Introduction to Robot Raconteur® using Python

Version 0.9 Beta

http://robotraconteur.com

[DRAFT]

John Wason, Ph.D.

Wason Technology, LLC

16 Sterling Lake Road

Tuxedo, NY 10987

wason@wasontech.com

http://wasontech.com

June 19, 2019

Contents

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 an Apache-2.0 licensed open-source 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.

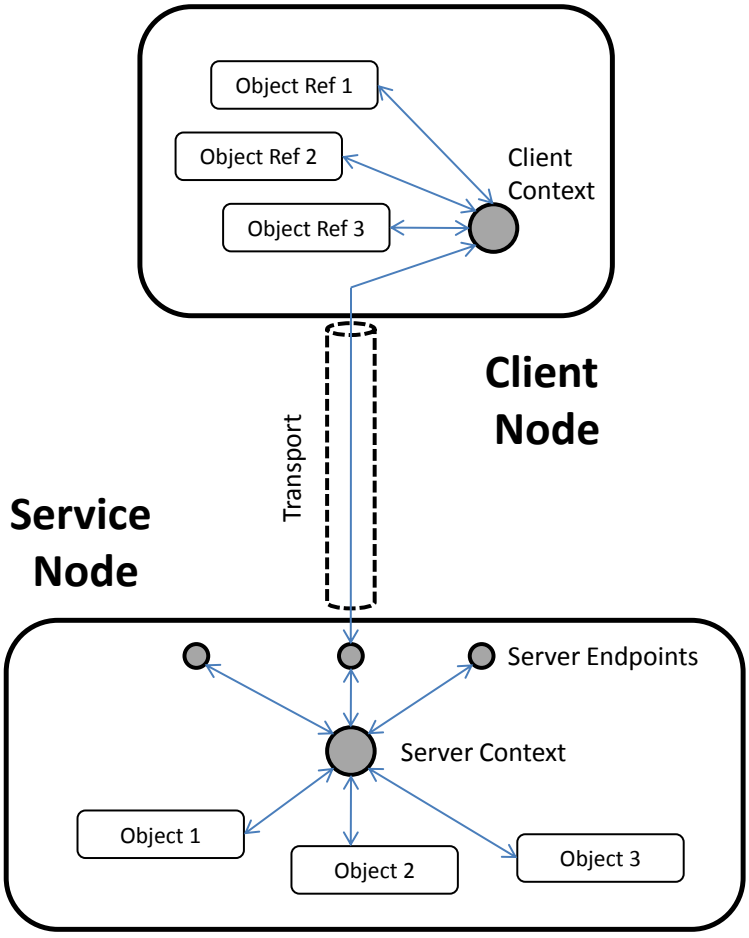

Figure 1: Configuration of Client-Service communication

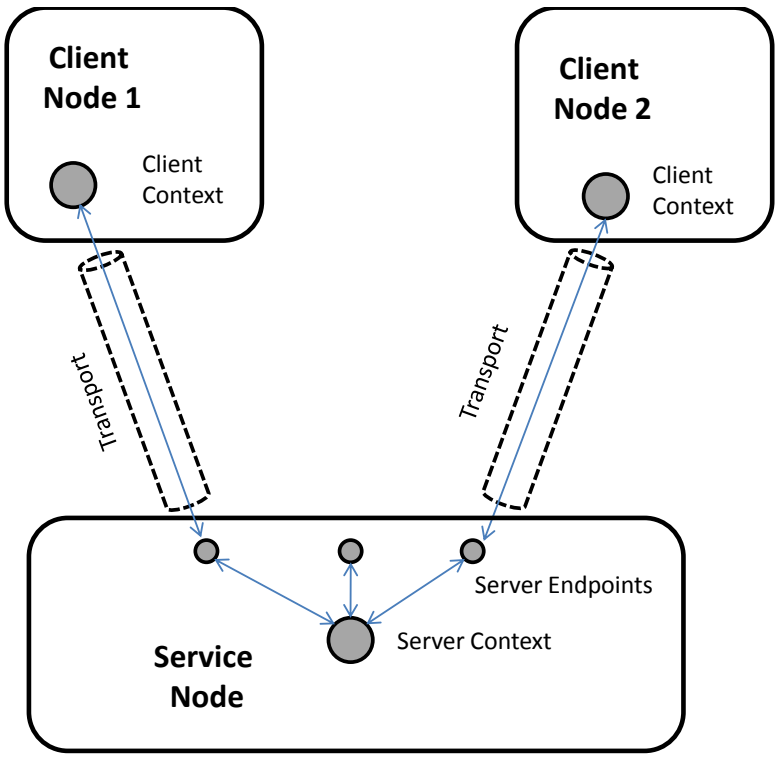

Figure 2: Configuration of Client-Service communication with multiple clients

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 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[**?**] 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[**?**] 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 cus-

tomized 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 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.

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.

The first line in the service definition contains the keyword `service` followed by the name of the service type. "Namespaces" follow similar rules to Java package names. For experimental software, the name should be prefixed with "experimental", for example "experimental.create2". 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.create2". 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 "stdver" and the minimum version of Robot Raconteur required to access the service. For now this should be "0.9". 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.)

Figure 3: Photo of the example robot

Example 1 Service definition file “experimental.create2.robdef”

```
#Service to provide sample interface to the iRobot Create
service experimental.create2

stdver 0.9

struct SensorPacket
        field uint8 ID
        field uint8[] Data
end

object Create
        constant int16 DRIVE_STRAIGHT 32767
        constant int16 SPIN_CLOCKWISE -1
        constant int16 SPIN_COUNTERCLOCKWISE 1

        function void Drive(int16 velocity, int16 radius)

        function void StartStreaming()
        function void StopStreaming()

        property int32 DistanceTraveled [readonly]
        property int32 AngleTraveled [readonly]
        property uint8 Bumpers [readonly]

        event Bump()

        wire SensorPacket packets [readonly]

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

2.1 Value types

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

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.

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
cdouble	16	Complex double precision floating point
csingle	8	Complex single precision floating point
bool	1	Logical boolean

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` keyword. 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.

Pods

Pods (short for "plain-old-data") are similar to structures, but are more restricted to ensure they have the same size. All data stored in pods are stored contiguously (c-style), while structs use pointers to the data. Pods can only contain pods, arrays of pods (fixed or max length), namedarrays, and namedarrays arrays (fixed or max length). Only numeric primitives may be used; strings, structs, lists, and maps may not be stored in pods. A pod is started with the keyword `pod` followed by the pod name. It is ended with the `end` keyword. The entries in the pod are defined with the keyword `field` followed by the type, and finally the name of the field. If a pod from a different service definition is used, first the referenced service definition is imported at the top of the service definition and the pod is referenced by the external service definition "dot" the name of the pod. Pods can be used with arrays and multi-dim arrays.

Namedarrays

Namedarrays are a union type designed to store numeric arrays that also have specific meanings attached to each entry. An example is a 3D vector. The vector can either be viewed as a 3x1 array, or as a structure containing (x,y,z). A namedarray stores the contained data as a primitive array, but allows the data to be viewed as a structure. Namedarrays should be used when possible since they have the most compact memory format. Namedarrays can only contain numeric primitives, fixed numeric primitive arrays (no multidimarrays), other namedarrays (with the same numeric type), and fixed arrays of namedarrays. A namedarray is started with the keyword `namedarray` followed by the namedarray name. It is ended with the `end` keyword. The entries in the namedarray are defined with the keyword `field` followed by the type, and finally the name of the field. If a namedarray from a different service definition is used, first the referenced service definition is imported at the top of the service definition and the namedarray is referenced by the external service definition "dot" the name of the namedarray. Namedarrays can be used with arrays and multi-dim arrays.

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,
`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.

Enums Enums are a special representation of int32 that names each value. Enums are aliases, with the value be stored as int32 internally. An enum is started with the keyword `enum` followed by the enum name. It is ended with the `end` keyword. The values are specified with a "name" = "value" format, separated by commas. Values can be signed integers, unsigned hexadecimal, or omitted to implicitly increment from the last value.

