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

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++ interface. 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.

Using Robot Raconteur with C++ is more difficult than other languages because of the extensive use of the Boost libraries. Knowledge of the STL libraries, the Boost libraries, and templates is required.

Boost is distributed with a CMake config file that will automatically create variables for Robot Raconteur includes. This can be utilized with a `FindPackage(RobotRaconteur)` command in CMake. A variable called `RobotRaconteur_DIR` is created, which can then be pointed to the downloaded directory to include Robot Raconteur. The C++ example package contains a file named "RobotRaconteurExamplesCommon.cmake". This file is an example of how to include Robot Raconteur and its dependencies in a project. "RobotRaconteurExamplesCommon.cmake" and "CMakeLists.txt" are listed in Appendix A for reference.

2 The Boost libraries

C++ has a very limited standard library that does not implement many of the design patterns used in modern software like Robot Raconteur. The Boost libraries provide significant functionality that would otherwise be difficult to implement in C++. The Boost libraries can be downloaded from http://boost.org . Version 0.8 of Robot Raconteur uses boost version 1.60. It is important to use the same Boost version unless you are an expert in C++ programming. The boost libraries must be built before use. See the Boost website for specific details; the following are general instructions for building Boost.

2.1 Building Boost

On Windows, Boost is built using the "Visual Studio Command Prompt" or "Windows SDK Command Prompt" that matches the version of Visual Studio being used. For Visual Studio 2010, use the following commands in the unzipped boost directory:

```
bootstrap
```

```
b2 --with-date_time --with-thread --with-system --with-regex
--with-filesystem --with-chrono --with-atomic --stagedir=stage32 --toolset=msvc-10.0
```

If 64-bit binaries are required, also run the following:

```
b2 --with-date_time --with-thread --with-system --with-regex --with-filesystem
--with-chrono --with-atomic --stagedir=stage64 --toolset=msvc-10.0 address-model=64
```

For Visual Studio 2012, run:

```
bootstrap
```

```
b2 --with-date_time --with-thread --with-system --with-regex --with-filesystem
--with-chrono --with-atomic --stagedir=stage32 --toolset=msvc-11.0
```

If 64-bit binaries are required, also run the following:

```
b2 --with-date_time --with-thread --with-system --with-regex --with-filesystem
--with-chrono --with-atomic --stagedir=stage64 --toolset=msvc-11.0 address-model=64
```

For Visual Studio 2013, run:

```
bootstrap
```

```
b2 --with-date_time --with-thread --with-system --with-regex --with-filesystem
```

```
--with-chrono --with-atomic --stagedir=stage32 --toolset=msvc-12.0
```

If 64-bit binaries are required, also run the following:

```
b2 --with-date_time --with-thread --with-system --with-regex --with-filesystem
--with-chrono --with-atomic --stagedir=stage64 --toolset=msvc-12.0 address-model=64
```

For Visual Studio 2015, run:

```
bootstrap
```

```
b2 --with-date_time --with-thread --with-system --with-regex --with-filesystem
--with-chrono --with-atomic --stagedir=stage32 --toolset=msvc-14.0
```

If 64-bit binaries are required, also run the following:

```
b2 --with-date_time --with-thread --with-system --with-regex --with-filesystem
--with-chrono --with-atomic --stagedir=stage64 --toolset=msvc-14.0 address-model=64
```

The 32-bit libraries will be created in the `stage32` directory and the 64-bit libraries will be created in the `stage64` directory.

On Linux, it is necessary to install the required dependencies for Boost and also gcc version 4.6 which is currently used with Robot Raconteur. Check the Boost documentation for build instructions for the distribution in use. Run the following commands in the untarred Boost directories.

```
./bootstrap
```

```
./b2 --with-date_time --with-thread --with-system --with-regex --with-filesystem
--with-chrono --with-atomic --layout=versioned variant=debug,release cflags=-fPIC
cxxflags=-fPIC linkflags=-fPIC
```

On Mac OSX, there are two major configurations support: the Clang toolchain, and the MacPorts toolchain. For the Clang toolchain, use the following commands to build fat binaries with both 32-bit and 64-bit code.

```
./bootstrap
```

```
./b2 toolset=darwin cxxflags="-arch i386 -arch x86_64" cxxflags=-stdlib=libc++
linkflags=-stdlib=libc++ --with-date_time --with-thread --with-system --with-regex
--with-filesystem --with-chrono --with-atomic variant=debug,release
--layout=versioned
```

2.2 Boost Libraries used by Robot Raconteur

The previous section builds the necessary Boost libraries for Robot Raconteur. The following classes are of particular interest by software using Robot Raconteur:

- `boost::shared_ptr<T>`
- `boost::make_shared<T>`
- `boost::enable_shared_from_this<T>`
- `boost::function<...>`
- `boost::bind<...>`
- `boost::signals2::signal<...>`
- `boost::thread`
- `boost::recursive_mutex`
- `boost::posix_time::ptime`

The Boost website contains documentation and examples for each of these classes and the reader is directed to review these before attempting to read the rest of this document.

Note: The type `RR_SHARED_PTR` used throughout the code is an alias for `boost::shared_ptr`.

2.3 CMake BoostFind

The CMake `BoostFind` command is used to find the Boost libraries. Using `BoostFind` requires setting several advanced CMake variables. After an initial configuration, click on "Advanced" and search for the following variables:

Boost_INCLUDE_DIR
Boost_LIBRARY_DIR_DEBUG
Boost_LIBRARY_DIR_RELEASE

If you used the instructions in this section to build the Boost libraries, **Boost_INCLUDE_DIR** should be set to the directory that contains "b2", and the two library directories should be set to the stage directory. This will be "stage", "stage32", or "stage64" depending on the configuration.

Sometimes `FindBoost` will not be able to locate the libraries due to the way that the version layout was generated. In this case, specifying `Boost_COMPILER` may also be necessary. See the documentation for `FindBoost` for more details.

3 OpenSSL

OpenSSL version 1.0 is required to use Robot Raconteur on Linux, Mac OSX, Android, and iOS. (Windows uses SChannel and does not require OpenSSL.) On most Linux distributions, the standard version is sufficient and can be used. On Debian based system, run the following to install:

```
sudo apt-get install libssl-dev
```

On OSX, things are more complicated. The built in version is very old and not compatible. To get an up-to-date version, install homebrew and run:

```
brew install openssl
```

OpenSSL is now available in the "Cellar" of brew. An example location is `/usr/local/Cellar/openssl/1.0.1e`. (The tweak version number may change.) Create a PATH variable named `OPENSSL_ROOT_DIR` in CMake by clicking on the "Add Entry" button in the CMake window and point it to the OpenSSL brew directory. Unfortunately CMake has a tendency to find the built in OpenSSL libraries that will not work, and Robot Raconteur will throw an error. Enable "Advanced" variables and delete the `OPENSSL_SSL_LIBRARY` and `OPENSSL_CRYPTO_LIBRARY` entries. Configure again and it should find the correct libraries.

4 "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++ does not have 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=cpp 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 three files, "Create_interface.h", "Create_interface_stubskel.h", and "Create_interface_stubskel.cpp". 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 namespace Create_interface;
```

to import the generated classes and interfaces. It is necessary to include the two generated header files for each service type in files that use them.

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".

5 Building with Robot Raconteur

Building with Robot Raconteur requires including the Robot Raconteur library, the libraries that Robot Raconteur depends on, and the appropriate header files. In general, the following lines are required:

```
#include<RobotRaconteur.h> using namespace RobotRaconteur;
```

Note that in "Windows.h", `SendMessage` is defined as as a different name and can cause problems in Robot Raconteur. Use the following line before including Robot Raconteur:

```
#undef SendMessage
```

The Robot Raconteur library is distributed as a static library.

Robot Raconteur provides examples and support files for CMake. While CMake is not required, it is highly recommended as it does most of the configuration automatically. Appendix A and the example projects for examples on how to use CMake. More detailed CMake instructions can be found at `http://cmake.org`.

6 C++ $\leftrightarrow$ Robot Raconteur Data Type Mapping

Each valid Robot Raconteur type has a corresponding C++ data type. This mapping is similar to Python. Table 1 shows the mapping between Robot Raconteur and C++ data types.

Many of the data types utilize `boost::shared_ptr` for memory management. This provides automatic reference counting so the data is destroyed when all references are released. Note that the pointer stored in `boost::shared_ptr` may be 0, corresponding to a "null reference" in other languages. Unlike the other languages supported by Robot Raconteur, trying to access a null pointer can crash the entire program. Make sure the pointer passed is valid by using `if (mysmrtptr) { }` blocks where `mysmrtptr` is a smart pointer to be checked if it is valid.

The class `RRArray` is used to store array values. It is a lightweight class that is used to include the length of the array and the type for data marshaling.

Table 1: Robot Raconteur $\leftrightarrow$ C++ Type Map

Robot Raconteur Type	C++ Type	Notes
double	`double`	
single	`float`	
int8	`int8_t`	
uint8	`uint8_t`	
int16	`int16_t`	
uint16	`uint16_t`	
int32	`int32_t`	
uint32	`uint32_t`	
int64	`int64_t`	
uint64	`uint64_t`	
double[]	`boost::shared_ptr<RRArray<double> >`	
single[]	`boost::shared_ptr<RRArray<float> >`	
int8[]	`boost::shared_ptr<RRArray<int8_t> >`	
uint8[]	`boost::shared_ptr<RRArray<uint8_t> >`	
int16[]	`boost::shared_ptr<RRArray<int16_t> >`	
uint16[]	`boost::shared_ptr<RRArray<uint16_t> >`	
int32[]	`boost::shared_ptr<RRArray<int32_t> >`	
uint32[]	`boost::shared_ptr<RRArray<uint32_t> >`	
int64[]	`boost::shared_ptr<RRArray<int64_t> >`	
uint64[]	`boost::shared_ptr<RRArray<uint64_t> >`	
string	`std::string` or `boost::shared_ptr<RRArray<char> >`	Strings are always UTF-8 encoded.
T{int32}	`boost::shared_ptr<RRMap<int32_t,T> >`	Map type, *T* is a template
T{string}	`boost::shared_ptr<RRMap<std::string,T> >`	Map type, *T* is a template
T{list}	`boost::shared_ptr<RRList<T> >`	List type, *T* is a template
structure	*varies*	Use generated class for corresponding structure.
N[*]	`boost::shared_ptr<RRMultiDimArray<N> >`	Multi-dim array of type *N*
varvalue	`boost::shared_ptr<RRObject>`	
varobject	`boost::shared_ptr<RRObject>`	

Strings are either `std::string` or `RRArray<char>` depending on the situation. Strings are always UTF-8 encoded. Helper functions are available to convert between the `RRArray<string>` form and `std::string` form.

The maps are stored using the type `boost::shared_ptr<RRMap<K,T> >` where `K` is either `int32_t` or `std::string`. See Section **??** for more details.

The `RRMultiDimArray<T>` 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 B.14 for more details on this class.

7 Service Objects

The service objects in Robot Raconteur are defined as abstract classes 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.

property
: Properties are implemented using getter/setter functions. The getter is The getters are pre-pended with "get_" and the setter is pre-pended with "set_". They are always declared `public virtual`.

function
: Functions are implemented as standard C++ functions. They are always declared `public virtual`.

event
: Events are implemented using `boost::signals2::signals`. A "get_" function returns a reference to the signals object.

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 except using "get_" and "set_" functions. 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 `PipeEndpoint<T>`.

callback

Callbacks are implemented using properties in the same manner as Python except using "get_" and "set_" functions. On the client side, the `Function` field in the `Callback<T>` object is set to the desired function. The type `T` is specified using `boost::function` that can be created using `boost::bind`. On the service side, the function `GetClientFunction(uint e)` function is used to retrieve a `boost::function` 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 except using "get_" and "set_" functions. 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 `WireConnection<T>`.

memory

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

C++ exceptions extending `std::exception` are transparently returned across the client/service boundary and are thrown on the calling side as in the other Robot Raconteur languages.

8 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 `boost::function<void(RR_SHARED_PTR<RobotRaconteurException>)>` if the original function returns `void` or `boost::function<void(T,RR_SHARED_PTR<RobotRaconteurException>)>` 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.

9 Conclusion

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

A CMake Examples

A.1 RobotRaconteurExamplesCommon.cmake

```
#Some versions of CMAKE don't search for Boost 1.60
SET (Boost_ADDITIONAL_VERSIONS 1.60.0 1.60)
SET (Boost_USE_STATIC_LIBS ON)
SET (Boost_USE_MULTITHREADED ON)
SET (Boost_USE_STATIC_RUNTIME OFF)
find_package(Boost COMPONENTS date_time filesystem system regex chrono atomic thread REQUIRED)
add_definitions(-DBOOST_ALL_NO_LIB)

if (CMAKE_COMPILER_IS_GNUCXX)
        find_package(OpenSSL)
        set (RobotRaconteur_EXTRA_LIBRARIES ${RobotRaconteur_EXTRA_LIBRARIES} ${OPENSSL_LIBRARIES
            } pthread rt z)
endif()

if (CMAKE_GENERATOR STREQUAL Xcode)

        set(OPENSSL_USE_STATIC_LIBS TRUE)
        find_package(OpenSSL)

        include(CMakeFindFrameworks)
        set(CMAKE_XCODE_ATTRIBUTE_GCC_VERSION "com.apple.compilers.llvm.clang.1_0")
        set(CMAKE_XCODE_ATTRIBUTE_CLANG_CXX_LIBRARY "libc++")
        CMAKE_FIND_FRAMEWORKS(CoreFoundation)
        CMAKE_FIND_FRAMEWORKS(Security)
        include_directories(${CoreFoundation_FRAMEWORKS}/Headers ${Security_FRAMEWORKS}/Headers)
endif()

include_directories(${Boost_INCLUDE_DIRS})

find_package(RobotRaconteur REQUIRED)
```

A.2 CMakeLists.txt for iRobotCreateClient

```
cmake_minimum_required(VERSION 2.8.12)
project(iRobotCreateClient)

include(../RobotRaconteurExamplesCommon.cmake)

add_executable(iRobotCreateClient
        experimental__create_stubskel.cpp
        iRobotCreateClient.cpp)

target_link_libraries(iRobotCreateClient ${RobotRaconteur_LIBRARY} ${Boost_LIBRARIES} ${
    RobotRaconteur_EXTRA_LIBRARIES} )

add_executable(FindiRobotCreateServiceNode
        experimental__create_stubskel.cpp
        FindiRobotCreateServiceNode.cpp)

target_link_libraries(FindiRobotCreateServiceNode ${RobotRaconteur_LIBRARY} ${Boost_LIBRARIES} ${
    RobotRaconteur_EXTRA_LIBRARIES} )
```

B Robot Raconteur Reference

B.1 RR_SHARED_PTR

`#define` **RR_SHARED_PTR** `boost::shared_ptr`

`RR_SHARED_PTR` is an alias to `boost::shared_ptr`.

B.2 rr_cast<T, U>

`template<typename T, typename U>`
`RR_SHARED_PTR<T>` **rr_cast**(`RR_SHARED_PTR<U>` *in*)

Casts the input *in* to the desired type `T`. The type `U` does not need to be specified and will be determined implicitly. This is the same as `dynamic_pointer_cast` but throws an exception `DataTypeMismatchException` if the cast cannot be completed.

Parameters:

- *in* (`RR_SHARED_PTR<U>`) - he object to be cast.

Return Value:

(`RR_SHARED_PTR<T>`) - The object cast to type `RR_SHARED_PTR<T>`

B.3 RRObject

`class` **RRObject**

`RRObject` is the base class for most classes in Robot Raconteur. Its subclasses are almost always contained by a `boost::shared_ptr`. It is used to help determine the type of the object at runtime since C++ has limited runtime reflection capabilities.

`std::string` **RRType**()

Returns a string representing the type of the object. This is used internally by Robot Raconteur.

Parameters:

None

Return Value:

(`std::string`) - A string representing the type of the object.

B.4 RRArray<T>

`template<typename T>`
`class` **RRArray**

The class `RRArray` is used to represent array types that are passed to and returned from Robot Raconteur. It is a lightweight wrapper to a pointer that is primarily intended to help the Robot Raconteur library determine the type of data and the number of elements. The `RRArray` may *own* the data, meaning that the data will be deleted when the `RRArray` is deleted. This class is always contained within a `boost::shared_ptr` for memory management. It is created through the functions `AllocateRRArray`, `AttachRRArray`, or `AttachRRArrayCopy`.

`T*` **ptr**()

Returns a pointer to the first number in the array.

Parameters:

None

Return Value:

(`T*`) - The pointer

`size_t` **Length**()

The number of elements in the array.

Parameters:

None

Return Value:

(`size_t`) - The number of elements in the array.

`size_t` **size**()

The number of elements in the array.

Parameters:

None

Return Value:

(`size_t`) - The number of elements in the array.

`size_t` **ElementSize**()

The number of bytes required to store one element.

Parameters:

None

Return Value:

(`size_t`) - The size of an element in bytes.

`T&` **operator[]**(`size_t` *pos*)

Returns a reference to the selected element in the array or throws the exception `std::out_of_range` if the position is outside the array.

Parameters:

- *pos* (`size_t`) - The index in the array of the element

Return Value:

(`T&`) - Reference to the selected element.

B.5 AllocateRRArray<T>

