArrayMemory`) - The new `ArrayMemory`

Length → `long`

The number of elements in the array.

Read(*memorypos*, *buffer*, *bufferpos*, *count*) → `None`

Reads data from the memory into *buffer*.

Parameters:

- *memorypos* (`long`) - The start position in the array.
- *buffer* (`list` or `numpy.array`) - The buffer to read the data into.
- *bufferpos* (`long`) - The start position in the supplied buffer.
- *count* (`long`) - The number of elements to read.

Return Value:

`None`

Write(*memorypos*, *buffer*, *bufferpos*, *count*) → `None`

Writes data from *buffer* into the memory.

Parameters:

- *memorypos* (`long`) - The start position in the array.
- *buffer* (`list` or `numpy.array`) - The buffer to write data from.

- *bufferpos* (long) - The start position in the supplied buffer.
- *count* (long) - The number of elements to write.

Return Value:

None

C.12 MultiDimArrayMemory

class MultiDimArrayMemory

The `MultiDimArrayMemory` is designed to represent a large multi-dimensional array that is read in smaller pieces. It is used with the “memory” member to allow for random access to an multi-dimensional array. It works with either the special class `MultiDimArray` that is not documented, or preferably the `numpy.array`. For the *memorypos*, *bufferpos*, and *count* parameters in the functions, a list is used to represent the dimensions. The dimensions are column-major as is `numpy.array`.

MultiDimArrayMemory(*array*) → MultiDimArrayMemory

Creates a new `MultiDimArrayMemory`.

Parameters:

- *array* (`MultiDimArray` or `numpy.array`) - The array

Return Value:

(`MultiDimArrayMemory`) - The new `MultiDimArrayMemory`.

DimCount → integer

The number of dimensions in the array.

Dims → list

The dimensions of the array in column-major order.

Complex → bool

True if the array is complex, otherwise False.

Read(*memorypos*, *buffer*, *bufferpos*, *count*) → None

Reads data from the memory into *buffer*.

Parameters:

- *memorypos* (list) - The start position in the array.
- *buffer* (`MultiDimArray` or `numpy.array`) - The buffer to read the data into.

- *bufferpos* (*list*) - The start position in the buffer.
- *count* (*list*) - The number of elements to read.

Return Value:

None

Write(*memorypos*, *buffer*, *bufferpos*, *count*) → None

Writes data from *buffer* into the memory.

Parameters:

- *memorypos* (*list*) - The start position in the array.
- *buffer* (*MultiDimArray* or *numpy.array*) - The buffer to write data from.
- *bufferpos* (*list*) - The start position in the buffer.
- *count* (*list*) - The number of elements to write.

Return Value:

None

C.13 ServerEndpoint

class **ServerEndpoint**

The **ServerEndpoint** represents a client connection on the service side. See Figure 1 that shows how the server endpoints work. For the Python bindings, this endpoint is used to access the current endpoint number and the current authenticated user.

ServerEndpoint.GetCurrentEndpoint() → long

Returns the endpoint number of the current client. This function works in “function” and “property” calls on the service side.

Parameters:

None

Return Value:

(long) - The current endpoint number

ServerEndpoint.GetCurrentAuthenticatedUser() → *AuthenticatedUser*

Returns the current authenticated user. This call will raise an exception if there is no user currently authenticated.

Parameters:

None

Return Value:

(AuthenticatedUser) - The current authenticated user.

C.14 ServerContext

class **ServerContext**

The `ServerContext` manages the service. For the Python bindings, a few functions are exposed.

ServerContext.GetCurrentServicePath() → unicode

Returns the service path of the current service object. The service path is a string with the name of the service and the name of the “objref”’s separated by “dots”. The objref indexes are put between square brackets, and the index is encoded in the HTTP URL style.

Parameters:

None

Return Value:

(unicode) - The current service path.

ServerContext.GetCurrentServerContext() → ServerContext

Returns the current server context for the current service object.

Parameters:

None

Return Value:

(ServerContext) - The current service context.

ReleaseServicePath(path, endpoints = None) → None

Releases an object and all “objref”’d object within the service path. This is the only way to release objects from the service without closing the service.

Parameters:

- *path* (string or unicode) - The service path.
- *endpoints* (list) - A list of endpoints to send the release notification to. The default of None will send the notification to all endpoints.

Return Value:

None

AddServerServiceListener(*listener*) → None

Adds a listener to be notified when a client connects, a client disconnects, or the service is closed.

Parameters:

- *listener* (function) - A function to call when a service event is generated. The function should have the form *def callback(context, code, param):*

Return Value:

None

SetServiceAttributes(*attributes*) → None

Sets the service attributes. These attributes can be retrieved by the client to help select the correct service.

Parameters:

- *attributes* (dict) - The attributes. Must match the type `varvalue{string}`.

Return Value:

None

C.15 AuthenticatedUser

class AuthenticatedUser

This class represents a user that has been authenticated for the service.

Username → unicode

The username of the authenticated user.

Privileges → list

The list of privileges for the user.

LoginTime → datetime.datetime

The login time of the user.

LastAccessTime → datetime.datetime

The time of last access by the user.

C.16 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-xxxx-xxxx-xxxxxxxxxxxx}` where the "x" is a hexadecimal digit. Note that if the `NodeID` is all zeros the shorthand `{0}` is supported as a convenience.

NodeID(*id*) → `NodeID`

Creates a new `NodeID`.

Parameters:

- *id* (string, unicode, or bytearray) - The value of the `NodeID` as a string or bytes.

Return Value:

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

__str__() → string

Returns the string representation of the `NodeID` using the standard Python `str` operator.

Parameters:

None

Return Value:

(string) - The string representation.

ToByteArray() → bytearray

Returns the bytearray representation of the `NodeID`.

Parameters:

None

Return Value:

(bytearray) - The bytearray representation.

operator ==

operator !=

Standard operators for use with `NodeID`.

C.17 RobotRaconteurException

class **RobotRaconteurException**

`RobotRaconteurException` represents an exception in Robot Raconteur. Every Robot Raconteur function may potentially throw an `Exception`. 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 associated with the error.

C.18 RobotRaconteurRemoteException

class **RobotRaconteurRemoteException**

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

errorname → unicode

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

errormessage → unicode

The message associated with the exception.

C.19 Transport

class **Transport**

The **Transport** class is the superclass for all transport types. It exposes one static method to get the current incoming connection URL.

Transport.GetCurrentTransportConnectionURL() → string

Returns the URL of the current incoming connection. Only valid when being called by a remote peer transaction

Parameters:

None

Return Value:

(string) - The URL as a string

C.20 LocalTransport

class **LocalTransport**

The **LocalTransport** provides communication between nodes on the same computer. It uses local transport mechanisms including named pipes and UNIX sockets. It also maintains “NodeIDs” that correspond to “NodeNames” when used with `StartServerAsNodeName()`. This means that services will have a unique “NodeID” associated with each “NodeName” on each computer. It also provides node detection within the same computer.

LocalTransport() → LocalTransport

Creates a new **LocalTransport** that can be registered with `RobotRaconteurNode.s.RegisterTransport()`

Parameters:

None

Return Value:

(LocalTransport) - The new LocalTransport

StartServerAsNodeID(*nodeid*) → None

Starts listening for connecting clients as “nodeid”. This function will also set the “NodeID” of `RobotRaconteurNode`. It must be called before registering the transport with the node. If the “NodeID” is already in use, an exception will be thrown.

Parameters:

- *nodeid* (NodeID) - The “NodeID” to use for the transport and node.

Return Value:

None

StartServerAsNodeName(*nodename*) → None

Starts listening for connection clients as “nodename”. This function will check the computer registry to find the corresponding “NodeID” for the supplied name. If one does not exist, a random one will be generated and saved. The function will set both the “NodeID” and “NodeName” of `RobotRaconteurNode`. It must be call before registering the transport with the node. If either the “NodeName” or “NodeID” is already in use, an exception will be thrown. If the “NodeName” is already in use and another instance needs to be started it is suggested that a “dot” and number will be appended to represent another instance. This function is the most common way that a server node will set its identification information. The generated “NodeID” can be determined by reading the property `RobotRaconteurNode.s.NodeID`

Parameters:

- *nodename* (string or unicode) - The nodename to use

Return Value:

None

StartClientAsNodeName(*nodename*) → None

This function is optional for the client and can be called to pull a “NodeID” from the registry. It starts the local client as “nodename”. This function will check the computer registry to find the corresponding “NodeID” for the supplied name. If one does not exist, a random one will be generated and saved. The function will set both the “NodeID” and “NodeName” of `RobotRaconteurNode`. It must be call before registering the transport with the node. If either the “NodeName” or “NodeID” is already in use, an exception will be thrown. If the “NodeName” is already in use and another instance needs to be started it is suggested that a “dot” and number be appended to represent another instance. The generated “NodeID” can be determined by reading the property `RobotRaconteurNode.s.NodeID`

Parameters:

- *nodename* (string or unicode) - The nodename to use

Return Value:

None

C.21 TcpTransport

class **TcpTransport**

The `TcpTransport` provides communication between different computers using standard TCP/IP communication (or on the same computer using loopback). A server can be started that opens a port on the computer to accept connection. This transport also provides node detection and service discovery for the local network. It fully supports both IPv4 and IPv6 communication.