```
enum myenum
    value1 = -1,
    value2 = 0xF1,
    value3,
    value4
end
```

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

Note: structs, maps, and lists can be null. All other types are not nullable. (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`. Objects have *members* that implement functionality. Within Robot Raconteur there are eight types of members: Properties, Functions, Events, ObjRefs, Pipes, Callbacks, Wires, and Memories . They are defined between `object` and `end`.

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
```

Properties can use modifiers `readonly`, `writeonly`, `urgent`, and/or `perclient`. See Section 12.

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)
```

Functions can also return a ”generator,” which is a form of iterator. (These generators are modeled after Python generators.) This is useful for long running operations or to return large amounts of data. Generators take three forms. The first is when each iteration of the generator takes a parameter and returns a value. This takes the form:

```
function double{generator} addManyNumbers(int32 a, double{generator} b)
```

In this example, the “a” parameter is sent with the function call, while “b” and “return” are sent and received using the “Next” function of the generator.

The next form of the generator returns a value each iteration of the generator.

```
function double{generator} getSequence(int32 a, double b)
```

In this example, “a” and “b” are sent with the function call, and “return” is returned using the “Next” function of the generator.

The last form takes a parameter each iteration.

```
function void accumulateNumbers(double{generator} b)
```

Note that the generator return must be “void” or a generator type. Each call to “Next” will receive a parameter.

Generators will throw either “StopIterationException” to signal that the generator is finished, or it will throw “OperationAbortedException” to signal that there was an error and the generator should be destroyed. Generators clients must call “Close” or “Abort” on a generator if a “StopIterationException” or other exception is not received.

Generators that represent long running operations should return from “Next” with updated status information at least every 10 seconds to prevent timeout errors.

Functions can use the `urgent` modifier. See Section 12.

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)
```

Note that events do not have flow control, so they should be used sparingly.

Events can use the `urgent` modifier. See Section 12.

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 is 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
```

Pipes can use modifiers `readonly`, `writeonly`, and `unreliable`. See Section 12.

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 as 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 10.) An example wire member definition:

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

Wires can use modifiers `readonly` or `writeonly`. See Section 12.

Memories (Keyword: `memory`)

Memories represent a random-access segment of numeric primitive arrays, numeric primitive multi-dim arrays, pod arrays, pod multi-dim arrays, namedarrays arrays, and namedarrays 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
```

Memories can use modifiers `readonly` or `writeonly`. See Section 12.

2.3 Constants

Constants can be specified using the `constant` keyword. The constants can be numbers, single dimensional arrays, or strings. Constants can exist either in the global service definition scope, in objects, or in structs.

```
constant uint32 myconst 0xFB
constant double[] myarray {10.3, 584.9, 594}
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 Using

To reduce the clutter in a service definition file, the "using" statement can be used to alias an imported type.

```
using example.importeddef.obj1
```

"as" can be used to change the name locally.

```
using exmaple.importeddef.obj1 as another_obj1
```

2.6 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

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.

For Windows and Mac OSX, use the `pip` command to install Robot Raconteur from PyPi:

```
pip install robotraconteur
```

Other packages are required to run the examples:

```
pip install pyserial pygame opencv
```

For Ubuntu, use the Robot Raconteur PPA:

```
sudo add-apt-repository ppa:robotraconteur/ppa sudo apt update
```

Once the PPA is configured, install the Robot Raconteur packages:

```
sudo apt install python-robotraconteur python3-robotraconteur
```

Other packages are required to run examples:

```
sudo apt install python-pygame python-opencv python-pyserial
```

Ubuntu packages are available for 16.04 (Xenial) and 18.04 (Bionic).

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 $\leftrightarrow$ 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 uses numpy arrays for all numeric arrays of all shapes.

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 `NewStructure`. The `NewStructure` command takes the fully qualified name of the structure, and an optional client object reference.

Pods are represented as `numpy.array` using custom `dtype`. These `dtype` are initialized using the `GetPodDType` command in `RobotRaconteurNode`. The `GetPodDType` command takes the fully qualified name of the structure, and an optional client object reference. The returned `dtype` can be used as parameter with the `numpy.zeros(shape, dtype)` to initialize an array with the pod type. Note that pods are always stored in `numpy.array`. For a scalar, use `numpy.zeros((1,),dtype)`. Note that `numpy.array` uses "array" style indexing for fields. For example, to access the "y" field in a 2 dimensional array at index (1,3), use `myarray[1][3]['y']`. This can be used to get or set the value.

Namedarrays are represented as `numpy.array` using custom `dtype`. These `dtype` are initialized using the `GetNamedArrayDType` command in `RobotRaconteurNode`. The `GetNamedArrayDType`

command takes the fully qualified name of the structure, and an optional client object reference. The returned `dtype` can be used as parameter with the `numpy.zeros(shape, dtype)` to initialize an array with the pod type. Note that pods are always stored in `numpy.array`. For a scalar, use `numpy.zeros((1,),dtype)`. " namedarray can be converted to a normal numeric array using the `NamedArrayToArray` command in `RobotRaconteurNode`. A normal numeric array can be converted to namedarray using `ArrayToNamedArray` command in `RobotRaconteurNode`. Note that the first dimension of the numeric array must match the total number of numeric elements in a scalar namedarray. The normal numeric arrays will have one more dimension than the namedarray. Note that `numpy.array` uses "array" style indexing for fields. For example, to access the "y" field in a 2 dimensional array at index (1,3), use `myarray[1][3]['y']`. This can be used to get or set the value.

Maps are `dict` in Python.

Lists are `list` in Python.

Enums are stored as `int` in Python.

3.2 Python Reference to Functions

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 objecs depending on the author). Consider a simple module "MyModule" shown in Example 2.

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")
```

