



Robot Raconteur® using C#

Version 0.8 Beta

<http://robotraconteur.com>

Dr. John Wason

Wason Technology, LLC

PO Box 669

Tuxedo, NY 10987

wason@wasontech.com

<http://wasontech.com>

May 3, 2016

Contents

1	Introduction	3
2	“Thunk” Code	3
3	Using RobotRaconteurNET.dll	4
4	C# ↔ Robot Raconteur Data Type Mapping	4
5	Service Objects	4
6	Asynchronous Operations	7
7	Conclusion	7
A	Robot Raconteur Reference	7
A.1	RobotRaconteurNode	7
A.2	ClientServiceListenerDelegate	24
A.3	MultiDimArray	24
A.4	ServiceInfo2	26
A.5	NodeInfo2	26
A.6	Pipe<T>	27
A.7	Pipe<T>.PipeEndpoint	28
A.8	Callback<T>	31
A.9	Wire<T>	31
A.10	Wire<T>.WireConnection	32
A.11	TimeSpec	34
A.12	ArrayMemory<T>	35
A.13	MultiDimArrayMemory<T>	37
A.14	ServerEndpoint	38
A.15	ServerContext	39
A.16	ServerServiceListenerDelegate	40
A.17	AuthenticatedUser	41
A.18	NodeID	41
A.19	PasswordFileUserAuthenticator	43
A.20	ServiceSecurityPolicy	43
A.21	RobotRaconteurException	44
A.22	RobotRaconteurRemoteException	45
A.23	Transport	45
A.24	LocalTransport	46
A.25	TcpTransport	47
A.26	CloudTransport	53
A.27	UsbTransport	53
A.28	PipeBroadcaster<T>	54
A.29	WireBroadcaster<T>	55

1 Introduction

This document serves as the reference for using Robot Raconteur™ with C#. This is a supplement that only focuses on issues related to the C# language bindings. It is assumed the user has reviewed “Introduction to Robot Raconteur™ using Python” which serves as the primary reference for the Robot Raconteur™ library itself. This document refers to the iRobot Create example software. Because of the verbosity of the code, it will not be reproduced in this document. The reader should download the software examples for reference.

2 “Thunk” Code

When using Robot Raconteur with Python, the individual types defined in service definitions are automatically marshaled between the software and the Robot Raconteur library. When using Robot Raconteur with C#, it is necessary to generate code at compile time because C# has limited dynamic typing. This is accomplished using the command line tool “RobotRaconteurGen”, which can be found in the C++ SDK download. The service definitions should be stored in files ending with the extension “.robdef”. The following example will generate the source files for the iRobot Create example:

```
RobotRaconteurGen --thunksource --lang=csharp Create_interface.robdef
```

The last argument “Create_interface.robdef” should be replaced with the desired service definition file, and can also be a list. It is also possible to list multiple service definition files for the command. If the “import” statement is used in the service definition, all imported service definitions must be listed in the same command.

Each service definition will create two files. For the “Create_interface.robdef” file it will produce two files, “Create_interface.cs” and “Create_interface_stubskel.cs”. The generated files should be added to the project. The generated classes will be in a namespace with the same name as the service definition name. Software using “Create_interface” can use the line

```
using Create_interface;
```

to import the generated classes and interfaces.

The service definition will result in classes that represent the Robot Raconteur structures, interfaces for the Robot Raconteur objects, and “thunk” code that handles the service and client marshalling. Service objects must implement the corresponding interface in order to be registered as service objects. The client will receive object references that implement the corresponding interface for the Robot Raconteur object type. This is frequently called a “stub”.

3 Using RobotRaconteurNET.dll

The library `RobotRaconteurNET.dll` contains the Robot Raconteur implementation for C#. It must be added as a reference to the C# project using the “Add Reference” command. The classes all reside in the `RobotRaconteur` namespace. The following command should be used to import the namespace near the top of the C# source files:

```
using RobotRaconteur;
```

The software library for Robot Raconteur is a thin C# library that primarily references the native C++ library. This thin library is called a “language binding” that allows a high-level language like C# to utilize the low level C++ library. For Windows, the C++ library is stored within the main DLL so there are no extra files. At the very beginning of the program, the following line must be executed:

```
RobotRaconteurNativeLoader.Load();
```

This command will load the native library for use by the program. On non-Windows platforms, it is necessary to have the shared libraries available externally. The contents of the “mono” directory must be in the same directory as `RobotRaconteurNET.dll`. On Windows the files in this directory are not necessary.

4 C# ↔ Robot Raconteur Data Type Mapping

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

As seen in Table 1, most of the data type maps are straightforward. The maps are stored in Dictionaries from namespace `System.Collections.Generic`. The `MultiDimArray` type contains the array stored as flat arrays in column-major order, which is frequently called “Fortran order”. This means that the columns are “stacked” on top of each other to create the flat array. This is opposite of C which uses row-major order. See Section A.3 for more details on this class.

5 Service Objects

The service objects in Robot Raconteur are defined by interfaces in C#. `RobotRaconteurGen` generates out the code that is specific to each service definition, and it also generates the interfaces that correspond to each object type. On the service side, the service objects must implement the corresponding interface by extending the interface and implementing all the members. The following section describes how member types are mapped to C#. See the example code for demonstrations of how these are used in practice.