TcpTransport() → `TcpTransport`

Creates a new `TcpTransport` that can be registered with `RobotRaconteurNode.s.RegisterTransport()`

Parameters:

None

Return Value:

(`TcpTransport`) - The new `TcpTransport`

StartServer(*port*) → `None`

Starts the server listening on port `port`. If `port` is "0", a random port is selected. Use `GetListenPort()` to find out what port is being used.

Parameters:

- *port* (integer) - The port to listen on

Return Value:

None

StartServerUsingPortSharer() → `None`

Starts the server listing using the *Robot Raconteur Port Sharer* service. The *Robot Raconteur Port Sharer* listens on Port 48653 (the official Robot Raconteur port) and forwards to the correct local service listening on the local computer. Specify the node name or node id in the connection URL to be connected to the correct node.

Parameters:

None

Return Value:

None

IsPortSharerRunning → `bool`

Used to determine if the port sharer is operational after connecting using `StartServerUsingPortSharer`.

GetListenPort() → integer

Returns the port that the transport is listening for connections on

Parameters:

None

Return Value:

(integer) - The port the transport is listening on

EnableNodeDiscoveryListening() → None

Starts listening for node discovery packets.

Parameters:

None

Return Value:

None

DisableNodeDiscoveryListening() → None

Stops listening for node discovery packets.

Parameters:

None

Return Value:

None

EnableNodeAnnounce() → None

Begins sending node discovery packets.

Parameters:

None

Return Value:

None

DisableNodeAnnounce() → None

Stops sending node discovery packets.

Parameters:

None

Return Value:

None

DefaultReceiveTimeout ↔ double

The TCP connections send heartbeat packets by default every 5 seconds to ensure that the connection is still active. After `DefaultReceiveTimeout`, the connection will be considered lost. Unit is in seconds. Default is 15 seconds.

DefaultConnectTimeout ↔ double

The TCP connect timeout in seconds. Default is 5 seconds.

DefaultHeartbeatPeriod ↔ double

The TCP connections send heartbeat packets every few seconds to test the connection status. Unit is in seconds. Default is 5 seconds.

MaxMessageSize ↔ integer

The maximum message size in bytes that can be sent through the connected TCP transport. Default is 12 MB. This should be made as small as possible for the node's application to minimize memory usage.

MaxConnectionCount ↔ integer

The maximum number of active connections that can be concurrently connected. This is used to throttle connections to preserve resources. The default is 0, meaning infinite connections.

LoadTlsNodeCertificate() → None

Loads a certificate for this node. The function will search for a certificate on the local machine matching the configured `NodeID`. Certificates can be installed using the *Robot Raconteur Certificate Utility*.

Parameters:

None

Return Value:

None

IsTlsNodeCertificateLoaded → bool

True if the certificate for this node has been loaded.

RequireTls ↔ bool

Set to True to require all TCP connections to use TLS. Highly recommended in any production environment!

IsTransportConnectionSecure(a) → bool

Returns True if the specified connection is secure.

Parameters:

- *a* (integer or object reference) - Either the local endpoint number or a client object reference.

Return Value:

(bool) - True if connection is secure.

IsPeerIdentityVerified(*a*) → bool

Returns True if the identity of the peer of this connection has been verified with a TLS certificate.

Parameters:

- *a* (integer or object reference) - Either the local endpoint number or a client object reference.

Return Value:

(bool) - True if peer identity has been verified

GetSecurePeerIdentity(*a*) → string

Returns the NodeID of the secure peer as a string. Throws an exception if connection is not secure.

Parameters:

- *a* (integer or object reference) - Either the local endpoint number or a client object reference.

Return Value:

(string) - The NodeID as a string.

AcceptWebSockets ↔ bool

Set to True to allow WebSockets to connect to the service. Enabled by default.

GetWebSocketAllowedOrigins() → list

Returns a list of the currently allowed WebSocket origins. The default values are:

- file://
- chrome-extension://
- http://robotraconteur.com
- http://*.robotraconteur.com
- https://robotraconteur.com

- `https://*.robotraconteur.com`

WebSocket origins are used to protect against Cross-Site Scripting attacks (XSS). When a web-browser client connects, they send the “origin” hostname of the webpage that is attempting to create the connection. For instance, a page that is located on the Robot Raconteur website would send “https://robotraconteur.com” as the hostname. The Robot Raconteur library by default has “https://robotraconteur.com” and “https://*.robotraconteur.com” listed as allowed origins, so this connection would be accepted. The “*” can be used to allow all subdomains to be accepted. Other hostnames can be added using the `AddWebSocketAllowedOrigin` function.

Parameters:

None

Return Value:

(list) - List containing the current allowed origins.

AddWebSocketAllowedOrigin(*origin*) → None

Add an allowed origin. See `GetWebSocketAllowedOrigins` for details on the format.

Parameters:

- *origin* (string) - The origin to add

Return Value:

None

RemoveWebSocketOrigin(*origin*) → None

Removes an origin from the allowed origin list.

Parameters:

- *origin* (string) - The origin to remove

Return Value:

None

C.22 CloudTransport

class **CloudTransport**

The `CloudTransport` provides the ability for the node to connect to the *Cloud Client* running locally on the same computer as the node. The *Cloud Client* provides a link between the node and Robot Raconteur Cloud. See the documentation for the Robot Raconteur Cloud for more information on how to use the Robot Raconteur Cloud.

StartAsClient() → None

Starts the transport and connects to the *Cloud Client* as a client node.

Parameters:

None

Return Value:

None

StartAsServer() → None

Starts the transport and connects to the *Cloud Client* as a client or server node.

Parameters:

None

Return Value:

None

C.23 HardwareTransport

class HardwareTransport

The `HardwareTransport` provides the ability to connect to USB devices using the *Robot Raconteur Hardware Service*. See the documentation for Robot Raconteur USB for more information. To use this class simply instantiate it and register it.

C.24 PipeBroadcaster

class PipeBroadcaster

Helper class that can be used in conjunction with a service side `Pipe` member to broadcast the same packets to all connected `PipeEndpoints`. It also provides a form of flow control by dropping packets if too many packets are still in transit. It automatically handles client connects and disconnects.

PipeBroadcaster(*pipe*, *backlog=-1*) → `PipeBroadcaster`

Creates a new `PipeBroadcaster` using the supplied `Pipe`. This should be called in the setter of the service object that is called by Robot Raconteur during initialization.

Parameters:

- *pipe* (`Pipe`) - The pipe to use for broadcasting
- *backlog* (`integer`) - (optional) - The maximum backlog allowed. The pipe will drop packets if more than the specified number of packets are still being transmitted. By default *backlog*

is set to -1, which means no packets will be dropped. During video transmission a normal value would be 3.

Return Value:

None

SendPacket(*packet*) → None

Sends a packet to all connected clients. The type must match the type specified by the pipe in the service definition.

Parameters:

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

Return Value:

None

AsyncSendPacket(*packet*, *handler*) → None

Sends a packet to be received by all connected clients. The type must match the type specified by the pipe in the service definition.

Parameters:

- *packet* (*) - The packet to send
- *handler* (function) - The handler function called by the thread pool when the packet has been sent. It must have the form `handler()`. Note that there are no parameters passed to this handler. In many cases it can safely be set to `lambda: None` so that it is ignored.

Return Value:

None

C.25 WireBroadcaster

class **WireBroadcaster**

Helper class that can be used in conjunction with a service side `Wire` member to broadcast the same value to all connected `WireConnections`. It automatically handles client connects and disconnects.

WireBroadcaster(*wire*) → `WireBroadcaster`

Creates a new `WireBroadcaster` using the provided `Wire`. This should be called in the setter of the service object that is called by Robot Raconteur during initialization.

Parameters:

- *wire* (`Wire`) - The wire to use for broadcasting

Return Value:

None

OutValue ↔ *

Sets the out value of this end of the wire connection. It is used to transmit a new value to all connected `WireConnections`. This property is write-only in this case. The type must match the wire defined in the service definition. This operation will place the new value into the send queues and return immediately.