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

Table 2: Robot Raconteur ↔ Python Type Map

Robot Raconteur Type	Python Type	Notes
double, single	`float`	
cdouble, csingle	`complex`	
int8, uint8, int16, uint16, int32, uint32, int64, uint64	`int` or `long`	Depends on sys.maxint size
double[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.float64)`
single[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.float32)`
int8[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.int8)`
uint8[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.uint8)`
int16[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.int16)`
uint16[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.uint16)`
int32[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.int32)`
uint32[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.uint32)`
int64[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.int64)`
uint64[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.uint64)`
cdouble[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.complex128)`
csingle[]	`numpy.array`	`numpy.array([1, 2, ...], dtype=numpy.complex64)`
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	*varies*	See text for more info
pods	`numpy.array`	See text for more info
namedarrays	`numpy.array`	See text for more info
enums	`int`	
varvalue	`RobotRaconteurVarValue`	See text for more info

Table 3: Members of Create object

Member	Description
`function void Drive(int16 velocity, int16 radius)`	Drives the create at `velocity` (mm/s) with `radius` (mm)
`function void StartStreaming()`	Starts the sensor packet streaming (Bumpers (17), Distance Traveled (19), Angle Traveled (20))
`function void StopStreaming()`	Stops the sensor packet streaming
`property int32 DistanceTraveled`	Total distance traveled (doesn't seem to be accurate...)
`property int32 AngleTraveled`	Total angle traveled (doesn't seem to be accurate...)
`property uint8 Bumpers`	Returns the byte with flags about the state of the bumper and wheel drops (See OI manual sensor packet id 7)
`event Bump()`	Event fired when the bumper goes from no contact to contact
`pipe SensorPacket packets`	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.
`callback uint8[] play_callback(int32 DistanceTraveled, int32 AngleTraveled)`	A callback that is called when the "Play" button is pressed and returns notes to play on the Create.

of all the capabilities the robot has to offer; it is intended to be instructive on the use of Robot 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 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.

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 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()
```

Note 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.

Example 4 Replacement code for the create server

```
def main():

    obj=Create_impl()
    comm_port=sys.argv[1]
    obj.Init(comm_port)

    with RR.ServerNodeSetup("experimental.create2.Create",2354):

        RRN.RegisterServiceTypeFromFile("experimental.create2")
        RRN.RegisterService("Create","experimental.create2.Create",obj)

        raw_input("Server started, press enter to quit...")

        obj.Shutdown()

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

A Robot Raconteur node requires several steps to initialize a service:

1. Assign the "NodeID" and "NodeName", or automatically generate random
2. Instantiate and register transports, begin listening for clients
3. Register the relevant service types (robdef)
4. Register the root object for the service

Each Robot Raconteur node is uniquely identified by a 128-bit UUID "NodeID". UUIDs are a widely used concept, and are statistically guaranteed to be unique when randomly generated[1]. A node also has a name, the "NodeName". A "NodeName" is intended to help clients find relevant services, and is not guaranteed to be unique. For client nodes, the "NodeID" is typically allowed to be automatically generated when needed, with the "NodeName" left emtpy. For a server node, the "NodeName" is normally specified, with the "NodeID" retrieved from a local cache based on the "NodeName". The "NodeID" is randomly generated the first time the "NodeName" is used, and is retrieved from the cache subsequently. TLS certificates for Robot Raconteur are assigned to the "NodeID". See *Robot Raconteur® Security using TLS and Certificates* for more details.

[1] The uniqueness guarantee depends on the quality of available entropy.

"Transports" are used to communicate between nodes. The currently available transports are `TcpTransport` for communication over a TCP/IP network, `LocalTransport` for communication between nodes running on the same computer, and `HardwareTransport` for communication over USB, Bluetooth, and PCIe. For most server nodes, the `TcpTransport` and `HardwareTransport` are configured to listen for incoming clients. The `TcpTransport` will listen for connections on a TCP port, while the `HardwareTransport` listens for connections on a file handle that is identified by the "NodeName" or "NodeID" of the server node[2]. If a TLS certificate is available, it can be loaded into the TCP transport. See *Robot Raconteur® Security using TLS and Certificates* for more details.

For most use cases, the Python class `ServerNodeSetup` can be used to initialize the server node. The `ServerNodeSetup` takes the "NodeName", the TCP listen port, and an optional set of flags as parameters. In Python, the `ServerNodeSetup` is used with the `with` statement. When the `with` statement scope is exited, the node is shut down.

Service types (stored in robdef files) can either be included in the Python source file as strings, or can be loaded from file. In this example, the service definition is loaded from a file using the `RRN.RegisterServiceDefinitionFromFile` function.

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 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

[2]The "NodeID" lookup is implemented using the the "StartServerAsNodeName" function in `LocalTransport`.

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&s
```

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.crea
```

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.

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 node when the object is registered as a service. Because the wire is marked `readonly` using a member modifier and the `packets` object attribute is not set, the node will assume that we want a `WireBroadcaster`. The node will create the attribute and assign a `WireBroadcaster`. The `WireBroadcaster` class is designed to send the same value to all connected clients. If the wire is marked `writeonly`, the node will provide a `WireUnicastReceiver` object. If the wire does not specify a direction, A `WireServer` is passed to the object through a property, which must be implemented by the object to receive the `WireServer`.

The `_SendSensorPackets` function is called by the serial receive thread when there is a new data packet. The `_SendSensorPackets` uses the `OutValue` of the `WireBroadcaster` to send the new value to all connected clients. 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.

Wires use the `InValue` and `OutValue` in `WireConnection` to send and receive values. For a `readonly` wire, the client will use the `InValue` while the service will use the `OutValue` property. Fir a `writeonly` wire, these roles are reversed and the client will use the `OutValue` property while the service will use the `InValue` property. If the wire does not specify the direction, both the client and service can use `InValue` and `OutValue`.

As of Version 0.9, wire clients can also "peek" and "poke" values. The peek and poke read the value synchronously without creating a streaming connection. (The behavior of "peek" and "poke" is similar to the behavior of properties.) `PeekInValue` is used to read the in value, while `PeekOutValue` and `PokeOutValue` are used to read and write the out value. (The "in" and "out" directions in the peek/poke functions are relative to the client.)

`WireConnection` also has the `LastValueReceivedTime` and `LastValueSentTime` to determine the last time that values were updated. These are relative to `InValue` and `OutValue` when using streaming data, and are received from the peek and poke functions as part of the return from the functions.

The `play_callback` member is assigned to the texttt_play_callback attribute of the `Create_impl` object by the node when the object is registered as a service. 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 `ServerNodeSetup` class by default will call `EnableNodeAnnounce`. This 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. The `RobotRaconteurNode` is shutdown automatically when the program exits.