Table 1: Robot Raconteur ↔ C# Type Map

Robot Raconteur Type	C# Type	Notes
double	double	
single	float	
int8	sbyte	
uint8	byte	
int16	short	
uint16	ushort	
int32	int	
uint32	uint	
int64	long	
uint64	ulong	
double[]	double[]	
single[]	float []	
int8[]	sbyte []	
uint8[]	byte []	
int16[]	short []	
uint16[]	ushort []	
int32[]	int []	
uint32[]	uint []	
int64[]	long []	
uint64[]	ulong []	
string	string	Strings are transmitted as UTF-8 but converted to normal C# strings
$T\{\text{int32}\}$	Dictionary<int, T >	Map type, T is a template
$T\{\text{string}\}$	Dictionary<string, T >	Map type, T is a template
$T\{\text{list}\}$	List< T >	List type, T is a template
structure	<i>varies</i>	Use generated class for corresponding structure.
$N[*]$	MultiDimArray	Multi-dim array of type N
varvalue	object	
varobject	object	

property

Properties are implemented as standard C# properties. They are always declared `public virtual`.

function

Functions are implemented as standard C# functions. They are always declared `public virtual`.

event

Events are implemented as standard C# events. The types are specified using the `Action` and `Func` template delegates which are part of the standard `System` namespace. They take the parameter and return type (if `Func`) as template arguments. See the .NET documentation for more information on `Action` and `Func`.

objref

The `objref` members are implemented through a function that is named “get_” pre-pended to the member name of the `objref`. The index is the argument to the function if there is an index. Note that on services the object will not be released until `ServerContext.ReleaseServicePath` is called.

pipe

Pipes are implemented using properties in the same manner as in Python. On the client, the property can be accessed to retrieve the `Pipe<T>` object. On the service, the `PipeConnectCallback` can be set to a function to receive the connected `Pipe<T>.PipeEndpoint`.

callback

Callbacks are implemented using properties in the same manner as Python. On the client side, the `Function` field in the `Callback<T>` object is set to the desired function. The type `T` is specified using `Action` or `Func` template delegates similar to events. On the service side, the function `GetClientFunction(uint e)` function is used to retrieve a delegate that will call the function on the client based on the `Robot Raconteur` endpoint `e` corresponding to the client.

wire

Wires are implemented using properties in the same manner as in Python. On the client, the property can be accessed to retrieve the `Wire<T>` object. On the service, the `WireConnectCallback` can be set to a function to receive the connected `Wire<T>.WireConnection`.

memory

Memories are implemented using properties in the same manner as in Python. A template type `T` is used to specify the numeric type of the `ArrayMemory<T>` or `MultiDimArrayMemmmory<T>`.

6 Asynchronous Operations

C# provides asynchronous operations in a similar manner to Python. The asynchronous functions in the built in library are similar to the original synchronous form except they return `void`, are prefixed with “Async”, and has two extra parameters: *handler* and *timeout*. The handler is either of the form `Action<Exception>` if the original function returns `void` or `Action<T,Exception>` if the original function returns type `T`. The timeout is in milliseconds.

Client object references can be used asynchronously for properties, functions, and objrefs. The asynchronous functions are accessed by using the asynchronous interfaces. The asynchronous interface is the name of the original service object type prefixed by “`async_`”. The asynchronous interface for `Create` would be `async_Create`.

The asynchronous names for members are similar by appending “`async_get_`”, “`async_set_`”, and “`async_`” in the same manner as in Python. The extra handler and timeout parameters are appended.

The C# Robot Raconteur library also supports the “Tasks” framework. It has the same form as the other asynchronous operations but without the handler parameter. This can be used in combination with `async/await` framework. See the documentation for C# asynchronous programming for more information on how to use this method.

7 Conclusion

This document serves as a reference for the C# bindings of Robot Raconteur. More information can be found on the project website.

A Robot Raconteur Reference

A.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.

```
string RobotRaconteurNode.s.RobotRaconteurVersion
```

Returns the version of the Robot Raconteur library.

string **RobotRaconteurNode.s.NodeName**

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

NodeID **RobotRaconteurNode.s.NodeID**

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

void **RobotRaconteurNode.s.RegisterTransport**(Transport *transport*)

Registers a transport with the node.

Parameters:

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

Return Value:

None

object **RobotRaconteurNode.s.ConnectService**(string *url*, string *username=null*, Dictionary<string,object> *credentials=null*, ClientServiceListenerDelegate *servicelistener=null*, string *objecttype=null*)

Creates a connection to a remote service located by the *url*. The *username* and *credentials* are optional if authentication is used. The *objecttype* parameter is the fully qualified type of the root service object. This should normally always be used even though it is optional.

Parameters:

- *url* (string) - The url to connect to.
- *username* (string) - (optional) The username to use with authentication.
- *credentials* (Dictionary<string,object>) - (optional) The credentials to use with authentication.
- *servicelistener* (ClientServiceListenerDelegate) - (optional) A function to call when a client event is generated such as disconnect.
- *objecttype* (string) - (optional) The fully qualified type of the root service object.

Return Value:

(object) - The connected object. This is a Robot Raconteur object reference that provides access to the remote service object. It should be cast to the C# interface type corresponding to the Robot Raconteur object type.

```
object RobotRaconteurNode.s.ConnectService(string[] url, string username=null, Dictionary<string,object> credentials=null, ClientServiceListenerDelegate servicelistener=null, string objecttype=null)
```

Creates a connection to a remote service located by the *url*. The *username* and *credentials* are optional if authentication is used. The *objecttype* parameter is the fully qualified type of the root service object. This should normally always be used even though it is optional.

Parameters:

- *url* (string[]) - An array of candidate URLs to use to connect to the service. All will be attempted and the first one to connect will be used.
- *username* (string) - (optional) The username to use with authentication.
- *credentials* (Dictionary<string,object>) - (optional) The credentials to use with authentication.
- *servicelistener* (ClientServiceListenerDelegate) - (optional) A function to call when a client event is generated such as disconnect.
- *objecttype* (string) - (optional) The fully qualified type of the root service object.