D Example Software

D.1 iRobotCreateService.py

```
#Example Robot Raconteur service in Python

import serial
import struct
import time
import RobotRaconteur as RR
#Convenience shorthand to the default node.
#RRN is equivalent to RR.RobotRaconteurNode.s
RRN=RR.RobotRaconteurNode.s
import threading
import numpy
import traceback
import sys

#Port names and NodeID of this service
serial_port_name="/dev/ttyUSB0"

#The service definition of this service.
create_servicedef="""
#Service to provide sample interface to the iRobot Create
service experimental.create

option version 0.5

struct SensorPacket
    field uint8 ID
    field uint8[] Data
end struct

object Create
    option constant int16 DRIVE.STRAIGHT 32767
    option constant int16 SPIN.CLOCKWISE -1
    option constant int16 SPIN.COUNTERCLOCKWISE 1

    function void Drive(int16 velocity , int16 radius)

    function void StartStreaming()
    function void StopStreaming()

    property int32 DistanceTraveled
    property int32 AngleTraveled
    property uint8 Bumpers

    event Bump()

    wire SensorPacket packets

    callback uint8[] play_callback(int32 DistanceTraveled , int32 AngleTraveled)

end object
"""

class Create_impl(object):
    def __init__(self):
        self.Bump=RR.EventHook()
        self._lock=threading.RLock()
        self._recv_lock=threading.RLock()
        self._packets=None
        self._play_callback=None
```

```

self._connected_wires=dict()

self._lastbump=False
self._Bumpers=0
self._Play=False
self._DistanceTraveled=0
self._AngleTraveled=0
self._streaming=False
self._downsample=0
self._ep=0

def Drive(self, velocity, radius):
    with self._lock:
        dat=struct.pack(">B2h",137,velocity, radius)
        self._serial.write(dat)

def StartStreaming(self):
    with self._lock:
        if (self._streaming):
            raise Exception("Already streaming")

        self._ep=RR.ServerEndpoint.GetCurrentEndpoint()
        #Start the thread that receives serial data
        self._streaming=True
        t=threading.Thread(target=self._recv_thread)
        t.start()
        #Send command to start streaming packets after a short delay
        time.sleep(.1)
        command=struct.pack(">6B", 148, 4, 7, 19, 20, 18)
        self._serial.write(command)

def StopStreaming(self):
    if (not self._streaming):
        raise Exception("Not streaming")
    with self._lock:
        command=struct.pack(">2B", 150, 0)
        self._serial.write(command)
        self._streaming=False

@property
def DistanceTraveled(self):
    return self._DistanceTraveled

@property
def AngleTraveled(self):
    return self._AngleTraveled

@property
def Bumpers(self):
    return self._Bumpers

@property
def packets(self):
    return self._packets
@packets.setter
def packets(self, value):
    self._packets=value
    #Set the wire connect callback for the wire server
    self._packets.WireConnectCallback=self._packet_wire_connected

#Add connected wire to list. You can also request an event when InValue
#changes here
def _packet_wire_connected(self, wire):

```

```

        self._connected_wires[wire.Endpoint]=wire
        #If you also want value changed event updates:
        #wire.WireValueChanged+=self.value_changed_callback
        #See client for usage example

    @property
    def play_callback(self):
        return self._play_callback;
    @play_callback.setter
    def play_callback(self, value):
        self._play_callback=value

    def Init(self, port):
        with self._lock:
            self._serial=serial.Serial(port=port, baudrate=57600)
            dat=struct.pack(">4B",128,132,150, 0)
            self._serial.write(dat)
            time.sleep(.1)
            self._serial.flushInput()

    def Shutdown(self):
        with self._lock:
            self._serial.close()

#Thread function that runs serial receive loop
    def _recv_thread(self):
        try:
            while self._streaming:
                if (not self._streaming): return
                self._ReceiveSensorPackets()
        except:
            #Exception will be thrown when the port is closed
            #just ignore it
            if (self._streaming):

                traceback.print_exc()
            pass

#Receive the packets and execute the right commands
    def _ReceiveSensorPackets(self):
        while self._serial.inWaiting() > 0:
            seed=struct.unpack('>B',self._serial.read(1))[0]

            if (seed!=19):
                continue
            nbytes=struct.unpack('>B',self._serial.read(1))[0]

            if nbytes==0:
                continue

            packets=self._serial.read(nbytes)

            checksum=self._serial.read(1)

            #Send packet to the client through wire. If there is a large backlog
            #of packets don't send
            if (self._serial.inWaiting() < 20):

                self._SendSensorPackets(seed, packets)

            readpos=0
            while (readpos < nbytes):

```

```

id=struct.unpack('B',packets[readpos])[0]
readpos+=1

#Handle a bumper packet
if (id==7):
    flags=struct.unpack("B",packets[readpos])[0]
    readpos+=1
    if (((flags & 0x1)!=0) or ((flags & 0x2)!=0)):
        if (not self._lastbump):
            self._fire_Bump()
        self._lastbump=True
    else:
        self._lastbump=False
    self._Bumpers=flags

#Handle distance packets
elif (id==19):
    try:
        distbytes=packets[readpos:(readpos+2)]
        self._DistanceTraveled+=struct.unpack(">h",distbytes)[0]
        readpos+=2
    except:
        print struct.unpack("%sB" % len(packets),packets)
        raise

#Handle angle packets
elif (id==20):
    distbytes=packets[readpos:(readpos+2)]
    self._DistanceTraveled+=struct.unpack(">h",distbytes)[0]
    readpos+=2

#Handle buttons packets
elif (id==18):
    buttons=struct.unpack("<B",packets[readpos])[0]
    play=buttons & 0x1
    if (play==1):
        if (not self._Play):
            self._play()
        self._Play=True
    else:
        self._Play=False
    readpos+=1
else:
    readpos+=1

def _SendSensorPackets(self,seed,packets):

    #Pack the data into the structure to send to the lient
    data=numpy.frombuffer(packets, dtype='u1')
    #Create the new structure using the "NewStructure" function
    strt=RRN.NewStructure('experimental.create.SensorPacket')
    #Set the data
    strt.ID=seed
    strt.Data=data

    #Iterate over all connected wires and set the OutValue
    eps=self._connected_wires.keys()
    for ep in eps:
        try:
            wire=self._connected_wires[ep]
            wire.OutValue=strt

```

```

        except:
            #If there is an error assume the wire has disconnected
            del(self._connected_wires[ep])

#Fire the bump event, all connected clients will receive
def _fire_Bump(self):
    self.Bump.fire()

def _play(self):
    if (self._ep==0):
        return

    try:
        cb_func=self.play_callback.GetClientFunction(self._ep)
        notes=cb_func(self._DistanceTraveled, self._AngleTraveled)
        notes2=list(notes) + [141,0]

        command=struct.pack("%sB" % (5+len(notes)),140,0,len(notes)/2,*notes2)
        with self._lock:
            self._serial.write(command)

    except:
        traceback.print_exc()

def main():

    #Enable numpy
    RRN.UseNumPy=True

    #Initialize the object in the service
    obj=Create_impl()

    if (len(sys.argv) >=2):
        port=sys.argv[1]
    else:
        port=serial_port_name

    obj.Init(port)

    #Create Local transport, start server as name, and register it
    t1=RRN.LocalTransport()
    t1.StartServerAsNodeName("experimental.create.Create")
    RRN.RegisterTransport(t1)

    #Create the transport, register it, and start the server
    t2=RRN.TcpTransport()
    RRN.RegisterTransport(t2)
    t2.StartServer(2354) #random port, any unused port is fine

    #Attempt to load a TLS certificate
    try:
        t2.LoadTlsNodeCertificate()
    except:
        print "warning:_could_not_load_TLS_certificate"

    t2.EnableNodeAnnounce()

    #Register the service type and the service
    RRN.RegisterServiceType(create_servicedef)
    RRN.RegisterService("Create","experimental.create.Create",obj)

```

```
#Wait for the user to stop the server
raw_input("Server_started ,_press_enter_to_quit...")

#Shutdown
obj.Shutdown()

#You MUST shutdown or risk segfault...
RRN.Shutdown()

if __name__ == '__main__':
    main()
```

D.2 iRobotCreateClient.py

```
#Example iRobot Create client in Python

from RobotRaconteur.Client import *
import time
import numpy
import sys

#Function to call when "Bump" event occurs
def Bumped():
    print "Bump!!"

def main():

    url='rr+tcp://localhost:2354?service=Create'
    if (len(sys.argv)>=2):
        url=sys.argv[1]

    #Instruct Robot Raconteur to use NumPy
    RRN.UseNumPy=True

    #Connect to the service
    c=RRN.ConnectService(url)

    #Start streaming data packets
    c.StartStreaming()

    #Add a function handler for the "Bump" event
    c.Bump += Bumped

    #Connect a WireConnection to the "packets" wire
    wire=c.packets.Connect()

    #Add a callback function for when the wire value changes
    wire.WireValueChanged+=wire_changed

    #Set the play_callback function for this client
    c.play_callback.Function=play_callback

    #Drive a bit
    c.Drive(100,1000)
    time.sleep(5)
    c.Drive(0,1000)
    time.sleep(10)

    #Stop streaming data
    c.StopStreaming()

#Function to call when the wire value changes
def wire_changed(w,value,time):

    val=w.InValue
    #Print the new value to the console. Comment out this line
    #to see the other output more clearly
    print val.Data

#Callback for when the play button is pressed on the Create
def play_callback(dist,angle):
    return numpy.array([69,16,60,16,69,16],dtype='u1')

if __name__ == '__main__':
    main()
```