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.createwebcam2` 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

Example 6 Service definition file “experimental.createwebcam2.robdef”

```
#Service to provide sample interface to webcams
service experimental.createwebcam2

stdver 0.9

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

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

object Webcam
    property string Name [readonly]
    function WebcamImage CaptureFrame()

    function void StartStreaming()
    function void StopStreaming()
    pipe WebcamImage FrameStream [readonly]

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

end

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

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_pipeconnect` function adds the passed
PipeEndpoint to the dictionary of connected `PipeEndpoint`s. 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.)

A `PipeBroadcaster` is used for this example. The `PipeBroadcaster` is similar to the `WireBroadcaster`, sending packets to all connected clients. While a `PipeBroadcaster` can be inferred for a `readonly` pipe and the attribute set in the same manner as a `WireBroadcaster`, for this example the `PipeBroadcaster` is initialized by the object so the `backlog` can be specified. The `backlog` is used for flow control. If there are more packets "in-flight" than the specified maximum, more will not be sent. The property "FrameStream" is implemented, with the getter initializing the `PipeBroadcaster`. The function `frame_threadfunc` demonstrates using the `PipeBroadcaster` to send frames to the clients.

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 in Python 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 and connection 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. Version 0.9 and up use a request-response method for autodiscovery. Nodes send a broadcast UDP packet every 55 seconds to announce that they are available and still listening. This low frequency broadcast is not frequent enough for nodes searching for services, so broadcasting nodes will also wait for request broadcast packets. Clients will send requests packets when they begin searching for services. Service nodes will then respond with their connection information, after a random backoff period to prevent congestion. By default, the TCP discovery is only enabled for IPv6 to reduce network traffic. IPv4 can be enabled by specifying different flags to the `TcpTransport` or `ServerNodeSetup`.

For the `LocalTransport`, the discovery information is stored on the filesystem.

The packet sent by the service nodes 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.

The auto-discovery functionality is automatically enabled by `ServerNodeSetup` and the `from RobotRaconteur.Cli import *` functions. To manually enable auto-discovery on transports, 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.create2.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)
```

Nodes can also be searched for by "NodeID" and "NodeName" separate from services. Use `FindNodeByID` and `FindNodeByName` in `RobotRaconteurNode`. These will return the "NodeID", "NodeName", and the possible "ConnectionURLs" without the query portion. Note that the URL returned by these functions is

7 Subscriptions

Subscriptions are an extension of the node auto-discovery that automatically forms connections to detected nodes. A subscription is created using the `SubscribeClients` function in `RobotRaconteurNode`. This function takes two arguments: a list of service types, and an optional `ServiceSubscriptionFilter`. The returned object is a `ServiceSubscription`, which can be used to retrieve the connected services. The `GetConnectedClients` function returns a `dict` of all the connected clients. The key contains the "NodeID" and service name of the connection. The value is the connected client object reference, which can be used directly. The `ClientConnected` and `ClientDisconnected` events in `ServiceSubscription` can be used to detect when clients are connected or the connection is lost.

Wires and Pipes can be subscribed once a `ServiceSubscription` is created. The `SubscribeWire` function takes the name of the member and returns a `WireSubscription` that can be used to interact with all connected wires. The `InValue` property will return the "most recent" value received from all wires. The `OutValue` property will set all wires to the specified value. For pipes, the `SubscripePipe` is used, and retuns a `PipeSubscription`. This function can receive packets using the `ReceivePacket` or `TryReceivePacket` functions. The received packets are not sorted and all received packets are retrieved in the order they arrived. The `SendPacket` function can be used to send packets to all connected services. Note that currently the Wire and Pipe subscriptions can only be made to the root object. Wires and Pipes for objects retrieved using ObjRefs cannot be used.

Authentication information can be sent as part of `ServiceSubscriptionFilter`.

See Appendix C.29 and **??** for more details.

The discovery information can be subscribed without creating connections to clients using the `SubscribeServiceInfo2` function in `RobotRaconteurNode`. This is essentially a constantly running version of `FindServiceByType`.

8 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 9. 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.

9 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”.

9.1 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.

9.2 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)
```

9.3 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:

`with RR.RobotRaconteurNode.ScopedMonitorLock(obj):` *your code*

The lock will be released when the `with` statement block is exited.

10 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`.

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.

11 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.

12 Member Modifiers

Member modifiers change the way a member behaves. Modifiers are specified between square brackets following the member definition. The members `DistanceTraveled`, `AngleTraveled`, `Bumbers`, and `packets` in Example 1 all use the `readonly` modifier. This means that the client can get these values but cannot set them. The available member modifiers are as follows:

readonly
Valid Members: `property`, `pipe`, `wire`, `memory`

Specifies a member as readonly. For a `wire`, the default implementation is `WireBroadcaster`. For a `pipe`, the default implementation is `PipeBroadcaster`. For `property` and `memory` the setters are either disabled or not declared.

writeonly
Valid Members: `property`, `pipe`, `wire`, `memory`

Specifies a member as write. For a `wire`, the default implementation is `WireUnicastReceiver`. There is no default implementation for `pipe`. For `property` and `memory` the getters are either disabled or not declared.

unreliable
Valid Members: `pipe`

Specifies that a pipe is unreliable, meaning that packets may be lost and and/or returned out of order. Note that the `unreliable` keyword cannot be used with a `PipeBroadcaster` with a `backlog` value set. The backlog flow control requires that all sent packets arrive at their destination.

perclient
Valid Members: `property`

Specifies that the value of a property is unique to the client. Most properties belong to the service and are the same for all connected clients. This modifier should be used for settings that are unique to each client such as session keys.

urgent
Valid Members: `property`, `function`, `event`

Specifies that a member requires high priority delivery. This should be used for operations like aborting a motion. **Note that this does not qualify as a method to implement an e-stop.**

13 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):`.

In Python 3, passing "None" for the handler will return a future that can be used with the "await" keyword.

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

14 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 13 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.

15 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.

16 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.

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(functools.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
```

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=mycandevice&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 one 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 shorthard. 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`. If the `RobotRaconteur.Client` convenience module is used, these are populated automatically.

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

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

RRN.RobotRaconteurVersion → `string`

Returns the version of the Robot Raconteur library.

RRN.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.

RRN.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.

RRN.RegisterTransport(*transport*) → None

Registers a transport with the node.

Parameters:

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

Return Value:

None

RRN.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.

RRN.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

RRN.AsyncConnectService(*url*, *username*, *credentials*, *servicelistener*, *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.
- *servicelistener* (`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* (`double`) - (optional) The timeout for the connect in seconds. Default is infinite.

Return Value:

None

RRN.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* (`double`) - (optional) The timeout for the disconnect in seconds. Default is infinite.

Return Value:

None

RRN.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:

(`<structtype>`) - The new structure

RRN.GetPodDType(*podtype*, *objectreference = None*) → `numpy.dtype`

Returns the numpy dtype for *podtype*.

Parameters:

- *podtype* (`string` or `unicode`) - The fully qualified type of the pod.
- *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:

(`numpy.dtype`) - The dtype for the specified podtype.

RRN.GetNamedArrayDType(*namedarraytype*, *objectreference = None*) → `numpy.dtype`

Returns the numpy dtype for *namedarraytype*.

Parameters:

- *namedarraytype* (`string` or `unicode`) - The fully qualified type of the namedarray.
- *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:

(`numpy.dtype`) - The dtype for the specified namedarray.