Return Value:

(object) - The connected object. This is a Robot Raconteur object reference that provides access to the remote service object. It should be cast to the C# interface type corresponding to the Robot Raconteur object type.

```
void RobotRaconteurNode.s.DisconnectService(object obj)
```

Disconnects a service.

Parameters:

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

Return Value:

None

```
void RobotRaconteurNode.s.AsyncConnectService(string url, string username, Dictionary<string,object> credentials, ClientServiceListenerDelegate servicelistener, string objecttype, Action<object,Exception> handler, int timeout=RR.TIMEOUT_IN
```

This function is the asynchronous version of `ConnectService`. The parameters are the same except for the last two that provide the asynchronous connection handler and the timeout. If there is an error, the `Exception` will not be null. Otherwise it will be null and the object will be passed to the handler.

Parameters:

- *url* (string) - The url to connect to.
- *username* (string) - The username to use with authentication.
- *credentials* (Dictionary<string,object>) - The credentials to use with authentication.
- *servicelistener* (ClientServiceListenerDelegate) - A function to call when a client event is generated such as disconnect.
- *objecttype* (string) - The fully qualified type of the root service object.
- *handler* (Action<object,Exception>) - The handler function. The connected object is passed as the first parameter. The second parameter will be null on success, otherwise an `Exception` instance is passed.
- *timeout* (int) - (optional) The timeout for the call in milliseconds. Default is infinite timeout.

Return Value:

None

```
void RobotRaconteurNode.s.AsyncConnectService(string[] url, string username, Dictionary<string,object> credentials, ClientServiceListenerDelegate servicelistener, string objecttype, Action<object,Exception> handler, int timeout=RR.TIMEOUT_IN
```

This function is the asynchronous version of `ConnectService`. The parameters are the same except for the last two that provide the asynchronous connection handler and the timeout. If there is an error, the `Exception` will not be null. Otherwise it will be null and the object will be passed to the handler.

Parameters:

- *url* (string[]) - An array of candidate URLs to use to connect to the service. All will be attempted and the first one to connect will be used.
- *username* (string) - The username to use with authentication.
- *credentials* (Dictionary<string,object>) - The credentials to use with authentication.
- *servicelistener* (ClientServiceListenerDelegate) - A function to call when a

client event is generated such as disconnect.

- *objecttype* (string) - The fully qualified type of the root service object.
- *handler* (Action<object,Exception>) - The handler function. The connected object is passed as the first parameter. The second parameter will be null on success, otherwise an Exception instance is passed.
- *timeout* (int) - (optional) The timeout for the call in milliseconds. Default is infinite timeout.

Return Value:

None

void **RobotRaconteurNode.s.AsyncDisconnectService**(object *obj*, Action *handler*)

Asynchronous version of DisconnectService

Parameters:

- *obj* (object) - The client object to disconnect. Must have been connected with the ConnectService function.
- *handler* (Action) - Handler to function when disconnect is complete.

Return Value:

None

void **RobotRaconteurNode.s.Shutdown**()

Shuts down Robot Raconteur and closes all connections. Must be called before program exit.

Parameters:

None

Return Value:

None

void **RobotRaconteurNode.s.RegisterServiceType**(ServiceFactory *servicetype*)

Registers a ServiceFactory with the node. The service factories are generated with the “think” code. In general, it will be named *servicenameFactory*, or the name of the service with Factory appended to the name. In C# the factories must be registered for clients as well as services. The client will not automatically register the service types.

Parameters:

- *servicetype* (ServiceFactory) - The service factory to register.

Return Value:

None

ServiceFactory **RobotRaconteurNode.s.GetServiceType**(string *name*)

Returns the ServiceFactory named *name*.

Parameters:

- *name* (string) - The name of the service type to retrieve the factory.

Return Value:

(ServiceFactory) - The requested service type.

bool **RobotRaconteurNode.s.IsServiceTypeRegistered**(string *name*)

Returns true if service named *name* is registered

Parameters:

- *name* (string) - The service factory named to check

Return Value:

None

string[] **RobotRaconteurNode.s.GetRegisteredServiceTypes**()

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

Parameters:

None

Return Value:

(string[]) - A string array of the names of the registered service factories.

ServiceFactory **RobotRaconteurNode.s.GetPulledServiceType**(object *obj*, string *name*)

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

Parameters:

- *obj* (object) - The connected service object
- *name* (string) - The name of the service type to retrieve the factory.

Return Value:

(ServiceFactory) - The requested service type.

string[] **RobotRaconteurNode.s.GetPulledServiceTypes**(object *obj*)

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

Parameters:

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

Return Value:

(string[]) - A string array of the names of the registered service factories.

Dictionary<string,object> **RobotRaconteurNode.s.GetServiceAttributes**(object *obj*)

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

Parameters:

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

Return Value:

(Dictionary<string,object>) - The attributes of the remote service.

string **RobotRaconteurNode.sFindObjectType**(object *obj*, string *objref*)

Retrieves the name of the objref named *objref* in object *obj*

Parameters:

- *obj* (object) - The object to search in.
- *objref* (string) - The membername of the objref.

Return Value:

(string) - The fully qualified type of the object referenced by the objref

string **RobotRaconteurNode.sFindObjectType**(object *obj*, string *objref*, string *index*)

Retrieves the name of the objref named *objref* in object *obj* at index *index*

Parameters:

- *obj* (object) - The object to search in.

- *objref* (string) - The membername of the objref.
- *index* (string) - The index to search for.

Return Value:

(string) - The fully qualified type of the object referenced by the objref

object **RobotRaconteurNode.s.FindObjRefTyped**(object *obj*, string *objrefname*, string *objecttype*)

This function is used to request an “objref” with a specific object type. This is mainly used with the `varobject` type where it is not guaranteed what type will be returned. This function is for non-indexed objref.