D.3 SimpleWebcamService.py

```
#Simple example Robot Raconteur webcam service

#Note: This example is intended to demonstrate Robot Raconteur
#and is designed to be simple rather than optimal.

import time
import RobotRaconteur as RR
#Convenience shorthand to the default node.
#RRN is equivalent to RR.RobotRaconteurNode.s
RRN=RR.RobotRaconteurNode.s
import threading
import numpy
import traceback
import cv2.cv as cv

#The service definition of this service.
webcam.servicedef="""
#Service to provide sample interface to webcams
service experimental.createwebcam

option version 0.5

struct WebcamImage
    field int32 width
    field int32 height
    field int32 step
    field uint8[] data
end struct

struct WebcamImage_size
    field int32 width
    field int32 height
    field int32 step
end struct

object Webcam
    property string Name
    function WebcamImage CaptureFrame()

    function void StartStreaming()
    function void StopStreaming()
    pipe WebcamImage FrameStream

    function WebcamImage_size CaptureFrameToBuffer()
    memory uint8[] buffer
    memory uint8[*] multidimbuffer
end object

object WebcamHost
    property string{int32} WebcamNames
    objref Webcam{int32} Webcams
end object

"""

#Class that implements a single webcam
class Webcam_impl(object):
    #Init the camera being passed the camera number and the camera name
    def __init__(self, cameraid, cameraname):
        self._lock=threading.RLock()
```

```

self._framestream=None
self._framestream_endpoints=dict()
self._framestream_endpoints_lock=threading.RLock()
self._streaming=False
self._cameraname=cameraname

#Create buffers for memory members
self._buffer=numpy.array([], dtype="u1")
self._multidimbuffer=numpy.array([], dtype="u1")

#Initialize the camera
with self._lock:
    self._capture=cv.CaptureFromCAM(cameraid)
    cv.SetCaptureProperty( self._capture ,cv.CV_CAP_PROP_FRAME_WIDTH,320)
    cv.SetCaptureProperty( self._capture ,cv.CV_CAP_PROP_FRAME_HEIGHT,240)

#Return the camera name
@property
def Name(self):
    return self._cameraname

#Capture a frame and return a WebcamImage structure to the client
def CaptureFrame(self):

    with self._lock:
        image=RRN.NewStructure("experimental.createwebcam.WebcamImage")
        frame1=cv.QueryFrame( self._capture )
        frame=cv.GetMat(frame1)
        image.width=frame.width
        image.height=frame.height
        image.step=frame.step
        image.data=numpy.frombuffer( frame.tostring() ,dtype="u1" )

    return image

#Start the thread that captures images and sends them through connected
#FrameStream pipes
def StartStreaming(self):
    if (self._streaming):
        raise Exception("Already_streaming")
    self._streaming=True
    t=threading.Thread(target=self.frame_threadfunc)
    t.start()

#Stop the streaming thread
def StopStreaming(self):
    if (not self._streaming):
        raise Exception("Not_streaming")
    self._streaming=False

#FrameStream pipe member property getter and setter
@property
def FrameStream(self):
    return self._framestream
@FrameStream.setter
def FrameStream( self , value ):
    self._framestream=value
    #Set the PipeConnectCallback. to FrameStream_pipeconnect that will be
    #called when a PipeEndpoint connects.
    value.PipeConnectCallback=self.FrameStream_pipeconnect

#Function called when a PipeEndpoint connects. pipe_ep is the endpoint
def FrameStream_pipeconnect(self , pipe_ep ):

```

```

#Lock the _framestream_endpoints dictionary, and place the pipe_ep in
#the dict
with self._framestream_endpoints_lock:
    #Add pipe_ep to the dictionary by endpoint and index
    self._framestream_endpoints[(pipe_ep.Endpoint, pipe_ep.Index)]=pipe_ep
    #Set the function to call when the pipe endpoint is closed
    pipe_ep.PipeEndpointClosedCallback=self.FrameStream.pipclosed

#Called when a pipe endpoint is closed; it will delete the endpoint
def FrameStream.pipclosed(self, pipe_ep):
    with self._framestream_endpoints_lock:
        try:
            del(self._framestream_endpoints[(pipe_ep.Endpoint, pipe_ep.Index)])
        except:
            traceback.print_exc()

#Function that will send a frame at ideally 4 fps, although in reality it
#will be lower because Python is quite slow. This is for
#demonstration only...
def frame_threadfunc(self):

    #Loop as long as we are streaming
    while(self._streaming):
        #Capture a frame
        try:
            frame=self.CaptureFrame()
        except:
            #TODO: notify the client that streaming has failed
            self._streaming=False
            return
        #Lock the pipe endpoints dictionary
        with self._framestream_endpoints_lock:
            #Iterate through the endpoint keys
            pipe_keys=self._framestream_endpoints.keys()
            for ind in pipe_keys:
                if (ind in self._framestream_endpoints):
                    #Try to send the frame to the connected
                    #PipeEndpoint
                    try:
                        pipe_ep=self._framestream_endpoints[ind]
                        pipe_ep.SendPacket(frame)
                    except:
                        #If there is an error, assume the
                        #pipe endpoint has been closed
                        self.FrameStream.pipclosed(pipe_ep)
            #Put in a 250 ms delay
            time.sleep(.25)

#Captures a frame and places the data in the memory buffers
def CaptureFrameToBuffer(self):

    #Capture and image and place it into the buffer
    image=self.CaptureFrame()

    self._buffer=image.data
    self._multidimbuffer=numpy.concatenate((image.data[2::3].reshape((image.height, image.
        width,1)),image.data[1::3].reshape((image.height, image.width,1)),image.data[0::3].
        reshape((image.height, image.width,1))), axis=2)

    #Create and populate the size structure and return it

```

```

        size=RRN.NewStructure("experimental.createwebcam.WebcamImage.size")
        size.height=image.height
        size.width=image.width
        size.step=image.step
        return size

#Return the memories. It would be better to reuse the memory objects,
#but for simplicity return new instances when called
@property
def buffer(self):
    return RR.ArrayMemory(self._buffer)

@property
def multidimbuffer(self):
    return RR.MultiDimArrayMemory(self._multidimbuffer)

#Shutdown the Webcam
def Shutdown(self):
    self._streaming=False
    del(self._capture)

#A root class that provides access to multiple cameras
class WebcamHost_impl(object):
    def __init__(self,camera_names):
        cams=dict()
        for i in camera_names:
            ind,name=i
            cam=Webcam_impl(ind,name)
            cams[ind]=cam

        self._cams=cams

    #Returns a map (dict in Python) of the camera names
    @property
    def WebcamNames(self):
        o=dict()
        for ind in self._cams.keys():
            name=self._cams[ind].Name
            o[ind]=name
        return o

    #objref function to return Webcam objects
    def get_Webcams(self,ind):
        #The index for the object may come as a string, so convert to int
        #before using. This is only necessary in Python
        int_ind=int(ind)

        #Return the object and the Robot Raconteur type of the object
        return self._cams[int_ind], "experimental.createwebcam.Webcam"

    #Shutdown all the webcams
    def Shutdown(self):
        for cam in self._cams.itervalues():
            cam.Shutdown()

def main():

    RRN.UseNumPy=True

    #Initialize the webcam host root object

```

```

camera_names=[(0,"Left"),(1,"Right")]
obj=WebcamHost_impl(camera_names)

#Create Local transport, start server as name, and register it
t1=RR.LocalTransport()
t1.StartServerAsNodeName("experimental.createwebcam.WebcamHost")
RRN.RegisterTransport(t1)

#Initialize the transport and register the root object
t2=RR.TcpTransport()
RRN.RegisterTransport(t2)
t2.StartServer(2355)

#Attempt to load a TLS certificate
try:
    t2.LoadTlsNodeCertificate()
except:
    print "warning: could not load TLS certificate"

t2.EnableNodeAnnounce()

RRN.RegisterServiceType(webcam.servicedef)
RRN.RegisterService("Webcam","experimental.createwebcam.WebcamHost",obj)

c1=obj.get.Webcams(0)[0]
c1.CaptureFrameToBuffer()

#Wait for the user to shutdown the service
raw_input("Server started, press enter to quit...")

#Shutdown
obj.Shutdown()

RRN.Shutdown()

if __name__ == '__main__':
    main()

```

D.4 SimpleWebcamClient.py

```
#Simple example Robot Raconteur webcam client
#This program will capture a frame from both webcams and show it
#on the screen
from RobotRaconteur.Client import *
import time
import thread
import numpy
import cv2
import sys

#Function to take the data structure returned from the Webcam service
#and convert it to an OpenCV array
def WebcamImageToMat(image):
    frame2=image.data.reshape([image.height, image.width, 3], order='C')
    return frame2

#Main program
def main():

    url='rr+tcp://localhost:2355/?service=Webcam'
    if (len(sys.argv)>=2):
        url=sys.argv[1]

    #Start up Robot Raconteur and connect, standard by this point
    RRN.UseNumPy=True
    c_host=RRN.ConnectService(url)

    #Use objref's to get the cameras. c_host is a "WebcamHost" type
    #and is used to find the webcams
    c1=c_host.get_Webcams(0)
    c2=c_host.get_Webcams(1)

    #Pull a frame from each camera, c1 and c2
    frame1=WebcamImageToMat(c1.CaptureFrame())
    frame2=WebcamImageToMat(c2.CaptureFrame())

    #Show the images
    cv2.imshow(c1.Name, frame1)
    cv2.imshow(c2.Name, frame2)

    #CV wait for key press on the image window and then destroy
    cv2.waitKey()
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

D.5 SimpleWebcamClient_streaming.py