RRN.NamedArrayToArray(*namedarray*) → `numpy.array`

Converts a namedarray type into a primitive array with the namedarray number type. This function will return an array with one more dimension than the input array, with the first dimension set to the element count of the named array.

Parameters:

- *namedarray* (`numpy.array`) - The namedarray to convert stored in a `numpy.array`

Return Value:

(`numpy.dtype`) - The converted numeric array

RRN.ArrayToNamedArray(*array*, `dt`) → `numpy.array`

Converts a numeric array into a namedarray. The type of the namedarray is specified using `dt`, which is returned from `RRN.GetNamedArrayDType`. The input numeric array must have the correct numeric type, and the first dimension must match the element count of the namedarray. The output array will have one fewer dimensions than the input array.

Parameters:

- *array* (`numpy.array`) - The numeric array to convert
- *dt* (`numpy.dtype`) - The data type of the desired namedarray type

Return Value:

(`numpy.array`) - The converted namedarray

RRN.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.

RRN.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.

RRN.Shutdown() → None

Shuts down Robot Raconteur and closes all connections.

Parameters:

None

Return Value:

None

RRN.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

RRN.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.

RRN.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.

RRN.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.

RRN.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.

RRN.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.

RRN.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.

RRN.FindObjectType(*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.

RRN.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.

RRN.CloseService(*name*) → None

Closes the service with name *name*.

Parameters:

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

Return Value:

None

RRN.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

RRN.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

RRN.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

RRN.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

RRN.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

RRN.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

RRN.ScopedMonitorLock(*obj*) → `ScopedMonitorLock`

Returns a `ScopedMonitorLock` that can be used with the `with` statement to create and hold the monitor lock and automatically release it when the scope exits.

Parameters:

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

Return Value:

(`ScopedMonitorLock`) - The lock to use with `with` block

RRN.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

RRN.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

RRN.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

RRN.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

RRN.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

RRN.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

RRN.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.

RRN.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.

RRN.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.

RRN.AsyncFindServiceByType(*servicetype*, *transportschemes*, *handler*, *timeout=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

RRN.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

RRN.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

RRN.UpdateDetectedNodes() → None

Updates the detected nodes. Update must be called before `GetDetectedNodes`

Parameters:

None

Return Value:

None

RRN.AsyncUpdateDetectedNodes(*handler*) → None

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

Parameters:

None

Return Value:

None

RRN.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.

RRN.SubscribeService(*service_types*, *filter = None*) → `ServiceSubscription`

Creates a `ServiceSubscription` for the specified *service_types* with an optional filter.

Parameters:

- *service_types* (`list`) - A list of strings containing the desired service type(s)
- *filter* (`ServiceSubscriptionFilter`) - A filter that is used to choose which nodes to connect

Return Value:

(`ServiceSubscription`) - The created subscription

RRN.SubscribeServiceInfo2(*service_types*, *filter = None*) → `ServiceSubscription`

Creates a `ServiceInfo2Subscription` for the specified *service_types* with an optional filter. The `ServiceInfo2Subscription` returns the connection info as `ServiceInfo2` structures rather than connecting to the services

Parameters:

- *service_types* (`list`) - A list of strings containing the desired service type(s)
- *filter* (`ServiceSubscriptionFilter`) - A filter that is used to choose which nodes to track

Return Value:

(`ServiceInfo2Subscription`) - The created subscription

RRN.PostToThreadPool(*f*) → None

Posts a function to be called by the node's native thread pool.

Parameters:

- *f* (`function`) - The function to call expecting zero parameters.

Return Value:

None

RRN.EndpointInactivityTimeout ↔ `double`

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

RRN.TransportInactivityTimeout ↔ `double`

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

RRN.TransactionTimeout ↔ `double`

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

RRN.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.

RRN.ThreadPoolCount ↔ `integer`

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

RRN.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.

RRN.CreateTimer(*period*, *handler*, *oneshot = False*) → `Timer`

Creates a timer with the specified *period* in seconds that calls the specifies *handler* with a `TimerEvent`. Note that Start must be called on the new timer.

Parameters:

- *period* (`float`) - The period of the timer in seconds.
- *handler* (`function`) - The function to call every period. This function must accept one parameter of type `TimerEvent`
- *oneshot* (`bool`) - If true, timer will only call the specified handler once. If false, timer will repeate infinitely.

Return Value:

(`Timer`) - The created timer. Call `Start` on timer to start.

RRN.CreateRate(*frequency*) → `Rate`

Creates a rate object that can be used to stabilize the period of loops.

Parameters:

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

Return Value:

(`Return`) - The created rate. Call `Seep` each loop iteration to stabilize loop frequency.

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.

ReceivePacketWait(*timeout = RR_TIMEOUT_INFINITE*) → *

Waits for and receives the next the next available packet. The type will match the type of the pipe specified in the service definition.

Parameters:

- *timeout* (`double`) - (optional) The timeout for the next packet in seconds. Default is infinite.

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.

ReceivePacketWait(*timeout = RR_TIMEOUT_INFINITE*) → *

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

Parameters:

- *timeout* (`double`) - (optional) The timeout for the next packet in seconds. Default is infinite.

Return Value:

(*) - The next packet.

TryReceivePacketWait(*timeout = RR_TIMEOUT_INFINITE*) → `(boolean,*)`

Same as `ReceivePacketWait` but returns if a packet was received as a boolean instead of throwing an exception. The return value is a tuple containing if the packet was received successfully as a boolean and the packet.

Parameters:

- *timeout* (`double`) - (optional) The timeout for the next packet in seconds. Default is infinite.

Return Value:

(`(boolean,*)`) - A tuple containing a success boolean and 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.
- *timeout* (double) - (optional) The timeout for the send operation in seconds. Default is infinite.

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 ↔ `bool`

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

PacketAckReceivedEvent → `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 ↔ `function`

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

GetClientFunction(*endpoint*) → `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` objects 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):`

PeekInValue() → *

Retrieves the current "InValue" from the service using a request without creating a `WireConnection`

Parameters:

None

Return Value:

(*) - The value retrieved from the service

PeekOutValue() → *

Retrieves the current "OutValue" from the service using a request without creating a `WireConnection`

Parameters:

None

Return Value:

(*) - The value retrieved from the service

PokeOutValue(*value*) → None

Sets the "OutValue" on the service using a request without creating a `WireConnection`

Parameters:

- *value* (*) - The value to set "OutValue"

Return Value:

None

AsyncPeekInValue(*handler*, *timeout = RR_TIMEOUT_INFINITE*) → None

Asynchronously retrieves the current "InValue" from the service using a request without creating a `WireConnection`

Parameters:

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

Return Value:

None

AsyncPeekOutValue(*handler*, *timeout = RR_TIMEOUT_INFINITE*) → None

Asynchronously retrieves the current "OutValue" from the service using a request without creating a `WireConnection`

Parameters:

- *handler* (`function`) - The handler function that will be called by the thread pool when the request is complete. It must have the form `handler(outvalue,err):`. `err` will be None if no error occurs.