Parameters:

- *obj* (object) - The object reference that contains the objref member to query. This object must have been created through `ConnectService` or returned by an objref.
- *objrefname* (string) - The member name of the objref to query.
- *objecttype* (string) - The fully qualified object type to return.

Return Value:

(object) - The objref object. This must be cast to the expected type.

object **RobotRaconteurNode.s.FindObjRefTyped**(object *obj*, string *objrefname*, string *index*, string *objecttype*)

This function is used to request an “objref” with a specific object type. This is mainly used with the `varobject` type where it is not guaranteed what type will be returned. This function is for non-indexed objref.

Parameters:

- *obj* (object) - The object reference that contains the objref member to query. This object must have been created through `ConnectService` or returned by an objref.
- *objrefname* (string) - The member name of the objref to query.
- *index* (string) - The index for this objref. If this is an `int32` indexed objref, use the base 10 string representation of the index, i.e. `int.ToString()`.
- *objecttype* (string) - The fully qualified object type to return.

Return Value:

(object) - The objref object. This must be cast to the expected type.

void **RobotRaconteurNode.s.AsyncFindObjectType**(object *obj*, string *objref*, Action<string,Exception> *handler*, int *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of FindObjectType

Parameters:

- *obj* (object) - The object to search in.
- *objref* (string) - The membername of the objref.
- *handler* (Action<string,Exception>) - The handler for the asynchronous operation.
- *timeout* (int) - (optional) The timeout in milliseconds. The default is infinite.

Return Value:

None

void **RobotRaconteurNode.s.AsyncFindObjectType**(object *obj*, string *objref*, string *index*, Action<string,Exception> *handler*, int *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of FindObjectType

Parameters:

- *obj* (object) - The object to search in.
- *objref* (string) - The membername of the objref.
- *index* (string) - The index for this objref..
- *handler* (Action<string,Exception>) - The handler for the asynchronous operation.
- *timeout* (int) - (optional) The timeout in milliseconds. The default is infinite.

Return Value:

None

void **RobotRaconteurNode.s.AsyncFindObjRefTyped**(object *obj*, string *objrefname*, string *objecttype*, Action<object,Exception> *handler*, int *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of AsyncFindObjRefTyped

Parameters:

- *obj* (object) - The object reference that contains the objref member to query. This object must have been created through `ConnectService` or returned by an objref.

- *objrefname* (string) - The member name of the objref to query.
- *objettype* (string) - The fully qualified object type to return.
- *handler* (Action<object,Exception>) - The handler for the asynchronous operation.
- *timeout* (int) - (optional) The timeout in milliseconds. The default is infinite.

Return Value:

None

void **RobotRaconteurNode.s.AsyncFindObjRefTyped**(object *obj*, string *objrefname*, string *index*, string *objecttype*, Action<object,Exception> *handler*, int *timeout*=RR.TIMEOUT_INFINITE)

Asynchronous version of AsyncFindObjRefTyped

Parameters:

- *obj* (object) - The object reference that contains the objref member to query. This object must have been created through `ConnectService` or returned by an objref.
- *objrefname* (string) - The member name of the objref to query.
- *index* (string) - The index for this objref.
- *objettype* (string) - The fully qualified object type to return.
- *handler* (Action<object,Exception>) - The handler for the asynchronous operation.
- *timeout* (int) - (optional) The timeout in milliseconds. The default is infinite.

Return Value:

None

ServerContext **RobotRaconteurNode.s.RegisterService**(string *name*, string *servicetype*, object *object*, SecurityPolicy *securitypolicy*=null)

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

Parameters:

- *name* (string) - The name of the service. This must be unique within the node.
- *servicetype* (string) - The name of the service definition. Note that this is different than the Python field. It is just the service definition name and not the full type of

the object.

- *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.

void **RobotRaconteurNode.s.CloseService**(string *name*)

Closes the service with name *name*.

Parameters:

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

Return Value:

None

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

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

Parameters:

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

Return Value:

None

void **RobotRaconteurNode.s.ReleaseObjectLock**(object *obj*)

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

Parameters:

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

Return Value:

None

void **RobotRaconteurNode.s.AsyncRequestObjectLock**(object *obj*, RobotRaconteurObjectLockFlags *flags*, Action<string,Exception> *handler*, int *timeout=RR_TIMEOUT.INFINITE*)

Asynchronous version of RequestObjectLock.

Parameters:

- *obj* (object) - The object to lock. This object must have been created through ConnectService or an objref.
- *flags* (RobotRaconteurObjectLockFlags) - The flags for the lock. Must be RobotRaconteurObjectLockFlags.USER_LOCK for a "User" lock, or RobotRaconteurObjectLockFlags.CLIENT_LOCK for a "Client" lock.
- *handler* (Action<string,Exception>) - The handler for the asynchronous operation. The string parameter can be ignored.
- *timeout* (int) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

void **RobotRaconteurNode.s.AsyncReleaseObjectLock**(object *obj*, Action<string,Exception> *handler*, int *timeout=RR_TIMEOUT.INFINITE*)

Asynchronous version of ReleaseObjectLock.

Parameters:

- *obj* (object) - The object to lock. This object must have been created through ConnectService or an objref.
- *handler* (Action<string,Exception>) - The handler for the asynchronous operation. The string parameter can be ignored.
- *timeout* (int) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

void **RobotRaconteurNode.s.MonitorEnter**(object *obj*, int *timeout=RR_TIMEOUT.INFINITE*)

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

Return Value:

None

void **RobotRaconteurNode.s.MonitorExit**(object *obj*)

Releases a monitor lock.