```
template<typename T>
RR_SHARED_PTR<RRArray<T> > AllocateRRArray(size_t count)
```

Allocates a new RRArray with element type `T` and number of elements *count*. The array data will be deleted when the RRArray is deleted.

Parameters:

- *count* (`size_t`) - The number of elements.

Return Value:

(`RR_SHARED_PTR<RRArray<T> >`) - The new array.

B.6 AttachRRArray<T>

```
template<typename T>
RR_SHARED_PTR<RRArray<T> > AttachRRArray(T* data, size_t count, bool owned)
```

Attaches an `RRArray<T>` to the supplied existing raw pointer array. If the `owned` parameter is true, the supplied pointer will be deleted when the RRArray is deleted.

Parameters:

- *data* (`T*`) - Pointer to the first element in the array to be attached to.
- *count* (`size_t`) - The number of elements in the supplied array.
- *owned* (`bool`) - If true, the data is owned by the RRArray and will be deleted when the RRArray is deleted.

Return Value:

(`RR_SHARED_PTR<RRArray<T> >`) - The RRArray with the supplied data.

B.7 AttachRRArrayCopy<T>

```
template<typename T>
RR_SHARED_PTR<RRArray<T> > AttachRRArray(T* data, size_t count)
```

Same as `AttachRRArray` but creates a deep copy of the data.

Parameters:

- *data* (`T*`) - Pointer to the first element in the array to be attached to.
- *count* (`size_t`) - The number of elements in the supplied array.

Return Value:

(`RR_SHARED_PTR<RRArray<T> >`) - The RRArray containing the copied supplied data.

B.8 ScalarToRRArray<T>

```
template<typename T>
RR_SHARED_PTR<RRArray<T> > ScalarToRRArray(T value)
```

Creates an RRArray with a single element with the supplied value.

Parameters:

- *value* (T) - The scalar

Return Value:

(RR_SHARED_PTR<RRArray<T> >) - An RRArray with one element containing *value*

B.9 RRArrayToScalar<T>

```
template<typename T>
T RRArrayToScalar(RR_SHARED_PTR<RRArray<T> > value)
```

Returns the first element in the RRArray or throws an exception if the array is length zero.

Parameters:

- *data* (RR_SHARED_PTR<RRArray<T> >) - The array containing the data.

Return Value:

(T) - The first element in the array.

B.10 stringToRRArray

```
RR_SHARED_PTR<RRArray<char> > stringToRRArray(std::string str)
```

Copies a std::string to a RR_SHARED_PTR<RRArray<char> >.

Parameters:

- *str* (std::string) - String data to copy.

Return Value:

(RR_SHARED_PTR<RRArray<char> >) - The RRArray with the copied supplied string.

B.11 RRArrayToString