- *timeout* (`float`) - The request timeout in seconds. Default is infinite timeout.

Return Value:

None

AsyncPokeOutValue(*value*, *handler*, *timeout = RR_TIMEOUT_INFINITE*) → None

Asynchronously sets the current "OutValue" from the service using a request without creating a `WireConnection`

Parameters:

- *value* (*) - The value to set "OutValue"
- *handler* (`function`) - The handler function that will be called by the thread pool when the request is complete. It must have the form `handler(invalue,err):`. `err` will be None if no error occurs.
- *timeout* (`float`) - The request timeout in seconds. Default is infinite timeout.

Return Value:

None

PeekInValueCallback ↔ `function`

A function to call when the service receives a "PeekInValue" request. The function must have the form `def peek_func(ep):`, where *ep* is the endpoint calling the peek function. This request is implemented automatically by `WireBroadcaster` and `WireUnicastReceiver`

PeekOutValueCallback ↔ `function`

A function to call when the service receives a "PeekOutValue" request. The function must have the form `def peek_func(ep):`, where *ep* is the endpoint calling the peek function. This request is implemented automatically by `WireBroadcaster` and `WireUnicastReceiver`

PokeOutValueCallback ↔ `function`

A function to call when the service receives a "PeokeOutValue" request. The function must have the form `def poke_func(value, ts, ep):`, where *value* is the new "OutValue", *ts* is a the time-spec for the request, and *ep* is the endpoint calling the peek function. This request is implemented automatically by `WireBroadcaster` and `WireUnicastReceiver`

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.

TryGetInValue() → `(boolean,*)`

Tries to get "InValue", and retuns a tuple containing a success boolean and the current "InValue" if success is true.

Parameters:

None

Return Value:

(`(boolean,*)`) - Tuple containing if the operation was successful and the current "InValue" if success is true.

TryGetOutValue() → `(boolean,*)`

Tries to get “OutValue”, and retuns a tuple containing a success boolean and the current “OutValue” if success is true.

Parameters:

None

Return Value:

`((boolean,*))` - Tuple containing if the operation was successful and the current “OutValue” if success is true.

Close() → None

Closes the wire connection pair.

Parameters:

None

Return Value:

None

WaitInValueValid(*timeout = RR_TIMEOUT_INFINITE*) → None

Waits until “InValue” is valid, or timeout expires

Parameters:

- *timeout* (`double`) - (optional) The timeout to wait in seconds. Default is infinite.

Return Value:

None

WaitOutValueValid(*timeout = RR_TIMEOUT_INFINITE*) → None

Waits until “OutValue” is valid, or timeout expires

Parameters:

- *timeout* (`double`) - (optional) The timeout to wait in seconds. Default is infinite.

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 ↔ `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 → `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 ↔ `long`

A 64-bit integer representing the seconds.

nanoseconds ↔ `integer`

A 32-bit integer representing the nanoseconds.

TimeSpec(`seconds`, `nanoseconds`) → `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() $\rightarrow$ None

Adjusts value so that `nanoseconds` is positive.

Parameters:

None

Return Value:

None

Now() $\rightarrow$ `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

RequestObjectLock(*servicepath*, *username*) → None

Assign an object lock to a username from the service side. This function allows for arbitrary object locks on *servicepath*, which is the full encoded service path

Parameters:

- *servicepath* (`str` or `unicode`) - The encoded servicepath of the object to lock
- *username* (`str` or `unicode`) - The username to assign the lock

Return Value:

None

RequestClientObjectLock(*servicepath*, *username*, *endpoint*) → None

Assign an object lock to a specific client from the service side. This function allows for arbitrary object locks on *servicepath*, which is the full encoded service path

Parameters:

- *servicepath* (`str` or `unicode`) - The encoded servicepath of the object to lock
- *username* (`str` or `unicode`) - The username to assign the lock
- *endpoint* (`integer`) - The endpoint of the client to assign the lock

Return Value:

None

ReleaseObjectLock(*servicepath*, *username*, *override*) → None

Releases an object lock from the service side. This function allows for arbitrary object locks on *servicepath*, which is the full encoded service path

Parameters:

- *servicepath* (`str` or `unicode`) - The encoded servicepath of the object to unlock
- *username* (`str` or `unicode`) - The username that currently owns the lock
- *override* (`boolean`) - If true, the function will always force the object to unlock

Return Value:

None

GetObjectLockUsername(*servicepath*) → `string`

Returns the username of the owner of the object lock, or None if the object is not locked.

Parameters:

- *servicepath* (`str` or `unicode`) - The encoded servicepath of the object to unlock

Return Value:

(`string`) - The username of the lock owner

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
- RequestTimeoutException
- AuthenticationException
- ObjectLockedException
- UnknownException
- InvalidOperationException
- InvalidArgumentException
- OperationFailedException
- NullValueException
- InternalErrorException
- SystemResourcePermissionDeniedException
- OutOfSystemResourceException
- SystemResourceException
- ResourceNotFoundException
- IOException
- BufferLimitViolationException
- ServiceDefinitionException
- OutOfRangeException
- KeyNotFoundException
- ReadOnlyMemberException
- WriteOnlyMemberException
- NotImplementedException

- `MemberBusyException`
- `ValueNotSetException`
- `AbortOperationException`
- `OperationAbortedException`
- `StopIterationException`

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

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 assosciated 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 is 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 as 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 HardwareTransport

class **HardwareTransport**

The `HardwareTransport` provides the ability to connect to USB, Bluetooth, and PCIe devices. See the supplemental documentation for more information.

C.23 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 `PipeEndpoint`s. 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.24 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 `WireConnection`s. It automatically handles client connects, disconnects, and peeks.

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 `WireConnection`s. 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.

C.25 WireUnicastReceiver

class **WireUnicastReceiver**

Helper class that can be used in conjunction with a service side `Wire` member to receive the most recent

"OutValue" from the most recently connected `WireConnection`. It automatically handles client connects, disconnects, peeks, and pokes.

WireUnicastReceiver(*wire*) → `WireUnicastReceiver`

Creates a new `WireUnicastReceiver` 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 receiving

Return Value:

None

InValue ↔ *

Gets the most recent value recived from clients. Throws a `ValueNotSetException` if a value has not been received.

InValueChanged ↔ `EventHook`

Event called when the value changes. Handler should have the form `def handler(value, ts, ep)`.

C.26 Generator

class **Generator**

Generators are used by *generator functions*. For clients, the generators are returned from *generator function* calls. For services, the implementation returns a generator that implements the `Next`, `Close`, and `Abort` functions.

Next(*param = None*) → *

The "Next" function is the primary function for the generator. It receives zero or one parameter, and returns a return value or Null if the return type is *void*. This function is called repeatedly until the client closes the generator, or the "Next" function throws a `StopIterationException`. The `StopIterationException` should be treated as an expected signal that the generator is complete.