Parameters:

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

Return Value:

None

DateTime **RobotRaconteurNode.s.NowUTC**

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.

void **RobotRaconteurNode.s.Sleep**(int *duration*)

Sleeps for the specified duration in milliseconds.

Parameters:

- *duration* (int) - The duration to sleep in milliseconds.

Return Value:

None

AutoResetEvent **RobotRaconteurNode.s.CreateAutoResetEvent**()

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

Parameters:

None

Return Value:

(AutoResetEvent) - A new AutoResetEvent

Rate **RobotRaconteurNode.s.CreateRate**(double *frequency*)

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

Parameters:

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

Return Value:

(Rate) - A new rate with the specified frequency

Timer **RobotRaconteurNode.s.CreateTimer**(int *period*, Action<TimerEvent> *handler*, bool *oneshot* = false)

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

Parameters:

- *period* (int) - The period of the timer in milliseconds.
- *handler* (Action<TimerEvent>) - 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

void **RobotRaconteurNode.s.SetExceptionHandler**(Action<Exception> *handler*)

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

Parameters:

- *handler* (Action<Exception>) - A function with one parameter that receives the exceptions.

Return Value:

None

ServiceInfo2[] **RobotRaconteurNode.s.FindServiceByType**(string *servicetype*, string[] *transportschemes*)

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

Parameters:

- *servicetype* (string) - The fully qualified type of the root object to search for.
- *transportschemes* (string[]) - An array of the schemes to search for.

Return Value:

(ServiceInfo2[]) - An array of ServiceInfo2 structures with the detected services.

NodeInfo2[] **RobotRaconteurNode.s.FindNodeByName**(string *servicetype*, string[] *transportschemes*)

Finds a node using auto-discovery based on the NodeName

Parameters:

- *nodename* (string) - The NodeName to search for.
- *transportschemes* (string[]) - A list of the schemes to search for

Return Value:

(NodeInfo2[]) - An array of NodeInfo2 structures with the detected nodes.

NodeInfo2[] **RobotRaconteurNode.s.FindNodeByID**(NodeID *nodeid*, string[] *transportschemes*)

Finds a node using auto-discovery based on the NodeID

Parameters:

- *nodeid* (NodeID) - The NodeID to search for.
- *transportschemes* (string[]) - A list of the schemes to search for

Return Value:

(NodeInfo2[]) - An array of NodeInfo2 structures with the detected nodes.

void **RobotRaconteurNode.s.AsyncFindServiceByType**(string *servicetype*, string[] *transportschemes*, Action<ServiceInfo2[]> *handler*, int *timeout=5000*)

Asynchronous version of FindServiceByType.

Parameters:

- *servicetype* (string) - The fully qualified type of the root object to search for.
- *transportschemes* (string[]) - An array of the schemes to search for.

- *handler* (Action<ServiceInfo2[]>) - The handler for the asynchronous operation.
- *timeout* (int) - (optional) The timeout in milliseconds. Default is 5 seconds

Return Value:

None

void **RobotRaconteurNode.s.AsyncFindNodeByName**(string *nodename*, string[] *transportschemes*, Action<NodeInfo2[]> *handler*, int *timeout=5000*)

Asynchronous version of FindNodeByName.

Parameters:

- *nodename* (string) - The node to search for.
- *transportschemes* (string[]) - An array of the schemes to search for.
- *handler* (Action<NodeInfo2[]>) - The handler for the asynchronous operation.
- *timeout* (int) - (optional) The timeout in milliseconds. Default is 5 seconds.

Return Value:

None

void **RobotRaconteurNode.s.AsyncFindNodeByID**(NodeID *nodeid*, string[] *transportschemes*, Action<NodeInfo2[]> *handler*, int *timeout=5000*)

Asynchronous version of FindNodeByID.

Parameters:

- *nodeid* (NodeID) - The node to search for.
- *transportschemes* (string[]) - An array of the schemes to search for.
- *handler* (Action<NodeInfo2[]>) - The handler for the asynchronous operation.
- *timeout* (int) - (optional) The timeout in milliseconds. Default is 5 seconds.

Return Value:

None

void **RobotRaconteurNode.s.UpdateDetectedNodes**()

Updates the detected nodes. Must be called before GetDetectedNodes

Parameters:

None

Return Value:

None

void **RobotRaconteurNode.s.AsyncUpdateDetectedNodes**(Action handler)

Asynchronous version of UpdateDetectedNodes

Parameters:

- *handler* (Action) - Completion callback function

Return Value:

None

NodeID[] **RobotRaconteurNode.s.GetDetectedNodes**()

Returns an array of NodeID containing the detected nodes.

Parameters:

None

Return Value:

(NodeID[]) - The detected nodes

int **RobotRaconteurNode.s.EndpointInactivityTimeout**

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

int **RobotRaconteurNode.s.TransportInactivityTimeout**

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

int **RobotRaconteurNode.s.TransactionTimeout**

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

int **RobotRaconteurNode.s.MemoryMaxTransferSize**

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

int **RobotRaconteurNode.s.NodeDiscoveryMaxCacheCount**

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.

A.2 ClientServiceListenerDelegate

```
delegate void ClientServiceListenerDelegate(ServiceStub client, ClientServiceListenerEventType ev, object parameter)
```

Delegate for client event listener function.

Parameters:

- *client* (ServiceStub) - The stub of the root object of the client connection.
- *ev* (ClientServiceListenerEventType) - The event type that occurred.
- *parameter* (object) - A parameter for the event.

Return Value:

None

A.3 MultiDimArray

```
class MultiDimArray
```

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

```
int DimCount
```

The number of dimensions.

```
int[] Dims
```

The dimensions in column-major order.

```
Array Real
```

The real data in column-major order.

```
bool Complex
```

true if the array has complex data.

```
Array Imag
```


The imaginary data in column-major order.