```
#Simple example Robot Raconteur webcam client
#This program will show a live streamed image from
#the webcams. Because Python is a slow language
#the framerate is low...

from RobotRaconteur.Client import *

import time
import thread
import numpy
import cv2
import sys

#Function to take the data structure returned from the Webcam service
#and convert it to an OpenCV array
def WebcamImageToMat(image):
    frame2=image.data.reshape([image.height, image.width, 3], order='C')
    return frame2

current_frame=None

def main():

    url='rr+tcp://localhost:2355?service=Webcam'
    if (len(sys.argv)>=2):
        url=sys.argv[1]

    #Startup, connect, and pull out the camera from the objref
    RRN.UseNumPy=True
    c_host=RRN.ConnectService(url)

    c=c_host.get_Webcams(0)

    #Connect the pipe FrameStream to get the PipeEndpoint p
    p=c.FrameStream.Connect(-1)

    #Set the callback for when a new pipe packet is received to the
    #new_frame function
    p.PacketReceivedEvent+=new_frame
    try:
        c.StartStreaming()
    except: pass

    cv2.namedWindow("Image")

    while True:
        #Just loop resetting the frame
        #This is not ideal but good enough for demonstration

        if (not current_frame is None):
            cv2.imshow("Image",current_frame)
        if cv2.waitKey(50)!=-1:
            break
    cv2.destroyAllWindows()

    p.Close()
    c.StopStreaming()

#This function is called when a new pipe packet arrives
def new_frame(pipe_ep):
    global current_frame
```

```
#Loop to get the newest frame
while (pipe_ep.Available > 0):
    #Receive the packet
    image=pipe_ep.ReceivePacket()
    #Convert the packet to an image and set the global variable
    current_frame=WebcamImageToMat(image)

if __name__ == '__main__':
    main()
```


D.6 SimpleWebcamClient_memory.py

```
#Simple example Robot Raconteur webcam client
#This program will capture a frame from both webcams and show it
#on the screen

from RobotRaconteur.Client import *
import time
import thread
import numpy
import cv2
import sys

#Main program
def main():

    url='rr+tcp://localhost:2355?service=Webcam'
    if (len(sys.argv)>=2):
        url=sys.argv[1]

    #Start up Robot Raconteur and connect, standard by this point
    RRN.UseNumPy=True
    c_host=RRN.ConnectService(url)

    c1=c_host.get_Webcams(0)

    #Save image to buffer
    size=c1.CaptureFrameToBuffer()

    #Read the data from the "buffer" memory. For this example just read the
    #entire buffer
    l=c1.buffer.Length
    data=numpy.zeros(l, dtype="u1")

    c1.buffer.Read(0, data, 0, l)

    frame1=data.reshape([size.height, size.width, 3], order='C')
    cv2.imshow("buffer", frame1)

    #Read segment from the "multidimbuffer" and display the "red" channel
    bufsize=c1.multidimbuffer.Dimensions
    print bufsize

    #create a smaller buffer with 1 channel
    segdata=numpy.zeros([100,100,1], dtype="u1")
    c1.multidimbuffer.Read([10,10,0], segdata, [0,0,0], [100,100,1])

    cv2.imshow("multidimbuffer", segdata)

    #CV wait for key press on the image window and then destroy
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

D.7 iRobotCreateAsyncClient.py

```
from RobotRaconteur.Client import *
import time
import thread
import numpy
import threading
import sys

ev=threading.Event()
err=None

def client_handler(e):
    global err
    err=e
    ev.set()

class AsyncCreateClient(object):
    def __init__(self, handler):
        self._handler=handler
        self._c=None
    def start(self):
        try:
            url='rr+tcp://localhost:2354?service=Create'
            if (len(sys.argv)>=2):
                url=sys.argv[1]
            RRN.AsyncConnectService(url, None, None, None, self._handler1, 5)
        except Exception as e:
            self._handler(e)

    def handler1(self, c, err):
        if (err is not None):
            self._handler(err)
        return
        try:
            c.async_get.Bumpers(self._handler2, 0.5)
            self.c=c
        except Exception as e:
            self._handler(e)

    def handler2(self, bumpers, err):
        if (err is not None):
            self._handler(err)
        return
        try:
            self.c.async_set.Bumpers(10, self._handler3, 0.5)
        except Exception as e:
            self._handler(e)

    def handler3(self, err):
        #In this case we expect an error because this is read only
        if (err is None):
            self._handler(Exception("Expected an error"))
        return
        try:
            RRN.AsyncDisconnectService(self.c, self._handler4)
        except Exception as e:
            self._handler(e)

    def handler4(self):
        self._handler(None)

def main():
```

```
RRN.UseNumPy=True

c=AsyncCreateClient(client_handler)
c.start()

ev.wait()

if (err is None):
    print "No error occurred!"
else:
    print "Error occurred:_" + repr(err)

if __name__ == '__main__':
    main()
```

D.8 iRobotCreateService_Broadcaster.py

```
#Example Robot Raconteur service in Python

import serial
import struct
import time
import RobotRaconteur as RR
#Convenience shorthand to the default node.
#RRN is equivalent to RR.RobotRaconteurNode.s
RRN=RR.RobotRaconteurNode.s
import threading
import numpy
import traceback
import sys

#Port names and NodeID of this service
serial_port_name="/dev/ttyUSB0"

#The service definition of this service.
create.servicedef="""
#Service to provide sample interface to the iRobot Create
service experimental.create

option version 0.5

struct SensorPacket
    field uint8 ID
    field uint8[] Data
end struct

object Create
    option constant int16 DRIVE.STRAIGHT 32767
    option constant int16 SPIN.CLOCKWISE -1
    option constant int16 SPIN.COUNTERCLOCKWISE 1

    function void Drive(int16 velocity , int16 radius)

    function void StartStreaming()
    function void StopStreaming()

    property int32 DistanceTraveled
    property int32 AngleTraveled
    property uint8 Bumpers

    event Bump()

    wire SensorPacket packets

    callback uint8[] play_callback(int32 DistanceTraveled , int32 AngleTraveled)

end object
"""

class Create_impl(object):
    def __init__(self):
        self.Bump=RR.EventHook()
        self._lock=threading.RLock()
        self._recv_lock=threading.RLock()
        self._packets=None
        self._play_callback=None
        self._connected_wires=dict()

        self._lastbump=False
```

```

self._Bumpers=0
self._Play=False
self._DistanceTraveled=0
self._AngleTraveled=0
self._streaming=False
self._downsample=0
self._ep=0

def Drive(self, velocity, radius):
    with self._lock:
        dat=struct.pack(">B2h",137,velocity, radius)
        self._serial.write(dat)

def StartStreaming(self):
    with self._lock:
        if (self._streaming):
            raise Exception("Already streaming")

        self._ep=RR.ServerEndpoint.GetCurrentEndpoint()
        #Start the thread that receives serial data
        self._streaming=True
        t=threading.Thread(target=self._recv_thread)
        t.start()
        #Send command to start streaming packets after a short delay
        time.sleep(.1)
        command=struct.pack(">6B", 148, 4, 7, 19, 20, 18)
        self._serial.write(command)

def StopStreaming(self):
    if (not self._streaming):
        raise Exception("Not streaming")
    with self._lock:
        command=struct.pack(">2B", 150, 0)
        self._serial.write(command)
        self._streaming=False

@property
def DistanceTraveled(self):
    return self._DistanceTraveled

@property
def AngleTraveled(self):
    return self._AngleTraveled

@property
def Bumpers(self):
    return self._Bumpers

@property
def packets(self):
    return self._packets
@packets.setter
def packets(self,value):
    self._packets=value
    #Create the wire broadcaster
    self._packets_broadcaster=RR.WireBroadcaster(self._packets)

@property
def play_callback(self):
    return self._play_callback;
@play_callback.setter
def play_callback(self,value):

```

```

        self._play_callback=value

    def Init(self,port):
        with self._lock:
            self._serial=serial.Serial(port=port,baudrate=57600)
            dat=struct.pack(">4B",128,132,150, 0)
            self._serial.write(dat)
            time.sleep(.1)
            self._serial.flushInput()

    def Shutdown(self):
        with self._lock:
            self._serial.close()

#Thread function that runs serial receive loop
    def _recv_thread(self):
        try:
            while self._streaming:
                if (not self._streaming): return
                self._ReceiveSensorPackets()
        except:
            #Exception will be thrown when the port is closed
            #just ignore it
            if (self._streaming):

                traceback.print_exc()
            pass

#Receive the packets and execute the right commands
    def _ReceiveSensorPackets(self):
        while self._serial.inWaiting() > 0:
            seed=struct.unpack('>B',self._serial.read(1))[0]

            if (seed!=19):
                continue
            nbytes=struct.unpack('>B',self._serial.read(1))[0]

            if nbytes==0:
                continue

            packets=self._serial.read(nbytes)

            checksum=self._serial.read(1)

            #Send packet to the client through wire. If there is a large backlog
            #of packets don't send
            if (self._serial.inWaiting() < 20):

                self._SendSensorPackets(seed, packets)

            readpos=0
            while (readpos < nbytes):
                id=struct.unpack('B',packets[readpos])[0]
                readpos+=1

                #Handle a bumper packet
                if (id==7):
                    flags=struct.unpack("B",packets[readpos])[0]
                    readpos+=1
                    if (((flags & 0x1)!=0) or ((flags & 0x2)!=0)):
                        if (not self._lastbump):
                            self._fire_Bump()