```
std::string RRArrayToString(RR_SHARED_PTR<RRArray<char> > arr)
```

Copies a RR_SHARED_PTR<RRArray<char> > to a std::string.

Parameters:

- *arr* (RR_SHARED_PTR<RRArray<char> >) - Pointer to the first element in the array to be attached to.

Return Value:

(std::string) - The std::string with the copied supplied string.

B.12 RRMap<K,T>

template<typename K, typename T>
class **RRMap**

The RRMap class is a thin wrapper around a std::map that contains extra data to help Robot Raconteur determine the type of data contained. The actual map is name map and is a field in the class.

Note that all data that stored in this map is always stored in a boost::shared_ptr. This means that scalars and std::string must be converted to type boost::shared_ptr<RRArray<T> > before being stored in the map. The helper function ScalarToRRArray, RRArrayToScalar, stringToRRArray, and RRArrayToString for these conversions.

std::map<K,RR_SHARED_PTR<T> > **map**

The std::map that contains the data.

RRMap()

Creates a new RRMap with empty data. This should always be used with RR_MAKE_SHARED.

Parameters:

None

Return Value:

(RRMap) - The new map.

RRMap(std::map<K,RR_SHARED_PTR<T> > *mapin*)

Creates a new RRMap with empty data. Creates a new RRMap copying the supplied *mapin* data. This should always be used with RR_MAKE_SHARED.

Parameters:

- *mapin* (std::map<K,RR_SHARED_PTR<T> >) - The initial data. It will be copied into map.

Return Value:

(RRMap) - The new map.

B.13 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()` is the "singleton" of the node. All functions must use this property to access the node.

`std::string` **RobotRaconteurNode::s()->GetRobotRaconteurVersion**()

Returns the version of the Robot Raconteur library.

`NodeID` **RobotRaconteurNode::s()->NodeID**()
`void` **RobotRaconteurNode::s()->SetNodeID**(`const NodeID&` *value*)

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.

`std::string` **RobotRaconteurNode::s()->NodeName**()
`void` **RobotRaconteurNode::s()->SetNodeName**(`const std::string&` *value*)

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.

`void` **RobotRaconteurNode::s()->RegisterTransport**(`RR_SHARED_PTR<Transport>` *transport*)

Registers a transport with the node.

Parameters:

- *transport* (`RR_SHARED_PTR<Transport>`) - The transport to be registered

Return Value:

None

`RR_SHARED_PTR<RRObject>` **RobotRaconteurNode::s()->ConnectService**(`const std::string&` *url*, `const std::string&` *username=""*, `RR_SHARED_PTR<RRMap<std::string,RRObject> >` *credentials = RR_SHARED_PTR<RRMap<std::string,RRObject>>()*,
`boost::function<void (RR_SHARED_PTR<ClientContext>, ClientServiceListenerEventType,RR_SHARED_PTR<void>)>` *servicelistener=NULL*,

const std::string& *objecttype=""*)

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* (const std::string&) - The URL to connect to.
- *username* (const std::string&) - (optional) The username to use with authentication.
- *credentials* (RR_SHARED_PTR<RRMap<std::string,RRObject> >) - (optional) The credentials to use with authentication.
- *servicelistener* (boost::function<void (RR_SHARED_PTR<ClientContext>, ClientServiceListenerEventType,RR_SHARED_PTR<void>)>) - (optional) A function to call when a client event is generated such as disconnect.
- *objecttype* (const std::string&) - (optional) The fully qualified type of the root service object.

Return Value:

(RR_SHARED_PTR<RRObject>) - 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.

RR_SHARED_PTR<RRObject> **RobotRaconteurNode::s()->ConnectService**(const std::vector<std::string>& *url*, const std::string& *username=""*, RR_SHARED_PTR<RRMap<std::string,RRObject> > *credentials = RR_SHARED_PTR<RRMap<std::string,RRObject>>()*, boost::function<void (RR_SHARED_PTR<ClientContext>, ClientServiceListenerEventType,RR_SHARED_PTR<void>)> *servicelistener=NULL*, const std::string& *objecttype=""*)

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* (const std::vector<std::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* (`const std::string&`) - (optional) The username to use with authentication.
- *credentials* (`RR_SHARED_PTR<RRMap<std::string,RRObject> >`) - (optional) The credentials to use with authentication.
- *servicelistener* (`boost::function<void (RR_SHARED_PTR<ClientContext>, ClientServiceListenerEventType,RR_SHARED_PTR<void>)>`) - (optional) A function to call when a client event is generated such as disconnect.
- *objecttype* (`const std::string&`) - (optional) The fully qualified type of the root service object.

Return Value:

(`RR_SHARED_PTR<RRObject>`) - 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**(`RR_SHARED_PTR<RRObject>` *obj*)

Disconnects a service.

Parameters:

- *obj* (`RR_SHARED_PTR<RRObject>`) - The client object to disconnect. Must have been connected with the `connectService` function.

Return Value:

None

`void` **RobotRaconteurNode::s()->AsyncConnectService**(
`const std::string&` *url*,
`const std::string&` *username*,
`RR_SHARED_PTR<RRMap<std::string,RRObject> >` *credentials*,
`boost::function<void (RR_SHARED_PTR<ClientContext>, ClientServiceListenerEventType,RR_SHARED_PTR<void>)>` *servicelistener*,
`const std::string&` *objecttype*,
`boost::function<void(RR_SHARED_PTR<RRObject>, RR_SHARED_PTR<RobotRaconteurException>)>` *handler*,
`int32_t` *timeout=RR_TIMEOUT_INFINITE*)

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 `RobotRaconteurException` will not be null. Otherwise it will be null and the object will be passed to the handler.

Parameters:

- *url* (`const std::string&`) - The URL to connect to.
- *username* (`const std::string&`) - The username to use with authentication.
- *credentials* (`RR_SHARED_PTR<RRMap<std::string,RRObject> >`) - The credentials to use with authentication.
- *servicelistener* (`boost::function<void (RR_SHARED_PTR<ClientContext>, ClientServiceListenerEventType,RR_SHARED_PTR<void>)>`) - A function to call when a client event is generated such as disconnect.
- *objecttype* (`const std::string&`) - The fully qualified type of the root service object.
- *handler* (`boost::function<void(RR_SHARED_PTR<RRObject>, RR_SHARED_PTR<RobotRaconteurException>)>`) - The handler function. The connected object is passed as the first parameter. The second parameter will be NULL on success, otherwise an `RobotRaconteurException` instance is passed.
- *timeout* (`int32_t`) - (optional) The timeout for the call in milliseconds. Default is infinite timeout.

Return Value:

None

`void` **RobotRaconteurNode::s()->AsyncConnectService**(
`const std::vector<std::string>&` *url*, `const std::string&` *username*,
`RR_SHARED_PTR<RRMap<std::string,RRObject> >` *credentials*,
`boost::function<void (RR_SHARED_PTR<ClientContext>, ClientServiceListenerEventType,RR_SHARED_PTR<void>)>` *servicelistener*,
`const std::string&` *objecttype*,
`boost::function<void(RR_SHARED_PTR<RRObject>, RR_SHARED_PTR<RobotRaconteurException>)>` *handler*,
`int32_t` *timeout=RR_TIMEOUT_INFINITE*)

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 `RobotRaconteurException` will not be null. Otherwise it will be null and the object will be passed to the handler.

Parameters:

- *url* (`const std::vector<std::string>&`) - A list of trial URLs to connect to.
- *username* (`const std::vector<std::string>&`) - The username to use with authentication.
- *credentials* (`RR_SHARED_PTR<RRMap<std::string,RRObject> >`) - The credentials

to use with authentication.

- *servicelistener* (`boost::function<void (RR_SHARED_PTR<ClientContext>, ClientServiceListenerEventType,RR_SHARED_PTR<void>)>`) - A function to call when a client event is generated such as disconnect.
- *objecttype* (`const std::string&`) - The fully qualified type of the root service object.
- *handler* (`boost::function<void(RR_SHARED_PTR<RRObject>, RR_SHARED_PTR<RobotRaconteurException>)>`) - The handler function. The connected object is passed as the first parameter. The second parameter will be NULL on success, otherwise an `RobotRaconteurException` instance is passed.
- *timeout* (`int32_t`) - (optional) The timeout for the call in milliseconds. Default is infinite timeout.

Return Value:

None

`void` **RobotRaconteurNode::s()->AsyncDisconnectService**(`RR_SHARED_PTR<RRObject>` *obj*, `boost::function<void()>` *handler*)

Asynchronous version of `DisconnectService`

Parameters:

- *obj* (`RR_SHARED_PTR<RRObject>`) - The client object to disconnect. Must have been connected with the `connectService` function.
- *handler* (`boost::function<void()>`) - 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**(RR_SHARED_PTR<ServiceFactory> *servicetype*)

Registers a ServiceFactory with the node. The service factories are generated with the "thunk" code. In general, it will be named *servicename*Factory, 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* (RR_SHARED_PTR<ServiceFactory>) - The service factory to register.

Return Value:

None

RR_SHARED_PTR<ServiceFactory> **RobotRaconteurNode::s()->GetServiceType**(const std::string& *name*)

Returns the ServiceFactory named *name*.

Parameters:

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

Return Value:

(RR_SHARED_PTR<ServiceFactory>) - The requested service type.

bool **RobotRaconteurNode::s()->IsServiceTypeRegistered**(const std::string& *name*)

Returns true if service named *name* is registered

Parameters:

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

Return Value:

None

std::vector<std::string> **RobotRaconteurNode::s()->GetRegisteredServiceTypes**()

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

Parameters:

None

Return Value:

(std::vector<std::string>) - A string array of the names of the registered service factories.

RR_SHARED_PTR<ServiceFactory> **RobotRaconteurNode::s()->GetPulledServiceType**(RR_SHARED_PTR<RRObject> *obj*, const std::string& *name*)

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

Parameters:

- *obj* (Object) - The connected service object
- *name* (const std::string&) - The name of the service type to retrieve the factory.

Return Value:

(RR_SHARED_PTR<ServiceFactory>) - The requested service type.

std::vector<std::string> **RobotRaconteurNode::s()->GetPulledServiceTypes**(RR_SHARED_PTR<RRObject> *obj*)

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

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The connected service object

Return Value:

(std::vector<std::string>) - A string array of the names of the registered service factories.

std::map<std::string,RR_SHARED_PTR<RRObject>
RobotRaconteurNode::s()->GetServiceAttributes(RR_SHARED_PTR<RRObject> *obj*)

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

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The connected service object

Return Value:

(std::map<std::string,RR_SHARED_PTR<Object> >) - The attributes of the remote service.

std::string **RobotRaconteurNode::s()->FindObjectType**(RR_SHARED_PTR<RRObject> *obj* const std::string& *objref*)

Retrieves the name of the objref named objref in object obj

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The object to search in.
- *objref* (const std::string&) - The membername of the objref.

Return Value:

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

std::string **RobotRaconteurNode::s()->FindObjectType**(RR_SHARED_PTR<RRObject> *obj*, const std::string& *objref*, const std::string& *index*)

Retrieves the name of the objref named objref in object obj

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The object to search in.
- *objref* (const std::string&) - The membername of the objref.
- *index* (const std::string&) - The index to search for.

Return Value:

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

RR_SHARED_PTR<RRObject> **RobotRaconteurNode::s()->FindObjRefTyped**(
RR_SHARED_PTR<RRObject> *obj*, const std::string& *objrefname*,
const std::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* (RR_SHARED_PTR<RRObject>) - The object reference that contains the objref member to query. This object must have been created through ConnectService or returned by an objref.
- *objrefname* (const std::string&) - The member name of the objref to query.
- *objettype* (const std::string&) - The fully qualified object type to return.

Return Value:

(RR_SHARED_PTR<RRObject>) - The objref object. This must be cast to the expected type.

RR_SHARED_PTR<RRObject> **RobotRaconteurNode::s()->FindObjRefTyped**(
RR_SHARED_PTR<Object> *obj*, const std::string& *objrefname*,

const std::string& *index*, const std::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* (RR_SHARED_PTR<RRObject>) - The object reference that contains the objref member to query. This object must have been created through ConnectService or returned by an objref.
- *objrefname* (const std::string&) - The member name of the objref to query.
- *index* (const std::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* (const std::string&) - The fully qualified object type to return.

Return Value:

(RR_SHARED_PTR<RRObject>) - The objref object. This must be cast to the expected type.

RobotRaconteurNode::s()->AsyncFindObjectType(RR_SHARED_PTR<RRObject> *obj* const std::string& *objref*, boost::function<void (RR_SHARED_PTR<std::string>, RR_SHARED_PTR<RobotRaconteurException>)> *handler*, int32_t *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of FindObjectType

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The object to search in.
- *objref* (const std::string&) - The membername of the objref.
- *handler* (boost::function<void (RR_SHARED_PTR<std::string>, RR_SHARED_PTR<RobotRaconteurException>)>) - The handler for the asynchronous operation.
- *timeout* (int32_t) - The timeout in milliseconds. The default is infinite.

Return Value:

None

RobotRaconteurNode::s()->AsyncFindObjectType(RR_SHARED_PTR<RRObject> *obj* const std::string& *objref*, const std::string& *index*, boost::function<void (RR_SHARED_PTR<std::string>, RR_SHARED_PTR<RobotRaconteurException>)> *handler*,

int32_t *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of FindObjectType

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The object to search in.
- *objref* (const std::string&) - The membername of the objref.
- *index* (const std::string&) - The index to search for.
- *handler* (boost::function<void (RR_SHARED_PTR<std::string>, RR_SHARED_PTR<RobotRaconteurException>)>) - The handler for the asynchronous operation.
- *timeout* (int32_t) - The timeout in milliseconds. The default is infinite.

Return Value:

None

void **RobotRaconteurNode::s()->AsyncFindObjRefTyped**(RR_SHARED_PTR<RRObject> *obj*, const std::string& *objrefname*, const std::string& *objecttype*, boost::function<void (RR_SHARED_PTR<RRObject>, RR_SHARED_PTR<RobotRaconteurException>)> *handler*, int32_t *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of AsyncFindObjRefTyped