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

Creates a new array with the provided data.

Parameters:

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

Return Value:

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

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

Reads data from the source array into buffer.

Parameters:

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

Return Value:

None

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

Writes data from buffer into the array.

Parameters:

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

Return Value:

None

A.4 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.

string **NodeName**

The name of the found node.

NodeID **NodeID**

The id of the found node.

string **Name**

The name of the service.

string **RootObjectType**

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

string[] **RootObjectImplements**

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

string[] **ConnectionURL**

A string array of URL that can be used to connect to the service.

Dictionary<string,object> **Attributes**

A Dictionary 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.

A.5 NodeInfo2

class **NodeInfo2**

NodeInfo2 contains the results of a search for a node using auto-detect by “NodeName or“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.

string **NodeName**

The name of the found node.

NodeID **NodeID**

The id of the found node.

string[] **ConnectionURL**

A string array of URL that can be used to connect to the service.

A.6 Pipe<T>

class **Pipe<T>**

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

string **MemberName**

Returns the member name of this pipe.

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

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

Parameters:

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

Return Value:

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

void **AsyncConnect**(int *index*, Action<Pipe<T>.PipeEndpoint,Exception> *handler*, int

timeout=RR_TIMEOUT_HANDLER)

Asynchronous version of `Connect`.

Parameters:

- *index* (`int`) - The index of the `PipeEndpoint` pair. This can be -1 to mean “any index”.
- *handler* (`Action<Pipe<T>.PipeEndpoint,Exception>`) - The handler for the asynchronous operation.
- *timeout* (`int`) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

`PipeConnectCallback` **`Action<Pipe<T>.PipeEndpoint>`**

Specifies the callback to call on the service when a client connects a `PipeEndpoint`.

A.7 `Pipe<T>.PipeEndpoint`

class **`Pipe<T>.PipeEndpoint`**

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

`uint` **`Endpoint`**

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

`int` **`Index`**

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

`int` **`Available`**

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

T **ReceivePacket()**

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

Parameters:

None

Return Value:

(T) - The next packet

T **PeekPacket()**

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

Parameters:

None

Return Value:

(T) - The next packet

uint **SendPacket**(T *packet*)

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

Parameters:

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

Return Value:

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

void **AsyncSendPacket**(T *packet*, Action<uint,Exception> *handler*)

Asynchronous version of `SendPacket`

Parameters:

- *packet* (T) - The packet to send
- *handler* (Action<uint,Exception>) - The handler for the asynchronous operation.

Return Value:

None

bool **RequestPacketAck**

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

void **Close()**

Closes the pipe endpoint connection pair.

Parameters:

None

Return Value:

None

void **AsyncClose**(Action<Exception> *handler*, int *timeout*=RR_TIMEOUT.INFINITE)

handler (Action<Exception>) - The handler for the asynchronous operation.

timeout (int) - (optional) The timeout in milliseconds. Default is infinite.

Parameters:

None

Return Value:

Action<PipeEndpoint> **PipeCloseCallback**

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

event Action<PipeEndpoint> **PacketReceivedEvent**

An event triggered when a packet is received by PipeEndpoint.

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

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

A.8 Callback<T>

```
class Callback<T>
```

The `Callback<T>` 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. The generic `T` is a delegate type matching the callback function and will typically be one of the `Action` or `Func` generic types found in the standard `System` namespace.

T Function

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

T GetClientFunction(*uint endpoint*)

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* (*uint*) - The endpoint identifying the client to execute the function on.

Return Value:

(*T*) - A delegate to the function that will be executed on the client.

A.9 Wire<T>

```
class Wire<T>
```

The `Wire<T>` class implements the “wire” member. The `Wire<T>` object is used to create `Wire<T>.WireConnection` objects which implement a connection between the client and the service. On the client side, the function `Connect` is used to connect the `Wire<T>.WireConnection` to the service. On the service side, a callback function `ConnectCallback` is called when clients connects.

string **MemberName**

Returns the member name of this wire.

`Wire<T>.WireConnection` **Connect**()

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

Parameters:

None

Return Value:

(Wire<T>.WireConnection) - The connected Wire<T>.WireConnection.

```
void AsyncConnect(Action<Wire<T>.WireConnection,Exception> handler, int timeout=RR.TIMEOUT
)
```

Asynchronous version of Connect.

Parameters:

- *handler* (Action<Wire<T>.WireConnection,Exception>) - The handler for the asynchronous operation.
- *timeout* (int) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

```
Action<Wire<T>, Wire<T>.WireConnection> WireConnectCallback
```

Specifies the callback to call on the service when a client connects a Wire<T>.WireConnection. Passes the parent wire and connection as parameters to the callback.

A.10 Wire<T>.WireConnection

```
class Wire<T>.WireConnection
```

The Wire<T>.WireConnection class represents one end of a wire connection which is formed by a pair of Wire<T>.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 Wire<T>WireConnection connections are created by the Wire<T> 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.

```
uint Endpoint
```

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.

T **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.

T **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.

bool **InValueValid**

Returns true if the `InValue` has been set, otherwise false.

bool **OutValueValid**

Returns true if the `OutValue` has been set, otherwise false.

TimeSpec **LastValueReceivedTime**

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.

TimeSpec **LastValueSentTime**

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

void **Close()**

Closes the wire connection pair.

Parameters:

None

Return Value:

None

void **AsyncClose**(Action<Exception> *handler*, int *timeout=RR.TIMEOUT.INFINITE*)

The asynchronous version of `Close`.