Parameters:

- *param* (*) - (Optional) The parameter to send to the generator "Next" function. This is only required if the generator has a parameter marked {generator}. Otherwise, this parameter should be None.

Return Value:

(*) - The return value from the "Next" call. If the generator function return type is void, this will be None.

Close() → None

Closes the generated. Internally, this works by sending the `StopIterationException` to the service. Once the generator is closed, future requests will return with an exception. The "Close" command is considered a clean operation and does not signal an error condition.

Parameters:

None

Return Value:

None

Abort() → None

Aborts the generated. Internally, this works by sending the `AbortOperationException` to the service. Once the generator is aborted, future requests will return with an exception. The "Aborted" command is considered an error condition, and the operation that the generator represents should be immediately aborted and all data discarded. If this generator represents a physical action such as a robot motion, the motion should be stopped immediately.

Parameters:

None

Return Value:

None

AsyncNext(*param*, *handler*, *timeout = RR_TIMEOUT_INFINITE*) → None

An asynchronous version of the "Next" function. Note that this will not be called on the service side. The service will always call `Next`.

Parameters:

- *param* (*) - (Optional) The parameter to send to the generator "Next" function. This is only required if the generator has a parameter marked {generator}. Otherwise, this parameter should be None.
- *handler* (`function`) - The handler for when the operation is complete. This should have the form `def handler(ret,exp):`.
- *timeout* (`double`) - (optional) The timeout for the operation in seconds. Default is infinite.

Return Value:

None

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

An asynchronous version of the "Close" function. Note that this will not be called on the service side. The service will always call `Close`

Parameters:

- *handler* (`function`) - The handler for when the operation is complete. This should have the form `def handler(exp):`.
- *timeout* (`double`) - (optional) The timeout for the operation in seconds. Default is infinite.

Return Value:

None

AsyncAbort(*handler*, *timeout = RR_TIMEOUT_INFINITE*) → None

An asynchronous version of the "Abort" function. Note that this will not be called on the service side. The service will always call `Abort`

Parameters:

- *handler* (`function`) - The handler for when the operation is complete. This should have the form `def handler(exp):`.
- *timeout* (`double`) - (optional) The timeout for the operation in seconds. Default is infinite.

Return Value:

None

C.27 Timer

class **Timer**

The `Timer` class calls a `handler` function repeatedly at a specified period. The `Timer` class is created using the `CreateTimer` function in `RobotRaconteurNode`.

Start() → None

Starts the timer. Must be called after timer creation

Parameters:

None

Return Value:

None

Stop() → None

Stops the timer.

Parameters:

None

Return Value:

None

IsRunning() → `bool`

Returns True is the timer is running.

Parameters:

None

Return Value:

(`bool`) - Returns True if timer is running

C.28 Rate

class **Rate**

The `Rate` class is used to stabilize the frequency of loops. The `Rate` class is created using the `CreateRate` in `RobotRaconteurNode`. Each iteration of the loop, call `Sleep`, which will wait until the correct time for the next iteration.

Sleep() → None

Wait until the correct time for the next loop iteration.

Parameters:

None

Return Value:

None

C.29 ServiceSubscription

class **ServiceSubscription**

Represents a service subscription that automatically connects to relevant services. The subscription is created using the `SubscribeServices` in `RobotRaconteurNode`.

GetConnectedClients() → `dict`

Returns the currently connected clients. The return contains a dictionary with the keys of type `ServiceSubscriptionClientID` and the values being the connected clients.

Parameters:

None

Return Value:

(`dict`) - Dictionary containing connected services

Close() → None

Closes and destroys the subscription. **All clients connected through this subscription will be closed.**

Parameters:

None

Return Value:

None

SubscribePipe(*pipe_name*) → `PipeSubscription`

Creates a `PipeSubscription` based on the pipe name. This is only available for pipes in the root object of the connect client.

Parameters:

- *pipe_name* (`str` or `unicode`) - The member name of the pipe

Return Value:

(`PipeSubscription`) - The created `PipeSubscription`

SubscribeWire(*wire_name*) → `WireSubscription`

Creates a `WireSubscription` based on the wire name. This is only available for pipes in the root object of the connect client.

Parameters:

- *wire_name* (`str` or `unicode`) - The member name of the wire

Return Value:

(`WireSubscription`) - The created `WireSubscription`

ConnectRetryDelay ↔ `float`

The delay before attempting to reconnect to a service after connection is lost.

ClientConnect ↔ `EvtHook`

Event fired when a client is connected. Handler should be of the form `def handler(subscription, client_id, client)`.

ClientDisconnected ↔ `EvtHook`

Event fired when a client is disconnected. Handler should be of the form `def handler(subscription, client_id, client)`.

C.30 PipeSubscription

class **PipeSubscription**

Represents a subscription to pipes in connected clients. Created through the `SubscribePipe` in `ServiceSubscription`.

ReceivePacket() → *

Receives the next available packet. All packets received from connect clients are stored in a central queue.

Parameters:

None

Return Value:

(*) - The next packet

ReceivePacketWait(*timeout = RR_TIMEOUT_INFINITE*) → *

Waits for and receives the next the next available packet from the shared receive.

Parameters:

- *timeout* (`double`) - (optional) The timeout for the next packet in seconds. Default is infinite.

Return Value:

(*) - The next packet

TryReceivePacketWait(*timeout = RR_TIMEOUT_INFINITE*) → `(boolean,*)`

Same as `ReceivePacketWait` but returns if a packet was received as a boolean instead of throwing an exception. The return value is a tuple containing if the packet was received successfully as a boolean and the packet.

Parameters:

- *timeout* (`double`) - (optional) The timeout for the next packet in seconds. Default is infinite.

Return Value:

(`(boolean,*)`) - A tuple containing a success boolean and the next packet

TryReceivePacket() → `(boolean,*)`

Same as `TryReceivePacketWait` but with timeout set to zero

Parameters:

None

Return Value:

((boolean,*)) - A tuple containing a success boolean and the next packet

Available → integer

The total number of packets available in the receive queue

ActivePipeEndpointCount → integer

The total number of pipe endpoints currently connected

Close() → None

Closes the PipeSubscription

Parameters:

None

Return Value:

None

PipePacketReceived ↔ EvtHook

Event fired when a new pipe packet is available in the receive queue. The handler should be of the form `def handler(pipe_subscription):`. Note that the handler receives a reference to the pipe subscription. The value must be read using one of the receive functions in the pipe subscriptions.

C.31 WireSubscription

class **WireSubscription**

Represents a subscription to wires in connected clients. Created through the SubscribeWire in ServiceSubscription.

InValue ↔ *

Gets the most recent value recived from client connections. Throws a ValueNotSetException if a value has not been received.

InValue ↔ (*,TimeSpec)

Gets the most recent value recived from client connections and the timestamp of the value. Throws a ValueNotSetException if a value has not been received.