```

```

        self._lastbump=True
    else:
        self._lastbump=False
    self._Bumpers=flags

#Handle distance packets
    elif (id==19):
        try:
            distbytes=packets[readpos:(readpos+2)]
            self._DistanceTraveled+=struct.unpack(">h",distbytes)[0]
            readpos+=2
        except:
            print struct.unpack("%sB" % len(packets),packets)
            raise

#Handle angle packets
    elif (id==20):
        distbytes=packets[readpos:(readpos+2)]
        self._DistanceTraveled+=struct.unpack(">h",distbytes)[0]
        readpos+=2

#Handle buttons packets
    elif (id==18):
        buttons=struct.unpack("<B",packets[readpos])[0]
        play=buttons & 0x1
        if (play==1):
            if (not self._Play):
                self._play()
                self._Play=True
            else:
                self._Play=False
            readpos+=1
        else:
            readpos+=1

def _SendSensorPackets(self,seed,packets):

    #Pack the data into the structure to send to the lient
    data=numpy.frombuffer(packets, dtype='u1')
    #Create the new structure using the "NewStructure" function
    strt=RRN.NewStructure('experimental.create.SensorPacket')
    #Set the data
    strt.ID=seed
    strt.Data=data

    #Set the OutValue for the broadcaster
    self._packets.broadcaster.OutValue=strt

#Fire the bump event, all connected clients will receive
def _fire.Bump(self):
    self.Bump.fire()

def _play(self):
    if (self._ep==0):
        return

    try:
        cb_func=self.play_callback.GetClientFunction(self._ep)
        notes=cb_func(self._DistanceTraveled, self._AngleTraveled)
        notes2=list(notes) + [141,0]

        command=struct.pack("%sB" % (5+len(notes)),140,0,len(notes)/2,*notes2)

```

```

        with self._lock:
            self._serial.write(command)

    except:
        traceback.print_exc()

def main():

    #Enable numpy
    RRN.UseNumPy=True

    #Initialize the object in the service
    obj=Create_impl()

    if (len(sys.argv) >=2):
        port=sys.argv[1]
    else:
        port=serial_port_name

    obj.Init(port)

    #Create Local transport, start server as name, and register it
    t1=RRN.LocalTransport()
    t1.StartServerAsNodeName("experimental.create.Create")
    RRN.RegisterTransport(t1)

    #Create the transport, register it, and start the server
    t2=RRN.TcpTransport()
    RRN.RegisterTransport(t2)
    t2.StartServer(2354) #random port, any unused port is fine

    #Attempt to load a TLS certificate
    try:
        t2.LoadTlsNodeCertificate()
    except:
        print "warning:_could_not_load_TLS_certificate"

    t2.EnableNodeAnnounce()

    #Register the service type and the service
    RRN.RegisterServiceType(create_servicedef)
    RRN.RegisterService("Create","experimental.create.Create",obj)

    #Wait for the user to stop the server
    raw_input("Server_started,_press_enter_to_quit...")

    #Shutdown
    obj.Shutdown()

    #You MUST shutdown or risk segfault...
    RRN.Shutdown()

if __name__ == '__main__':
    main()

```


D.9 SimpleWebcamService_Broadcaster.py

```
#Simple example Robot Raconteur webcam service

#Note: This example is intended to demonstrate Robot Raconteur
#and is designed to be simple rather than optimal.

import time
import RobotRaconteur as RR
#Convenience shorthand to the default node.
#RRN is equivalent to RR.RobotRaconteurNode.s
RRN=RR.RobotRaconteurNode.s
import threading
import numpy
import traceback
import cv2

#The service definition of this service.
webcam.servicedef= """
#Service to provide sample interface to webcams
service experimental.createwebcam

option version 0.5

struct WebcamImage
    field int32 width
    field int32 height
    field int32 step
    field uint8[] data
end struct

struct WebcamImage_size
    field int32 width
    field int32 height
    field int32 step
end struct

object Webcam
    property string Name
    function WebcamImage CaptureFrame()

    function void StartStreaming()
    function void StopStreaming()
    pipe WebcamImage FrameStream

    function WebcamImage_size CaptureFrameToBuffer()
    memory uint8[] buffer
    memory uint8[*] multidimbuffer
end object

object WebcamHost
    property string{int32} WebcamNames
    objref Webcam{int32} Webcams
end object

"""

#Class that implements a single webcam
class Webcam_impl(object):
    #Init the camera being passed the camera number and the camera name
    def __init__(self, cameraid, cameraname):
        self._lock=threading.RLock()
```

```

self._framestream=None
self._framestream_endpoints=dict()
self._framestream_endpoints_lock=threading.RLock()
self._streaming=False
self._cameraname=cameraname

#Create buffers for memory members
self._buffer=numpy.array([], dtype="u1")
self._multidimbuffer=numpy.array([], dtype="u1")

#Initialize the camera
with self._lock:
    self._capture=cv2.VideoCapture(cameraid)
    self._capture.set(cv2.CAP_PROP_FRAME_WIDTH,320)
    self._capture.set(cv2.CAP_PROP_FRAME_HEIGHT,240)

#Return the camera name
@property
def Name(self):
    return self._cameraname

#Capture a frame and return a WebcamImage structure to the client
def CaptureFrame(self):
    with self._lock:
        image=RRN.NewStructure("experimental.createwebcam.WebcamImage")
        frame=self._capture.read()[1]
        image.width=frame.shape[1]
        image.height=frame.shape[0]
        image.step=frame.shape[1]*3
        image.data=frame.reshape(frame.size, order='C')

    return image

#Start the thread that captures images and sends them through connected
#FrameStream pipes
def StartStreaming(self):
    if (self._streaming):
        raise Exception("Already streaming")
    self._streaming=True
    t=threading.Thread(target=self.frame_threadfunc)
    t.start()

#Stop the streaming thread
def StopStreaming(self):
    if (not self._streaming):
        raise Exception("Not streaming")
    self._streaming=False

#FrameStream pipe member property getter and setter
@property
def FrameStream(self):
    return self._framestream
@FrameStream.setter
def FrameStream(self, value):
    self._framestream=value
    #Create the PipeBroadcaster and set backlog to 3 so packets
    #will be dropped if the transport is overloaded
    self._framestream_broadcaster=RR.PipeBroadcaster(value,3)

#Function that will send a frame at ideally 4 fps, although in reality it
#will be lower because Python is quite slow. This is for
#demonstration only...
def frame_threadfunc(self):

```

```

#Loop as long as we are streaming
while( self._streaming):
    #Capture a frame
    try:
        frame=self.CaptureFrame()
    except:
        #TODO: notify the client that streaming has failed
        self._streaming=False
        return
    #Send the new frame to the broadcaster. Use AsyncSendPacket
    #and a blank handler. We really don't care when the send finishes
    #since we are using the "backlog" flow control in the broadcaster.
    self._framestream_broadcaster.AsyncSendPacket(frame,lambda: None)

    #Put in a 100 ms delay
    time.sleep(.1)

#Captures a frame and places the data in the memory buffers
def CaptureFrameToBuffer( self):

    #Capture and image and place it into the buffer
    image=self.CaptureFrame()

    self._buffer=image.data
    self._multidimbuffer=numpy.concatenate((image.data[2::3].reshape((image.height,image.
        width,1)),image.data[1::3].reshape((image.height,image.width,1)),image.data[0::3].
        reshape((image.height,image.width,1))),axis=2)

    #Create and populate the size structure and return it
    size=RRN.NewStructure("experimental.createwebcam.WebcamImage.size")
    size.height=image.height
    size.width=image.width
    size.step=image.step
    return size

#Return the memories. It would be better to reuse the memory objects,
#but for simplicity return new instances when called
@property
def buffer( self):
    return RR.ArrayMemory( self._buffer)

@property
def multidimbuffer( self):
    return RR.MultiDimArrayMemory( self._multidimbuffer)

#Shutdown the Webcam
def Shutdown( self):
    self._streaming=False
    del(self._capture)

#A root class that provides access to multiple cameras
class WebcamHost_impl(object):
    def __init__( self,camera_names):
        cams=dict()
        for i in camera_names:
            ind,name=i
            cam=Webcam_impl(ind,name)
            cams[ind]=cam

        self._cams=cams