Parameters:

- *handler* (`Action<Exception>`) - The handler for the asynchronous operation.
- *timeout* (`int`) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

`Action<Wire<T>.WireConnection>` **WireCloseCallback**

A callback function called when the `Wire<T>.WireConnection` is closed. This is used to detect when it has been closed.

`event Action<Wire<T>.WireConnection, T, TimeSpec>` **WireValueChangedEvent**

An event triggered when `InValue` has changed. It will pass the wire connection, the new value, and the timestamp of the new value to the event handler.

A.11 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.

`long` **seconds**

A 64-bit integer representing the seconds.

`int` **nanoseconds**

A 32-bit integer representing the nanoseconds.

TimeSpec(`long` *seconds*, `int` *nanoseconds*)

Creates a new `TimeSpec`.

Parameters:

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

Return Value:

Creates a new `TimeSpec` for use with the `new` keyword

operator ==
operator !=
operator >
operator <
operator >=
operator <=
operator -
operator +

Standard operators for use with `TimeSpec`.

`void cleanup_nanosecs()`

Adjusts value so that nanoseconds is positive.

Parameters:

None

Return Value:

None

`static TimeSpec Now()`

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

Parameters:

None

Return Value:

(`TimeSpec`) - The current time.

A.12 `ArrayMemory<T>`

`class ArrayMemory<T>`

The `ArrayMemory<T >` 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. T is a numeric primitive scalar.

`ArrayMemory(T[] array)`

Creates a new `ArrayMemory<T>`.

Parameters:

- *array* (T[]) - The array data.

Return Value:

Creates a new `ArrayMemory` for use with the `new` keyword.

`ulong` **Length**

The number of elements in the array.

`void` **Read**(`ulong` *memorypos*, T[] *buffer*, `ulong` *bufferpos*, `ulong` *count*)

Reads data from the memory into `buffer`.

Parameters:

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

Return Value:

None

`void` **Write**(`ulong` *memorypos*, T[] *buffer*, `ulong` *bufferpos*, `ulong` *count*)

Writes data from `buffer` into the memory.

Parameters:

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

Return Value:

None

A.13 MultiDimArrayMemory<T>

class **MultiDimArrayMemory**<T>

The `MultiDimArrayMemory<T>` 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`. For the *memorypos*, *bufferpos*, and *count* parameters in the functions, a `ulong` array is used. These are all in column-major order. T is a numeric primitive scalar type.

MultiDimArrayMemory(`MultiDimArray array`)

Creates a new `MultiDimArrayMemory<T>`.

Parameters:

- *array* (`MultiDimArray`) - The array data.

Return Value:

Creates a new `MultiDimArrayMemory<T>` for use with the `new` keyword.

`ulong` **DimCount**

The number of dimensions in the array.

`ulong[]` **Dims**

The dimensions of the array in column-major order.

`bool` **Complex**

true if the array is complex, otherwise false.

`void` **Read**(`ulong[] memorypos`, `MultiDimArray buffer`, `ulong[] bufferpos`, `ulong[] count`)

Reads data from the memory into `buffer`.

Parameters:

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

Return Value:

None

```
void Write(ulong[] memorypos, MultiDimArray buffer, ulong[] bufferpos, ulong[] count)
```

Writes data from *buffer* into the memory.

Parameters:

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

Return Value:

None

A.14 ServerEndpoint

```
class ServerEndpoint
```

The `ServerEndpoint` represents a client connection on the service side. For the Python bindings, this endpoint is used to access the current endpoint number and the current authenticated user.

```
static uint GetCurrentEndpoint()
```

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

Parameters:

None

Return Value:

(uint) - The current endpoint number.

```
static AuthenticatedUser GetCurrentAuthenticatedUser()
```

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.

A.15 ServerContext

class **ServerContext**

The `ServerContext` manages the service. A few functions are exposed.

static string **GetCurrentServicePath()**

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:

(string) - The current service path.

static ServerContext **GetCurrentServerContext()**

Returns the current server context for the current service object.

Parameters:

None

Return Value:

(ServerContext) - The current service context.

void **ReleaseServicePath**(string *path*)

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) - The service path.

Return Value:

None

`void ReleaseServicePath(string path, uint[] endpoints)`

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) - The service path.
- *endpoints* (uint[]) - The endpoints to notify of the release.

Return Value:

None

`void AddServerServiceListener(ServerServiceListenerDelegate listener)`

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

Parameters:

- *listener* (ServerServiceListenerDelegate) - A function to call when a service event is generated.

Return Value:

None

`void SetServiceAttributes(Dictionary<string,object> attributes)`

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

Parameters:

- *attributes* (Dictionary<string,object>) - The attributes. Must match the type `varvalue{string}`.

Return Value:

None

A.16 ServerServiceListenerDelegate

`delegate void ServerServiceListenerDelegate(ServerContext context, ServerServiceListenerEventTy
ev, object parameter)`

Delegate for server event listener function.

Parameters:

- *client* (ServerContext) - The server context of the service.
- *ev* (ServerServiceListenerEventType) - The event type that occurred.
- *parameter* (object) - A parameter for the event.

Return Value:

None

A.17 AuthenticatedUser

class **AuthenticatedUser**

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

string **Username**

The username of the authenticated user.

string[] **Privileges**

The list of privileges for the user.

DateTime **LoginTime**

The login time of the user.

DateTime **LastAccessTime**

The time of last access by the user.

A.18 NodeID

class **NodeID**

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

NodeID(string *id*)

Creates a new `NodeID`.

Parameters:

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

Return Value:

(NodeID) - The new NodeID.

NodeID(byte[] *id*)

Creates a new NodeID.

Parameters:

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

Return Value:

(NodeID) - The new NodeID.

string **ToString**()

Returns the string representation of the NodeID.