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The object reference that contains the objref member to query. This object must have been created through ConnectService or returned by an objref.
- *objrefname* (const std::string&) - The member name of the objref to query.
- *objettype* (const std::string&) - The fully qualified object type to return.
- *handler* (boost::function<void (RR_SHARED_PTR<RRObject>, RR_SHARED_PTR<RobotRaconteurException>)>) - The handler for the asynchronous operation.
- *timeout* (int32_t) - The timeout in milliseconds. The default is infinite.

Return Value:

None

void **RobotRaconteurNode::s()->AsyncFindObjRefTyped**(RR_SHARED_PTR<Object> *obj*, const std::string& *objrefname*, const std::string& *index*, const std::string& *object-*

type, boost::function<void (RR_SHARED_PTR<RRObject>, RR_SHARED_PTR<RobotRaconteurException>)> *handler*, int32_t *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of AsyncFindObjRefTyped

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The object reference that contains the objref member to query. This object must have been created through ConnectService or returned by an objref.
- *objrefname* (const std::string&) - The member name of the objref to query.
- *index* (const std::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* (const std::string&) - The fully qualified object type to return.
- *handler* (boost::function<void (RR_SHARED_PTR<RRObject>, RR_SHARED_PTR<RobotRaconteurException>)>) - The handler for the asynchronous operation.
- *timeout* (int32_t) - The timeout in milliseconds. The default is infinite.

Return Value:

None

RR_SHARED_PTR<ServerContext> **RobotRaconteurNode::s()->RegisterService**(const std::string& *name*, const std::string& *servicetype*, RR_SHARED_PTR<RRObject> *obj*,RR_SHARED_PTR<SecurityPolicy> *securitypolicy=RR_SHARED_PTR¡SecurityPolicy¿()*)

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* (const std::string&) - The name of the service. This must be unique within the node.
- *servicetype* (const std::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* (RR_SHARED_PTR<RRObject>) - The root object. It must be compatible with the object type specified in the *servicetype* parameter.

- *securitypolicy* (RR_SHARED_PTR<SecurityPolicy>) - (optional) The security policy for this service.

Return Value:

(ServerContext) - The server context for this service.

void **RobotRaconteurNode::s()->CloseService**(const std::string& *name*)

Closes the service with name *name*.

Parameters:

- *name* (const std::string&) - The name of the service to close.

Return Value:

None

void **RobotRaconteurNode::s()->RequestObjectLock**(
RR_SHARED_PTR<RRObject> *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* (RR_SHARED_PTR<RRObject>) - 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**(RR_SHARED_PTR<RRObject> *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* (RR_SHARED_PTR<RRObject>) - The object to lock. This object must have been created through ConnectService or an objref.

Return Value:

None

void **RobotRaconteurNode::s()->AsyncRequestObjectLock**(RR_SHARED_PTR<RRObject> *obj*, RobotRaconteurObjectLockFlags *flags*, boost::function<void(RR_SHARED_PTR<std::string>, RR_SHARED_PTR<RobotRaconteurException>)> *handler*, int32_t *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of RequestObjectLock.

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - 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* (boost::function<void(RR_SHARED_PTR<std::string>, RR_SHARED_PTR<RobotRaconteurException>)>) - The handler for the asynchronous operation. The string parameter can be ignored.
- *timeout* (int32_t) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

void **RobotRaconteurNode::s()->AsyncReleaseObjectLock**(RR_SHARED_PTR<RRObject> *obj*, boost::function<void(RR_SHARED_PTR<std::string>, RR_SHARED_PTR<RobotRaconteurException>)> *handler*, int32_t *timeout=RR_TIMEOUT_INFINITE*)

Asynchronous version of ReleaseObjectLock.

Parameters:

- *obj* (RR_SHARED_PTR<RRObject>) - The object to lock. This object must have been created through ConnectService or an objref.
- *handler* (boost::function<void(RR_SHARED_PTR<std::string>, RR_SHARED_PTR<RobotRaconteurException>)>) - The handler for the asynchronous operation. The string parameter can be ignored.
- *timeout* (int32_t) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

void **RobotRaconteurNode::s()->MonitorEnter**(RR_SHARED_PTR<RRObject> *obj*, int32_t

timeout=RR_TIMEOUT_INFINITE)

Requests a monitor lock for a connected service object.

Parameters:

- *obj* (`RR_SHARED_PTR<RRObject>`) - The object to lock. This object must have been created through `ConnectService` or an objref.
- *timeout* (`int32_t`) - (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* (`RR_SHARED_PTR<RRObject>`) - The object to lock. This object must have been created through `ConnectService` or an objref.

Return Value:

None

`boost::posix_time::ptime` **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.

Parameters:

None

Return Value:

(`boost::posix_time::ptime`) - The current node time.

`void` **RobotRaconteurNode::s()->Sleep**(`const boost::posix_time::time_duration&` *duration*)

Sleeps for the specified duration in milliseconds.

Parameters:

- *duration* (`const boost::posix_time::time_duration&`) - The duration to sleep.

Return Value:

None

`RR_SHARED_PTR<AutoResetEvent>` **RobotRaconteurNode::s()->CreateAutoResetEvent**()

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

Parameters:

None

Return Value:

(`RR_SHARED_PTR<AutoResetEvent>`) - A new AutoResetEvent

`RR_SHARED_PTR<Rate>` **RobotRaconteurNode::s()->CreateRate**(`double` *frequency*)

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

Parameters:

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

Return Value:

(`RR_SHARED_PTR<Rate>`) - A new rate with the specified frequency

`RR_SHARED_PTR<Timer>` **RobotRaconteurNode::s()->CreateTimer**(
`const boost::posix_time::time_duration&` *period*,
`boost::function<void (const TimerEvent&)>` *handler*, `bool` *oneshot = false*)

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

Parameters:

- *period* (`const boost::posix_time::time_duration&`) - The period of the timer.
- *handler* (`boost::function<void (const 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**(
`boost::function<void (std::exception*)>` *handler*)

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

Parameters:

- *handler* (`boost::function<void (std::exception*)>`) - A function with one parameter that receives the exceptions.

Return Value:

None

`std::vector<ServiceInfo2>` **RobotRaconteurNode::s()->FindServiceByType**(
`const std::string&` *servicetype*, `const std::vector<std::string>&` *transportschemes*)

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

Parameters:

- *servicetype* (`const std::string&`) - The fully qualified type of the root object to search for.
- *transportschemes* (`const std::vector<std::string>&`) - A vector of the schemes to search for.

Return Value:

(`std::vector<ServiceInfo2>`) - A vector of `ServiceInfo2` structures with the detected services.

`std::vector<NodeInfo2>` **RobotRaconteurNode::s()->FindNodeByName**(
`const std::string&` *servicetype*, `const std::vector<std::string>&` *transportschemes*)

Finds a node using auto-discovery based on the NodeName

Parameters:

- *nodename* (`const std::string&`) - The NodeName to search for.
- *transportschemes* (`const std::vector<std::string>`) - A vector of the schemes to search for

Return Value:

(`std::vector<NodeInfo2>`) - A vector of `NodeInfo2` structures with the detected nodes.

`std::vector<NodeInfo2>` **RobotRaconteurNode::s()->FindNodeByID**(
`const NodeID&` *nodeid*, `const std::vector<std::string>&` *transportschemes*)

Finds a node using auto-discovery based on the NodeID

Parameters:

- *nodeid* (`const NodeID&`) - The NodeID to search for.
- *transportschemes* (`const std::vector<std::string>&`) - A vector of the schemes to search for

Return Value:

(`std::vector<NodeInfo2>`) - A vector of `NodeInfo2` structures with the detected nodes.

`void` **RobotRaconteurNode::s()->AsyncFindServiceByType**(
`const std::string&` *servicetype*, `const std::vector<td::string>&` *transportschemes*,
`boost::function<RR_SHARED_PTR<std::vector<ServiceInfo2> > >` *handler*,
`int32_t` *timeout=5000*)

Asynchronous version of `FindServiceByType`.

Parameters:

- *servicetype* (`const std::string&`) - The fully qualified type of the root object to search for.
- *transportschemes* (`const std::vector<std::string>&`) - An array of the schemes to search for.
- *handler* (`boost::function<RR_SHARED_PTR<std::vector<ServiceInfo2> > >`) - The handler for the asynchronous operation.
- *timeout* (`int32_t`) - (optional) The timeout in milliseconds. Default is 5 seconds

Return Value:

None

`void` **RobotRaconteurNode::s()->AsyncFindNodeByName**(
`const std::string&` *nodename*, `const std::string&` *transportschemes*,
`boost::function<RR_SHARED_PTR<std::vector<NodeInfo2> > >` *handler*,
`int32_t` *timeout=5000*)

Asynchronous version of `FindNodeByName`.

Parameters:

- *nodename* (`const std::string&`) - The node to search for.
- *transportschemes* (`const std::vector<std::string>&`) - An array of the schemes to search for.

- *handler* (`boost::function<RR_SHARED_PTR<std::vector<NodeInfo2> > >`) - The handler for the asynchronous operation.
- *timeout* (`int32_t`) - (optional) The timeout in milliseconds. Default is 5 seconds.

Return Value:

None

`void` **RobotRaconteurNode::s()->AsyncFindNodeByID**(
`const NodeID&` *nodeid*, `const std::vector<std::string>&` *transportschemes*,
`boost::function<RR_SHARED_PTR<std::vector<NodeInfo2> > >` *handler*,
`int32_t` *timeout=5000*)

Asynchronous version of `FindNodeByID`.

Parameters:

- *nodeid* (`const NodeID&`) - The node to search for.
- *transportschemes* (`const std::vector<std::string>&`) - An array of the schemes to search for.
- *handler* (`boost::function<RR_SHARED_PTR<std::vector<NodeInfo2> > >`) - The handler for the asynchronous operation.
- *timeout* (`int32_t`) - (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**(`boost::function<void ()>` *handler*)

Asynchronous version of `UpdateDetectedNodes`

Parameters:

- *handler* (`boost::function<void ()>`) - Completion callback function

Return Value:

None

`std::vector<NodeID>` **RobotRaconteurNode::s()->GetDetectedNodes**()

Returns an array of NodeID containing the detected nodes.

Parameters:

None

Return Value:

(`std::vector<NodeID>`) - The detected nodes

`uint32_t` **RobotRaconteurNode::s()->GetEndpointInactivityTimeout**()
`void` **RobotRaconteurNode::s()->SetEndpointInactivityTimeout**(`uint32_t` value)

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

`uint32_t` **RobotRaconteurNode::s()->GetTransportInactivityTimeout**()
`void` **RobotRaconteurNode::s()->SetTransportInactivityTimeout**(`uint32_t` value)

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

`uint32_t` **RobotRaconteurNode::s()->GetTransactionTimeout**()
`void` **RobotRaconteurNode::s()->SetTransactionTimeout**(`uint32_t` value)

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

`uint32_t` **RobotRaconteurNode::s()->GetMemoryMaxTransferSize**()
`void` **RobotRaconteurNode::s()->SetMemoryMaxTransferSize**(`uint32_t` value)

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

`uint32_t` **RobotRaconteurNode::s()->GetNodeDiscoveryMaxCacheCount**()
`void` **RobotRaconteurNode::s()->SetNodeDiscoveryMaxCacheCount**(`uint32_t` value)

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.

B.14 RRMultiDimArray<T>

`template<typename T>`
`class` **RRMultiDimArray**

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

`int32_t` **DimCount**

The number of dimensions.

`RR_SHARED_PTR<RRArray<int32_t> >` **Dims**

The dimensions in column-major order.

`RR_SHARED_PTR<RRArray<T> >` **Real**

The real data in column-major order.

`bool` **Complex**

`true` if the array has complex data.

`RR_SHARED_PTR<RRArray<T> >` **Imag**

The imaginary data in column-major order.

RRMultiDimArray(`RR_SHARED_PTR<RRArray<int32_t> >` *dims*,
`RR_SHARED_PTR<RRArray<T> >` *real*,
`RR_SHARED_PTR<RRArray<T> >` *imag=RR_SHARED_PTR<RRArray<T>>()*)

Creates a new array with the provided data.

Parameters:

- *dims* (`RR_SHARED_PTR<RRArray<int32_t> >`) - The dimensions of the array in column-major order.
- *real* (`RR_SHARED_PTR<RRArray<T> >`) - The real data in column-major order.
- *imag* (`RR_SHARED_PTR<RRArray<T> >`) - (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**(`std::vector<int32_t>` *memorypos*,

RR_SHARED_PTR<RRMultiDimArray<T> > *buffer*, std::vector<int32_t> *bufferpos*, std::vector<int32_t> *count*)

Reads data from the source array into buffer.

Parameters:

- *memorypos* (std::vector<int32_t>) - The start position in the array.
- *buffer* (RR_SHARED_PTR<RRMultiDimArray<T> >) - The buffer to read the data into.
- *bufferpos* (std::vector<int32_t>) - The start position in the buffer.
- *count* (std::vector<int32_t>) - The number of elements to read.

Return Value:

None

void **AssignSubArray**((std::vector<int32_t> *memorypos*, RR_SHARED_PTR<RRMultiDimArray<T> > *buffer*, std::vector<int32_t> *bufferpos*, std::vector<int32_t> *count*)

Writes data from buffer into the array.

Parameters:

- *memorypos* (std::vector<int32_t>) - The start position in the array.
- *buffer* (RR_SHARED_PTR<RRMultiDimArray<T> >) - The buffer to write data from.
- *bufferpos* (std::vector<int32_t>) - The start position in the buffer.
- *count* (std::vector<int32_t>) - The number of elements to read.

Return Value:

None

B.15 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.

`std::string` **NodeName**

The name of the found node.

`NodeID` **NodeID**

The id of the found node.

`std::string` **Name**

The name of the service.

`std::string` **RootObjectType**

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

`std::vector<std::string>` **RootObjectImplements**

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

`std::string` **ConnectionURL**

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

`std::map<std::string,RR_SHARED_PTR<RRObject> >` **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.

B.16 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.

`std::string` **NodeName**

The name of the found node.

`NodeID` **NodeID**

The id of the found node.

`std::vector<std::string>` **ConnectionURL**

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

B.17 Pipe<T>