```

```

#Returns a map (dict in Python) of the camera names
@property
def WebcamNames(self):
    o=dict()
    for ind in self._cams.keys():
        name=self._cams[ind].Name
        o[ind]=name
    return o

#objref function to return Webcam objects
def get_Webcams(self,ind):
    #The index for the object may come as a string, so convert to int
    #before using. This is only necessary in Python
    int_ind=int(ind)

    #Return the object and the Robot Raconteur type of the object
    return self._cams[int_ind], "experimental.createwebcam.Webcam"

#Shutdown all the webcams
def Shutdown(self):
    for cam in self._cams.itervalues():
        cam.Shutdown()

def main():

    RRN.UseNumPy=True

    #Initialize the webcam host root object
    camera_names=[(0,"Left"),(1,"Right")]
    obj=WebcamHost.impl(camera_names)

    #Create Local transport, start server as name, and register it
    t1=RR.LocalTransport()
    t1.StartServerAsNodeName("experimental.createwebcam.WebcamHost")
    RRN.RegisterTransport(t1)

    #Initialize the transport and register the root object
    t2=RR.TcpTransport()
    RRN.RegisterTransport(t2)
    t2.StartServer(2355)

    #Attempt to load a TLS certificate
    try:
        t2.LoadTlsNodeCertificate()
    except:
        print "warning: could not load TLS certificate"

    t2.EnableNodeAnnounce()

    RRN.RegisterServiceType(webcam_servicedef)
    RRN.RegisterService("Webcam","experimental.createwebcam.WebcamHost",obj)

    c1=obj.get_Webcams(0)[0]
    c1.CaptureFrameToBuffer()

    #Wait for the user to shutdown the service
    raw_input("Server started, press enter to quit...")

    #Shutdown
    obj.Shutdown()

```

```
RRN.Shutdown()
```

```
if __name__ == '__main__':  
    main()
```

E Software License

Robot Raconteur® - A communication library for robotics and automation systems

Copyright (C) 2014 John Wason <wason (at) wasontech.com>, Wason Technology, LLC

END USER LICENSE AGREEMENT

READ THIS BEFORE CLICKING “I ACCEPT”

THIS AGREEMENT IS A CONTRACT

THIS END USER LICENSE AGREEMENT (THE “AGREEMENT”) IS A LEGAL AGREEMENT BETWEEN YOU AND WASON TECHNOLOGY, LLC, A NEW YORK LIMITED LIABILITY COMPANY (“LICENSOR”).

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE CLICKING “I ACCEPT.” CLICKING “I ACCEPT” MEANS YOU ACCEPT THESE TERMS AND CONDITIONS AND UNDERSTAND THAT THEY WILL BE LEGALLY BINDING ON YOU PERSONALLY, IF YOU ARE INSTALLING THE SOFTWARE FOR PERSONAL USE, OR THE ORGANIZATION IF YOU ARE INSTALLING THE SOFTWARE AS PART OF YOUR WORK FOR AN ORGANIZATION (“LICENSEE”).

IF YOU DO NOT AGREE WITH THE TERMS OF THIS AGREEMENT, OR DO NOT HAVE THE AUTHORITY TO BIND THE LICENSEE, DO NOT CLICK “I ACCEPT.”

Licensor and Licensee may be referred to collectively herein as the “Parties” and individually as a “Party.”

IN CONSIDERATION and exchange of the mutual covenants of the Parties and the mutual benefits to be received under this Agreement, the Parties agree as follows:

1. As used in this Agreement, the following terms shall have the following meanings, unless otherwise expressly stated. Certain other terms may be defined elsewhere in this Agreement.

1. "Documentation" means Licensor's standard functional specifications regarding the use of the Licensed Software generally supplied by Licensor to its end users, whether in printed or digital format, as well as complete or partial copies of the foregoing.
2. "Licensed Software" means the proprietary software library known as Robot Raconteur® in machine-readable, object code form.
3. "Open Source Components" means all those components of the Licensed Software covered by "open source" software licenses.
4. "Open Source Licenses" means all licenses for the Open Source Components, as they may be amended from time to time.
5. "Library" means the Licensed Software and the Documentation, collectively.
6. "Use" means to download, install, store, execute and operate the Library and/or any individual component thereof, for the Library's intended purposes only.
7. "Platform" means any software or platform with which Licensee Uses the Library (including, for the avoidance of doubt, unauthorized Use outside of the Library's intended purposes).

2. USE OF LICENSED SOFTWARE

1. Licensee agrees to Use the Library subject to all of the terms and conditions of this Agreement, which Licensee hereby affirmatively accepts.
2. Without limiting any other provisions of this Agreement, Licensee understands and accepts that Licensee's use of the Library is subject to Licensee's full compliance with the Open Source Licenses and any applicable Platform licenses. Licensee accepts sole responsibility for complying with Open Source Licenses and Platform licenses.
3. Open Source Components included with the Licensed Software as of the date of this Agreement are listed in Schedule A hereto, which is attached and by this reference incorporated as a part of this Agreement.
4. Licensee understands and agrees that future versions of the Licensed Software may add or remove certain Open Source Components. Licensee may at any time request, and upon such request Licensor shall provide, a current list of Open Source Components and links to applicable Open Source Licenses.

5. Licensor has no obligation to provide support or maintenance for the Library under this Agreement.

3. **OWNERSHIP**

1. It is hereby understood and agreed that Licensor is the exclusive owner of all worldwide rights, title and interest to the Library, and all copies thereof, regardless of the media or form in which such copies may exist. The Library is made available to Licensee through the within license only, not sale, and Licensee does not and shall never acquire any ownership rights in or to the Library.
2. Licensee shall not cause, nor permit either through its direct efforts or through any third party, the modification, disassembly, de-compilation, or reverse engineering of the Licensed Software, nor otherwise attempt to discover any source code or trade secret related to the Licensed Software. Without limiting other remedies available to Licensor, all of which are expressly reserved: any use of the Library not expressly authorized by this Agreement shall automatically terminate this license and Agreement.

4. **LICENSE**

1. Conditioned upon Licensee's compliance with all the terms and conditions of this Agreement, Licensor hereby grants Licensee a nonexclusive, non-transferable, revocable right to Use the Library, without the payment of a fee.
 2. Licensor reserves the right to impose fees for premium services and/or updated versions of the Library.
 3. Without limitation, Licensee shall not:
 1. Copy, reproduce or make use of the Library except as expressly set forth in this Agreement;
 2. Modify, disassemble, de-compile or reverse engineer the Library;
- Undertake custom implementations without Licensor's express written consent;
 - 1. Use encryption without a certificate issued by Wason Technology, LLC except incidentally over a VPN or SSH tunnel;
 - 2. Distribute the Library other than as it may be incorporated into software of Licensee's that Licensee lawfully creates in accordance with the terms and conditions of this Agreement;
 - 3. Publicly perform or display the Library other than in the course of Licensee's authorized Use; or
 - Exercise any other right or privilege not expressly granted herein.

1. Reference anywhere in this Agreement to the “Library” shall mean the entire Library and/or any portion thereof, as required by the context.
2. Licensee acquires no rights whatever, under this Agreement, to use or display for any purpose any trademarks, brand names, logos or trade dress of Licensor.
3. Projects using the Library must be identified on product packaging, webpages and documentation as, “Robot Raconteur® Compatible” using Licensor-supplied logos.
4. All rights not expressly granted herein are reserved by Licensor.
5. This license terminates immediately upon termination of this Agreement.
6. The terms of this license are conditions and not merely covenants.

5. TERM AND TERMINATION

1. This Agreement is effective until terminated as provided herein.
2. Licensee may terminate this Agreement at any time by destroying all copies of the Library, in all media, in Licensee’s possession or control.
3. This Agreement will terminate immediately if Licensee fails to comply with any term or condition of this Agreement. In such event, no notice shall be required by Licensor to effect such termination. Any breach by Licensee of any terms or conditions herein shall be deemed material.
4. Upon termination of this Agreement, Licensee agrees immediately to destroy all copies of the Library in Licensee’s possession or control, in any form or media, along with related printed or written materials; or to return same to Licensor at Licensee’s expense.

6. RESTRICTIONS ON TRANSFER

1. This Agreement shall run with the Library.
2. Licensee shall not sublicense, assign, or transfer this Agreement except as expressly provided in this Agreement. Any attempt by Licensee otherwise to sublicense, assign, or transfer any of Licensee’s rights, duties, or obligations hereunder is void.
3. This Agreement is binding upon the heirs, executors, administrators and other legal representatives of the parties and will be for the benefit of the parties’ successors and assigns.

7. UPDATES AND SUGGESTIONS

1. Licensor may, from time to time, make available to Licensee updates and enhancements to the Library. Use of all such updates and enhancements by Licensee shall be subject to the terms and conditions of this Agreement and to any additional terms and conditions Licensor may require.
2. At no time shall any end-user suggestion for enhancements, updates, fixes or customizations be considered the confidential information of the Licensee or end-

user. All end-user suggestions or enhancement requests from any end-user of the Library automatically become the sole property of Licensor, without attribution or compensation of any sort to the Licensee or end-user. Licensee agrees that all copyright, patent, trade secret and other intellectual property or ownership rights to such suggestions or enhancements, are hereby assigned to Licensor and shall remain the sole property of Licensor.