Parameters:

None

Return Value:

(string) - The string representation.

byte[] **ToByteArray**()

Returns the byte[] representation of the NodeID.

Parameters:

None

Return Value:

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

operator ==

operator !=

Standard operators for use with NodeID.

A.19 PasswordFileUserAuthenticator

class **PasswordFileUserAuthenticator**

The `PasswordFileUserAuthenticator` implements a basic user authentication system based on a string containing the password information. It has the same functionality as the Python version. This class extends `UserAuthenticator`.

PasswordFileUserAuthenticator(string *data*)

Creates a new `PasswordFileUserAuthenticator`.

Parameters:

- *data* (string) - A string containing the user, password, and privileges information.

Return Value:

(`PasswordFileUserAuthenticator`) - The new `PasswordFileUserAuthenticator`.

A.20 ServiceSecurityPolicy

class **ServiceSecurityPolicy**

The `ServiceSecurityPolicy` class represents the security policy for the service. It has the same functionality as the Python version.

ServiceSecurityPolicy(`UserAuthenticator` *authenticator*, Dictionary<string, string> *policies*)

Creates a new `ServiceSecurityPolicy`.

Parameters:

- *authenticator* (`UserAuthenticator`) - The authenticator used to authenticate user. Will typically be `PasswordFileUserAuthenticator`.
- *policies* (Dictionary<string, string>) - The privileges for this security policy.

Return Value:

(`ServiceSecurityPolicy`) - The new `ServiceSecurityPolicy`.

A.21 RobotRaconteurException

class **RobotRaconteurException**

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

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

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

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

`MessageErrorType` **ErrorCode**

The error code of the error.

`string` **Error**

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

`string` **Message**

The message associated with the exception.

A.22 RobotRaconteurRemoteException

`class` **RobotRaconteurRemoteException**

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

`MessageErrorType` **ErrorCode**

The error code of the error.

`string` **Error**

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

`string` **Message**

The message associated with the exception.

A.23 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.

`static string` **GetCurrentTransportConnectionURL()**

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

A.24 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()

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

Parameters:

None

Return Value:

The new `LocalTransport`

void **StartServerAsNodeID**(NodeID *nodeid*)

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

```
void StartServerAsNodeName(string nodename)
```

Parameters:

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`

Return Value:

nodename (string) - The nodename to use

```
void StartClientAsNodeName(string nodename)
```

Parameters:

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`

Return Value:

nodename (string) - The nodename to use

A.25 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()

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

Parameters:

None

Return Value:

The new `TcpTransport`

`void StartServer(int port)`

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 (int)` - The port to listen on

Return Value:

None

`void StartServerUsingPortSharer()`

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

`bool IsPortSharerRunning`

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

`int GetListenPort()`

Returns the port that the transport is listening for connections on

Parameters:

None

Return Value:

(int) - The port the transport is listening on

`void EnableNodeDiscoveryListening()`

Starts listening for node discovery packets.

Parameters:

None

Return Value:

None

`void DisableNodeDiscoveryListening()`

Stops listening for node discovery packets.

Parameters:

None

Return Value:

None

`void EnableNodeAnnounce()`

Begins sending node discovery packets.

Parameters:

None

Return Value:

None

`void DisableNodeAnnounce()`

Stops sending node discovery packets.

Parameters:

None

Return Value:

None

`double` **DefaultReceiveTimeout**

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.

`double` **DefaultConnectTimeout**

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

`double` **DefaultHeartbeatPeriod**

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

`int` **MaxMessageSize**

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.

`int` **MaxConnectionCount**

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.

`void` **LoadTlsNodeCertificate()**

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

`bool` **IsTlsNodeCertificateLoaded**

True if the certificate for this node has been loaded.

bool RequireTls

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

bool IsTransportConnectionSecure(object a)

Returns true if the specified connection is secure.

Parameters:

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

Return Value:

(bool) - true if connection is secure.

bool IsPeerIdentityVerified(object a)

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

Parameters:

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

Return Value:

(bool) - True if peer identity has been verified

string GetSecurePeerIdentity(object a)

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

Parameters:

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

Return Value:

(string) - The NodeID as as string.

bool AcceptWebSockets

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

string[] GetWebSocketAllowedOrigins()

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:

(`string[]`) - Array containing the current allowed origins.

`void AddWebSocketAllowedOrigin(string origin)`

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

Parameters:

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

Return Value:

None

`void RemoveWebSocketOrigin(string origin)`

Removes an origin from the allowed origin list.

Parameters:

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

Return Value:

None

A.26 CloudTransport

class **CloudTransport**

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

void **StartAsClient()**

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

Parameters:

None

Return Value:

None

void **StartAsServer()**

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

Parameters:

None

Return Value:

None

bool **IsCloudClientRunning**

Returns if the cloud client is running and connected.

A.27 HardwareTransport

class **HardwareTransport**

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

A.28 PipeBroadcaster<T>

class **PipeBroadcaster**<T>

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

PipeBroadcaster(`Pipe`<T> *pipe*, int *backlog*==1)

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`<T>) - The pipe to use for broadcasting
- *backlog* (int) - (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:

(`PipeBroadcaster`) - The new instance.

void **SendPacket**(T *packet*)

Parameters:

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

Return Value:

packet (T) - The packet to send

void **AsyncSendPacket**(T *packet*, Action *handler*)

Parameters:

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

Return Value:

packet (T) - The packet to send

handler (Action) - 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.

A.29 WireBroadcaster<T>

class **WireBroadcaster**<T>

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

WireBroadcaster(`Wire`<T> *wire*)

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`<T>) - The wire to use for broadcasting

Return Value:

(`WireBroadcaster`) - The new instance.

OutValue **T**

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