```
template<typename T>
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.

`std::string` **GetMemberName**()

Returns the member name of this pipe.

`PipeEndpoint<T>` **Connect**(`int32_t` *index=-1*)

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

Parameters:

- *index* (`int32_t`) - (optional) The index of the `PipeEndpoint` pair. This can be -1 to mean "any index".

Return Value:

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

`void` **AsyncConnect**(`int32_t` *index*, `boost::function<void(RR_SHARED_PTR<PipeEndpoint<T> >, RR_SHARED_PTR<RobotRaconteurException> >)` *handler*, `int32_t` *timeout=RR_TIMEOUT_HANDLER*)

Asynchronous version of `Connect`.

Parameters:

- *index* (`int32_t`) - The index of the `PipeEndpoint` pair. This can be -1 to mean "any

index".

- *handler* (boost::function<void(RR_SHARED_PTR<PipeEndpoint<T> >, RR_SHARED_PTR<RobotRaconteurException> >>)) - The handler for the asynchronous operation.
- *timeout* (int32_t) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

boost::function<void(RR_SHARED_PTR<PipeEndpoint<T> >)> **GetPipeConnectCallback**()
void **SetPipeConnectCallback**(boost::function<void(RR_SHARED_PTR<PipeEndpoint<T> >)> value)

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

B.18 PipeEndpoint<T>

template<typename T>
class **PipeEndpoint**

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

uint32_t **GetEndpoint**()

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.

int32_t **GetIndex**()

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

size_t **GetAvailable**()

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*, `boost::function<void(uint32_t,RR_SHARED_PTR<RobotRaconteurExc` *handler*)

Asynchronous version of `SendPacket`

Parameters:

- *packet* (`T`) - The packet to send
- *handler* (`boost::function<void(uint32_t,RR_SHARED_PTR<RobotRaconteurException>)>`) - The handler for the asynchronous operation.

Return Value:

None

`bool` **GetRequstPacketAck**()
`void` **SetRequstPacketAck**(`bool` value)

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**(`boost::function<void(RR_SHARED_PTR<RobotRaconteurException>)>` *handler*, `int32_t` *timeout=2000*)

handler (`boost::function<void(RR_SHARED_PTR<RobotRaconteurException>)>`) - The handler for the asynchronous operation.

timeout (`int32_t`) - (optional) The timeout in milliseconds. Default is 2 seconds.

Parameters:

None

Return Value:

`boost::function<void (RR_SHARED_PTR<PipeEndpoint<T>>)>` **GetPipeEndpointClosedCallback**()
`void` **SetPipeEndpointClosedCallback**(`boost::function<void (RR_SHARED_PTR<PipeEndpoint<T>>)` value)

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

`boost::signals2::signal<void (RR_SHARED_PTR<PipeEndpoint<T> >)>` **PacketReceivedEvent**

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

`boost::signals2::signal<void (RR_SHARED_PTR<PipeEndpoint<T> >,uint32_t)>` **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`.