WaitInValueValid(*timeout = RR_TIMEOUT_INFINITE*) → None

Waits until "InValue" is valid, or timeout expires

Parameters:

- *timeout* (double) - (optional) The timeout to wait in seconds. Default is infinite.

Return Value:

None

ActiveWireConnectionCount → `integer`

The total number of wire clients connected.

IgnoreInValue ↔ `bool`

Set to True to ignore all received values. Use if the subscription will only use the "SetOutValue" function.

SetOutValueAll(*value*) → None

Sets all connect wire clients "OutValue". This is essentially a reverse `WireBroadcaster`.

Parameters:

- *value* (*) - The value to send to all wires.

Return Value:

None

Close() → None

Closes the `WireSubscription`

Parameters:

None

Return Value:

None

WireValueChanged ↔ `EvtHook`

Event fired when the "InValue" changes. An event triggered when `InValue` has changed. The event callback function should have the form `def callback(wiresubscription, value, timestamp):`

C.32 ServiceSubscriptionFilter

class **ServiceSubscriptionFilter**

The `ServiceSubscriptionFilter` is used to filter which nodes should be connected by the subscription.

Nodes ↔ `list`

A list of `ServiceSubscriptionFilterNode` that should be connected. Authentication information is specified in this field.

ServiceNames ↔ `list`

A list of strings containing the names of services to connect.

TranportSchemes ↔ `list`

A list of strings containing the schemes types that should be used to form connections.

Predicate ↔ `function`

A callback that returns a bool if the client should be connected. The handler should be of the form `def cb(serviceinfo2):`.

MaxConnections ↔ `integer`

The maximum number of clients that should be connected.

C.33 ServiceSubscriptionFilterNode

class **ServiceSubscriptionFilterNode**

Specifies the node that should be connected and optionally authentication information.

NodeID ↔ `NodeID`

The “NodeID” that should be connected, or “NodeID.Any” to match any node.

NodeName ↔ `str`

The name of the node to connect. `Node` for any “NodeName”

Username ↔ `str`

The username for authentication. Leave as “None” for no authentication

Credentials ↔ `dict`

Dictionary containing credentials for authentication

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"

class Create_impl(object):
    def __init__(self):
        self.Bump=RR.EventHook()
        self._lock=threading.RLock()
        self._recv_lock=threading.RLock()
        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 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.create2.SensorPacket')
        #Set the data
```

```
        strt.ID=seed
        strt.Data=data

        #Set the OutValue for the broadcaster
        self.packets.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():

    #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)

    with RR.ServerNodeSetup("experimental.create2.Create",2354):

        #Register the service type and the service
        RRN.RegisterServiceTypeFromFile("experimental.create2")
        RRN.RegisterService("Create","experimental.create2.Create",obj)

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

        #Shutdown
        obj.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

#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(3,320)
            self._capture.set(4,240)

    #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.createwebcam2.WebcamImage")
            ret, frame=self._capture.read()
            if not ret:
                raise Exception("Could not read from webcam")
            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):
        with self._lock:
            #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.createwebcam2.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.createwebcam2.Webcam"

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



def main():

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


    with RR.ServerNodeSetup("experimental.createwebcam2.WebcamHost",2355):

        RRN.RegisterServiceTypeFromFile("experimental.createwebcam2")
        RRN.RegisterService("Webcam","experimental.createwebcam2.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()


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
    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
    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
    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():
```

```
    c=AsyncCreateClient(client_handler)
    c.start()

    ev.wait()

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

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

E Software License

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and

subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, **any** Contribution intentionally submitted **for** inclusion **in** the Work by You to the Licensor shall be under the terms **and** conditions of this License, without **any** additional terms **or** conditions. Notwithstanding the above, nothing herein shall supersede **or** modify the terms of **any** separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does **not** grant permission to use the trade names, trademarks, service marks, **or** product names of the Licensor, **except** as required **for** reasonable **and** customary use **in** describing the origin of the Work **and** reproducing the content of the NOTICE **file**.

7. Disclaimer of Warranty. Unless required by applicable law **or** agreed to **in** writing, Licensor provides the Work (**and** each Contributor provides its Contributions) on an "AS␣IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express **or** implied, including, without limitation, **any** warranties **or** conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, **or** FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible **for** determining the appropriateness of using **or** redistributing the Work **and** assume **any** risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event **and** under no legal theory, whether **in** tort (including negligence), contract, **or** otherwise, unless required by applicable law (such as deliberate **and** grossly negligent acts) **or** agreed to **in** writing, shall **any** Contributor be liable to You **for** damages, including **any** direct, indirect, special, incidental, **or** consequential damages of **any** character arising as a result of this License **or** out of the use **or** inability to use the Work (including but **not** limited to damages **for** loss of goodwill, work stoppage, computer failure **or** malfunction, **or any and all** other commercial damages **or** losses), even **if** such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty **or** Additional Liability. While redistributing the Work **or** Derivative Works thereof, You may choose to offer, **and** charge a fee **for**, acceptance of support, warranty, indemnity, **or** other liability obligations **and**/**or** rights consistent with this License. However, **in** accepting such obligations, You may act only on Your own behalf **and** on Your sole responsibility, **not** on behalf of **any** other Contributor, **and** only **if** You agree to indemnify, defend, **and** hold each Contributor harmless **for any** liability incurred by, **or** claims asserted against, such Contributor by reason of your accepting **any** such warranty **or** additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to **apply** the Apache License to your work.

To **apply** the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't␣include
␣␣␣␣␣␣the␣brackets!)␣␣The␣text␣should␣be␣enclosed␣in␣the␣appropriate
␣␣␣␣␣␣comment␣syntax␣for␣the␣file␣format.␣We␣also␣recommend␣that␣a
␣␣␣␣␣␣file␣or␣class␣name␣and␣description␣of␣purpose␣be␣included␣on␣the
␣␣␣␣␣␣same␣"printed␣page"␣as␣the␣copyright␣notice␣for␣easier
␣␣␣␣␣␣identification␣within␣third-party␣archives.

␣␣␣Copyright␣2011-2019␣Wason␣Technology,␣LLC

␣␣␣Licensed␣under␣the␣Apache␣License,␣Version␣2.0␣(the␣"License");
␣␣␣you␣may␣not␣use␣this␣file␣except␣in␣compliance␣with␣the␣License.
␣␣␣You␣may␣obtain␣a␣copy␣of␣the␣License␣at

```
␣␣␣␣␣␣␣http://www.apache.org/licenses/LICENSE-2.0

␣␣␣Unless␣required␣by␣applicable␣law␣or␣agreed␣to␣in␣writing,␣software
␣␣␣distributed␣under␣the␣License␣is␣distributed␣on␣an␣"AS␣IS"␣BASIS,
␣␣␣WITHOUT␣WARRANTIES␣OR␣CONDITIONS␣OF␣ANY␣KIND,␣either␣express␣or␣implied.
␣␣␣See␣the␣License␣for␣the␣specific␣language␣governing␣permissions␣and
␣␣␣limitations␣under␣the␣License.
```