8. COPYRIGHT AND PROPRIETARY INFORMATION

1. Excepting the Open Source Components, Licensor hereby reserves all right, title, and interest in and to the Library, including but not limited to all worldwide copyright, patent, trademark, trade secret and other proprietary rights relating thereto. Licensor's claim includes but is not limited to the compilation of the Licensed Software and of the Library. In no case shall Licensee or any end-user be considered a joint author or owner of the Library.
2. Licensee acknowledges that the Library may contain Licensor's trade secrets and that, in any event, the Library constitutes valuable confidential property of Licensor, the disclosure or misappropriation of which would cause irreparable harm to Licensor.
3. Licensee's obligation to maintain the confidentiality of any information reasonably known by Licensee to constitute a trade secret of Licensor shall survive termination of this Agreement, and shall be perpetual.
4. Except as otherwise provided in this Agreement, Licensee shall not cause or permit copying, reproduction or disclosure of the Library, or cause or permit the delivery, performance, display or distribution of any part thereof to any third person or entity, for any purpose whatsoever, without the prior written consent of Licensor.

9. LIMITED WARRANTY; LIMITED LIABILITY

1. LICENSEE ACCEPTS THE LIBRARY "AS IS" AND "AS AVAILABLE." LICENSOR DISCLAIMS ALL REPRESENTATIONS, WARRANTIES OR CONDITIONS OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY RIGHTS. LICENSOR DOES NOT ADOPT ANY REPRESENTATION OR WARRANTY OF ANY MANUFACTURER, SUPPLIER OR MERCHANT OF ANY INFORMATION, PRODUCT OR SERVICE, INCLUDING BUT NOT LIMITED TO THIRD-PARTY SOFTWARE. LICENSOR DOES NOT WARRANT THAT ERRORS IN THE LIBRARY WILL BE CORRECTED.
2. USE OF THE LIBRARY IS AT THE USER'S OWN RISK. UNLESS OTHERWISE EXPRESSLY SPECIFIED, LICENSOR, ITS PRINCIPALS, EMPLOYEES AND AGENTS SHALL NOT BE LIABLE TO LICENSEE OR

TO ANY THIRD PARTY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, PERSONAL INJURY, PRIVACY OR DATA BREACH, DAMAGES FOR LOST REVENUE, LOST OR DAMAGED DATA OR OTHER COMMERCIAL OR ECONOMIC LOSS, WHETHER BASED IN CONTRACT, TORT (INCLUDING NEGLIGENCE) OR ANY OTHER THEORY OF LIABILITY, ARISING OUT OF USE OF THE LIBRARY OR ANY CONSEQUENCES FLOWING, DIRECTLY OR INDIRECTLY, THEREFROM. IN NO EVENT SHALL LICENSOR BE LIABLE FOR HARDWARE OR SOFTWARE PROBLEMS DUE TO INTERFACING OF THE LIBRARY WITH LICENSEE'S EXISTING HARDWARE OR SOFTWARE. LICENSOR SHALL NOT BE LIABLE FOR DAMAGES RESULTING FROM THE IMPROPER OR INCORRECT USAGE OR OPERATION OF THE LIBRARY BY LICENSEE, ITS EMPLOYEES, OR ANY THIRD PARTIES. LICENSOR SHALL NOT BE LIABLE FOR DAMAGES RESULTING FROM LICENSEE'S BREACH OF ANY OPEN SOURCE LICENSES OR PLATFORM LICENSES. THE FOREGOING LIMITATIONS SHALL APPLY EVEN IF LICENSOR HAS BEEN ADVISED OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGE OR LOSS. IN NO EVENT WILL THE AGGREGATE LIABILITY OF LICENSOR AND ITS REPRESENTATIVES FOR ANY DAMAGES OR CLAIMS ARISING OUT OF OR RELATING TO USE OF THE LIBRARY, WHETHER IN CONTRACT, TORT, OR OTHERWISE, EXCEED THE TOTAL LICENSE FEE, IF ANY, LICENSEE HAS PAID TO LICENSOR UNDER THIS AGREEMENT. THIS SECTION WILL SURVIVE AND APPLY NOTWITHSTANDING THE FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

3. LICENSEE ACKNOWLEDGES THAT IT HAS READ THE FOREGOING DISCLAIMERS OF WARRANTY AND LIMITATION OF LIABILITY AND UNDERSTANDS THAT LICENSEE ASSUMES THE ENTIRE RISK OF USE OF THE LIBRARY.
4. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES AND/OR LIABILITIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU. THIS AGREEMENT GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY FROM STATE TO STATE.

10. **REMEDIES** Licensor's entire liability and Licensee's exclusive remedy for any claim arising under this Agreement shall be delivery of a replacement copy of the Library that is free of defects in materials or workmanship.

11. **COMPLIANCE WITH LAWS; EXPORT RESTRICTIONS** Licensee agrees to use the Library only in accordance with all applicable laws, regulations and statutes, worldwide. This includes, without limitation, privacy and data security laws. Licensee warrants and represents that Licensee does not intend to, and will not, directly or indirectly, export or transmit the Library to any country in violation of United States export restrictions.
12. **INDEMNITY** Licensee agrees fully to defend and indemnify Licensor from any and all claims, liabilities, settlements, judgments, awards, costs and expenses (including reasonable attorney fees) related to Licensee's use of the Library and/or Licensee's violation of any of the terms or conditions of this Agreement. Without limitation, this indemnity of Licensor includes and extends to any and all claims, liabilities, settlements, judgments, awards, costs and expenses (including reasonable attorney fees) arising from Licensee's breach of Open Source Licenses and/or Platform licenses, agreements, terms or conditions.
13. **ACCEPTANCE OF ELECTRONIC CONTRACT** Licensee agrees that this Agreement has the same legal force and effect as a written contract with the written signature of an authorized representative and that it satisfies any laws that require a writing or signature, including any applicable Statute of Frauds. Licensee further agrees that it shall not challenge the validity, enforceability or admissibility of this Agreement on the grounds that it was electronically transmitted or authorized. A printed version of this Agreement shall be admissible in judicial or administrative proceedings based upon or relating to the Agreement to the same extent and subject to the same conditions as other business documents and records originally generated and maintained in printed form. Licensee acknowledges that it has had the opportunity to print this Agreement.
14. **MISCELLANEOUS TERMS**
1. Nothing in this Agreement shall be construed to constitute or appoint either Party as the agent, representative or employee of the other Party for any purpose, or to grant either Party any rights or authority to assume or create any obligation or responsibility, whether express or implied, for or on behalf of or in the name of the other, or to bind the other in any way or manner.
 2. Licensee agrees to execute any and all documents reasonably necessary in order to effectuate the purposes and intent of this Agreement, including but not limited to documents affirming the authority of the individual who executed this Agreement to bind Licensee.
 3. This Agreement is the entire agreement between the Parties and supersedes all prior written or oral agreements between the Parties relating to the subject matter hereof. No amendment to this Agreement shall be binding or enforceable unless

reduced to a writing executed by the Parties. If any portion of this Agreement is found to be void or unenforceable, the remaining portion shall be enforceable with the invalid portion removed, giving all reasonable construction to permit the essential purposes of the Agreement to be achieved. The Parties' various rights and remedies hereunder shall be construed to be cumulative.

4. This Agreement shall be deemed to have been made in the State of New York, and it shall be governed by the substantive laws of the State of New York, USA without regard to any applicable conflict of laws provisions. The Parties submit to jurisdiction in the state and Federal courts serving Orange County, New York and hereby waive any jurisdictional, venue or inconvenient forum defenses otherwise available. Provided, however, that if Licensor is sued or joined in an action in any other court or forum in respect of any matter which may give rise to a claim hereunder, Licensee consents to the jurisdiction of such court or forum over any such claim.
5. Nothing contained in this Agreement shall be construed to require the commission of any act contrary to law. Captions and headings used in this Agreement are for purposes of convenience only and shall not be deemed to limit, affect the scope, meaning or intent of this Agreement, nor shall they otherwise be given any legal effect.
6. All sections of this Agreement, which by their nature should survive termination, shall survive termination, including, but not limited to, confidentiality obligations, indemnification obligations, warranty disclaimers and limitations of liability.

LICENSEE HEREBY ACKNOWLEDGES THAT LICENSEE HAS READ THIS AGREEMENT, UNDERSTANDS IT AND AGREES TO BE BOUND BY ITS TERMS AND CONDITIONS.

THE INDIVIDUAL EXECUTING THIS AGREEMENT PERSONALLY AFFIRMS, WARRANTS AND REPRESENTS THAT HE/SHE HAS AUTHORITY TO BIND LICENSEE TO THIS AGREEMENT, AND UNDERSTANDS THAT LICENSOR IS RELYING UPON SUCH AFFIRMATION, WARRANTY AND REPRESENTATION.

SCHEDULE A

BOOST <http://www.boost.org/users/license.html>

OpenSSL <https://www.openssl.org/source/license.html>

Robot Raconteur® is a registered trademark of Wason Technology, LLC and may only be used with the supplied software libraries. Custom implementations of the protocol must be certified by Wason Technology, LLC.

References

- [1] Boost C++ Libraries. [Online]. Available: <http://www.boost.org/>
- [2] *The WebSocket Protocol*, IETF Std. RFC 6455, 2011.