B.19 Callback<T>

`template<typename T>`
`class` **Callback**

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` **GetFunction**()
`void` **SetFunction**(`T` value)

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

`T` **GetClientFunction**(`uint32_t` *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* (`uint32_t`) - 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.

`T` **GetClientFunction**(`RR_SHARED_PTR<Endpoint>` *endpoint*)

Retrieves a function that will be executed on the client selected by the *endpoint* parame-

ter. The *endpoint* can be determined through `ServerEndpoint::GetCurrentEndpoint()`. This is only available in a service.

Parameters:

- *endpoint* (`uint32_t`) - 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.

B.20 Wire<T>

`template<typename T>`
`class` **Wire**

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

`std::string` **GetMemberName**()

Returns the member name of this wire.

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

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

Parameters:

None

Return Value:

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

`void` **AsyncConnect**(`boost::function<void (RR_SHARED_PTR<WireConnection<T> >, RR_SHARED_PTR<RobotRaconteurException>)>` *handler*,
`int32_t` *timeout=RR_TIMEOUT_HANDLER*)

Asynchronous version of `Connect`.

Parameters:

- *handler* (boost::function<void (RR_SHARED_PTR<WireConnection<T> >, RR_SHARED_PTR<RobotRaconteurException>)>) - The handler for the asynchronous operation.
- *timeout* (uint32_t) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

boost::function<void(RR_SHARED_PTR<WireConnection<T>>)> **GetWireConnectCallback**()
void **SetWireConnectCallback**(boost::function<void(RR_SHARED_PTR<WireConnection<T>>)> valu

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

B.21 WireConnection<T>

template<typename T>
class **WireConnection**

The WireConnection<T> class represents one end of a wire connection which is formed by a pair of WireConnection<T> 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<T> 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.

uint32_T **GetEndpoint**()

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 **GetInValue**()

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` **GetOutValue**()
`void` **SetOutValue**(`T` value)

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` **GetInValueValid**()

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

`bool` **GetOutValueValid**()

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

`TimeSpec` **GetLastValueReceivedTime**()

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` **GetLastValueSentTime**()

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**(`boost::function<void(RR_SHARED_PTR<RobotRaconteurException>)>` *handler*, `int32_t` *timeout=RR_TIMEOUT_INFINITE*)

The asynchronous version of `Close`.

Parameters:

- *handler* (`boost::function<void(RR_SHARED_PTR<RobotRaconteurException>)>`) - The handler for the asynchronous operation.
- *timeout* (`int32_t`) - (optional) The timeout in milliseconds. Default is infinite.

Return Value:

None

`boost::function<void (RR_SHARED_PTR<WireConnection<T> >)>` **GetWireConnectionClosed-Callback**()
`void` **SetWireConnectionClosedCallback**(`boost::function<void (RR_SHARED_PTR<WireConnection<T> >)>` value)

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

`boost::signals2::signal<void (RR_SHARED_PTR<WireConnection<T> > connection, T value, TimeSpec time)>` **WireValueChanged**

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.

B.22 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.

`int64_t` **seconds**

A 64-bit integer representing the seconds.

`int32_t` **nanoseconds**

A 32-bit integer representing the nanoseconds.

TimeSpec(`int64_t` *seconds*, `int32_t` *nanoseconds*)

Creates a new `TimeSpec`.

Parameters:

- *seconds* (`int64_t`) - Seconds
- *nanoseconds* (`int32_t`) - 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.

B.23 ArrayMemory<T>

```
template<typename T>
class ArrayMemory
```

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()

RR_SHARED_PTR<RRArray<T> > *array*

Parameters:

Creates a new ArrayMemory<T>.

Return Value:

array (RR_SHARED_PTR<RRArray<T> >) - The array data.

Creates a new ArrayMemory for use with the new keyword.

uint64_T **LengthGetThe number of elements in the array.**()

void **Read**(uint64_t *memorypos*, RR_SHARED_PTR<RRArray<T> > *buffer*, uint64_t *bufferpos*, uint64_t *count*)

Reads data from the memory into buffer.

Parameters:

- *memorypos* (uint64_t) - The start position in the array.
- *buffer* (RR_SHARED_PTR<RRArray<T> >) - The buffer to read the data into.
- *bufferpos* (uint64_t) - The start position in the buffer.
- *count* (uint64_t) - The number of elements to read.

Return Value:

None

void **Write**(uint64_t *memorypos*, RR_SHARED_PTR<RRArray<T> > *buffer*, uint64_t *bufferpos*, uint64_t *count*)

Writes data from buffer into the memory.

Parameters:

- *memorypos* (uint64_t) - The start position in the array.
- *buffer* (RR_SHARED_PTR<RRArray<T> >) - The buffer to write the data from.
- *bufferpos* (uint64_t) - The start position in the buffer.

- *count* (uint64_t) - The number of elements to read.

Return Value:

None

B.24 MultiDimArrayMemory<T>

template<typename T>
class **MultiDimArrayMemory**

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(RR_SHARED_PTR<RRMultiDimArray<T> > *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.

uint64_t **GetDimCount**()

The number of dimensions in the array.

std::vector<uint64_t> **GetDims**()

The dimensions of the array in column-major order.

bool **GetComplex**()

true if the array is complex, otherwise false.

void **Read**(std::vector<uint64_t> *memorypos*, RR_SHARED_PTR<RRMultiDimArray<T> > *buffer*, std::vector<uint64_t> *bufferpos*, std::vector<uint64_t> *count*)

Reads data from the memory into buffer.

Parameters:

- *memorypos* (`std::vector<uint64_t>`) - The start position in the array.
- *buffer* (`RR_SHARED_PTR<RRMultiDimArray<T> >`) - The buffer to read the data into.
- *bufferpos* (`std::vector<uint64_t>`) - The start position in the buffer.
- *count* (`std::vector<uint64_t>`) - The number of elements to read.

Return Value:

None

`void` **Write**(`std::vector<uint64_t>` *memorypos*, `RR_SHARED_PTR<RRMultiDimArray<T> >` *buffer*, `std::vector<uint64_t>` *bufferpos*, `std::vector<uint64_t>` *count*)

Writes data from `buffer` into the memory.

Parameters:

- *memorypos* (`std::vector<uint64_t>`) - The start position in the array.
- *buffer* (`RR_SHARED_PTR<RRMultiDimArray<T> >`) - The buffer to write the data from.
- *bufferpos* (`std::vector<uint64_t>`) - The start position in the buffer.
- *count* (`std::vector<uint64_t>`) - The number of elements to read.

Return Value:

None

B.25 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 RR_SHARED_PTR<ServerEndpoint>` **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:

(RR_SHARED_PTR<ServerEndpoint>) - The current endpoint number.

static RR_SHARED_PTR<AuthenticatedUser> **GetCurrentAuthenticatedUser**()

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

Parameters:

None

Return Value:

(RR_SHARED_PTR<AuthenticatedUser>) - The current authenticated user.

B.26 ServerContext

class **ServerContext**

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

static std::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:

(std::string) - The current service path.

static RR_SHARED_PTR<ServerContext> **GetCurrentServerContext**()

Returns the current server context for the current service object.

Parameters:

None

Return Value:

(RR_SHARED_PTR<ServerContext>) - The current service context.

void **ReleaseServicePath**(const std::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* (const std::string&) - The service path.

Return Value:

None

void **ReleaseServicePath**(const std::string& *path*, const std::vector<uint32_t>& *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* (const std::string&) - The service path.
- *endpoints* (const std::vector<uint32_t>&) - The endpoints to notify of the release.

Return Value:

None

boost::signals2::signal<void (RR_SHARED_PTR<ServerContext>, ServerServiceListenerEventType, RR_SHARED_PTR<void>)> **ServerServiceListener**

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

std::map<std::string, RR_SHARED_PTR<RRObject> > **GetAttributes**()
void **SetAttributes**(std::map<std::string, RR_SHARED_PTR<RRObject> > value)

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

B.27 AuthenticatedUser

`class` **AuthenticatedUser**

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

`std::string` **GetUsername**()

The username of the authenticated user.

`std::vector<std::string> >` **GetPrivileges**()

The list of privileges for the user.

`boost::posix_time::ptime` **GetLoginTime**()

The login time of the user.

`boost::posix_time::ptime` **GetLastAccessTime**()

The time of last access by the user.

B.28 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(`std::string` *id*)

Creates a new `NodeID`.

Parameters:

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

Return Value:

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

NodeID(`boost::array<uint8_t,16>` *id*)

Creates a new `NodeID`.

Parameters:

- *id* (boost::array<uint8_t,16>) - The value of the NodeID as a 16 byte array.

Return Value:

(NodeID) - The new NodeID.

NodeID(boost::uuids::uuid *id*)

Creates a new NodeID.

Parameters:

- *id* (boost::uuids::uuid) - The value of the NodeID as a uuid.

Return Value:

(NodeID) - The new NodeID.

std::string **ToString**()

Returns the string representation of the NodeID.

Parameters:

None

Return Value:

(std::string) - The string representation.

boost::array<uint8_t,16> **ToByteArray**()

Returns the byte array representation of the NodeID.

Parameters:

None

Return Value:

(boost::array<uint8_t,16>) - The byte array representation.

boost::uuids::uuid **ToUuidArray**()

Returns the byte array representation of the NodeID.

Parameters:

None

Return Value:

(`boost:uuids::uuid`) - The byte array representation.

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

B.29 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(`const std::string&` *data*)

Creates a new `PasswordFileUserAuthenticator`.

Parameters:

- *data* (`const std::string&`) - A string containing the user, password, and privileges information.

Return Value:

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

PasswordFileUserAuthenticator(`std::istream&` *file*)

Creates a new `PasswordFileUserAuthenticator` from a file.

Parameters:

- *file* (`std::istream&`) - An input stream containing the data.

Return Value:

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

B.30 ServiceSecurityPolicy

`class` **ServiceSecurityPolicy**

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

ServiceSecurityPolicy(`RR_SHARED_PTR<UserAuthenticator>` *authenticator*, `const std::map<std::string, std::string>&` *policies*)

Creates a new `ServiceSecurityPolicy`.

Parameters:

- *authenticator* (`RR_SHARED_PTR<UserAuthenticator>`) - The authenticator used to authenticate user. Will typically be `PasswordFileUserAuthenticator`.
- *policies* (`const std::map<std::string,std::string>&`) - The policies for the service.

Return Value:

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

B.31 RobotRaconteurException

`class` **RobotRaconteurException**

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

- `ConnectionException`
- `ProtocolException`
- `ServiceNotFoundException`
- `ObjectNotFoundException`
- `InvalidEndpointException`
- `EndpointCommunicationFatalException`

- NodeNotFoundException
- ServiceException
- MemberNotFoundException
- DataTypeMismatchException
- DataTypeException
- DataSerializationException
- MessageEntryNotFoundException
- UnknownException
- RobotRaconteurRemoteException
- TransactionTimeoutException
- AuthenticationException
- ObjectLockedException

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

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

MessageErrorType **ErrorCode**

The error code of the error.

std::string **Error**

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

std::string **Message**

The message associated with the exception.

B.32 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.

`std::string` **Error**

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

`std::string` **Message**

The message associated with the exception.

B.33 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 std::string` **GetCurrentTransportConnectionURL**()

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

Parameters:

None

Return Value:

(`std::string`) - The URL as a string

B.34 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**(`const 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* (`const NodeID&`) - The "NodeID" to use for the transport and node.

Return Value:

None

`void` **StartServerAsNodeName**(`const std::string&` *nodename*)

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* (`const std::string&`) - The nodename to use

Return Value:

None

`void` **StartClientAsNodeName**(`const std::string&` *nodename*)

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* (`const std::string&`) - The nodename to use

Return Value:

None

B.35 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**(`int32_t` *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* (`int32_t`) - 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`.

Parameters:

None

Return Value:

`int32_t` **GetListenPort**()

Returns the port that the transport is listening for connections on

Parameters:

None

Return Value:

(`int32_t`) - 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` **GetDefaultReceiveTimeout**()
`void` **SetDefaultReceiveTimeout**(`double` value)

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` **GetDefaultConnectTimeout**()
`void` **SetDefaultConnectTimeout**(`double` value)

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

`double` **GetDefaultHeartbeatPeriod**()
`void` **SetDefaultHeartbeatPeriod**(`double` value)

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

`int32_t` **GetMaxMessageSize**()
`void` **SetMaxMessageSize**(`int32_t` value)

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.

`int32_t` **GetMaxConnectionCount**()
`void` **SetMaxConnectionCount**(`int32_t` value)

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.

Parameters:

None

Return Value:

None

`bool` **GetRequireTls**()
`void` **SetRequireTls**(`bool` value)

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

`bool` **IsTransportConnectionSecure**(`RR_SHARED_PTR<RRObject>` *a*)

Returns true if the specified connection is secure.

Parameters:

- *a* (`RR_SHARED_PTR<RRObject>`) - A client object reference

Return Value:

(`bool`) - true if connection is secure.

`bool` **IsTransportConnectionSecure**(`uint32_t` *e*)

Returns true if the specified connection is secure.

Parameters:

- *e* (`uint32_t`) - The local endpoint number

Return Value:

(`bool`) - true if connection is secure

`bool` **IsTransportConnectionSecure**(`RR_SHARED_PTR<Endpoint>` *a*)

Returns true if the specified connection is secure.

Parameters:

- *a* (`RR_SHARED_PTR<Endpoint>`) - An endpoint

Return Value:

(`bool`) - true if connection is secure

`bool` **IsPeerIdentityVerified**(`RR_SHARED_PTR<RRObject>` *a*)

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

Parameters:

- *a* (`RR_SHARED_PTR<RRObject>`) - A client object reference.

Return Value:

(`bool`) - True if peer identity has been verified

`bool` **IsPeerIdentityVerified**(`uint32_t` *e*)

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

Parameters:

- *e* (`uint32_t`) - The local endpoint number

Return Value:

(`bool`) - True if peer identity has been verified

`bool` **IsPeerIdentityVerified**(`RR_SHARED_PTR<Endpoint>` *e*)

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

Parameters:

- *a* (`RR_SHARED_PTR<Endpoint>`) - An endpoint

Return Value:

(`bool`) - True if peer identity has been verified

`std::string` **GetSecurePeerIdentity**(`RR_SHARED_PTR<RRObject>` *a*)

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

Parameters:

- *a* (`RR_SHARED_PTR<RRObject>`) - A client object reference

Return Value:

(`std::string`) - The NodeID as as string.

`std::string` **GetSecurePeerIdentity**(`uint32_t` *e*)

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

Parameters:

- *a* (`uint32_t`) - The local endpoint number

Return Value:

(`std::string`) - The NodeID as as string.

`std::string` **GetSecurePeerIdentity**(`RR_SHARED_PTR<Endpoint>` *e*)

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

Parameters:

- *e* (`RR_SHARED_PTR<Endpoint>`) - An endpoint

Return Value:

(`std::string`) - The NodeID as as string.

`bool` **GetAcceptWebSockets**()
`void` **SetAcceptWebSockets**(`bool` value)

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

`std::vector<std::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:

(`std::vector<std::string>`) - Array containing the current allowed origins.

`void` **AddWebSocketAllowedOrigin**(`const std::string&` *origin*)

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

Parameters:

- *origin* (`const std::string&`) - The origin to add

Return Value:

None

`void` **RemoveWebSocketOrigin**(`const std::string&` *origin*)

Removes an origin from the allowed origin list.

Parameters:

- *origin* (`const std::string&`) - The origin to remove

Return Value:

None

B.36 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.

Parameters:

None

Return Value:

None

B.37 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.

B.38 PipeBroadcaster<T>

`template<typename T>`
`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()

Creates a new `PipeBroadcaster<T>`.

Parameters:

None

Return Value:

None

`void` **Init**(`RR_SHARED_PTR<Pipe<T> >` *pipe*, `int32_t` *backlog=-1*)

Initializes 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* (`RR_SHARED_PTR<Pipe<T> >`) - The pipe to use for broadcasting
- *backlog* (`int32_t`) - (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*, `boost::function<void()>` *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 (`boost::function<void()>`) - 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.

B.39 WireBroadcaster<T>

```
template<typename T>
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 and disconnects.

WireBroadcaster()

Creates a new `WireBroadcaster<T>`.

Parameters:

None

Return Value:

None

`void` **Init**(`RR_SHARED_PTR<Wire<T> >` *wire*)

Initializes 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* (`RR_SHARED_PTR<Wire<T> >`) - The wire to use for broadcasting

Return Value:

(`WireBroadcaster`) - The new instance.

`void` **OutValue**(T value)

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.

Parameters:

- *value* (`T`) - The new value

Return Value:

None