NymphRPC library

Documentation

Date	Author	Changes
2017/12/05	Maya Posch	Original version.
2021/11/26	Maya Posch	Updated to new API.

Table of Contents

Introduction

The Nymph RPC library ('NymphRPC') is an implementation of the Nymph binary Remote Procedure Call (RPC) protocol.

In its current iteration it's fully based on C++ (pre-C++11) and the POCO libraries for cross-platform support, as well as utility functions such as text formatting and logging.

This C++ NymphRPC implementation is light-weight and contains the full implementation for both client and server-based applications.

Folder layout

Being a simple, header-only project, the layout reflects this:

```
/
|- src
|     |- <source files>
|- test
      |- nymph_test_client
      |- nymph_test_server
```

The `src` folder contains the library, with a test client and server implementation in the `test` folder.

Unimplemented features

N/A

Building NymphRPC

NymphRPC is a header-only library. For a C++ project, simply include the <nymph/nymph.h> header and use the `NymphRemoteServer` or `NymphRemoteClient` methods (see below).

NymphRPC's source files (*.cpp) should be compiled and linked into the project as usual. One can also compile the source files into an archive (e.g. .a for GCC) and link against it using just the header files.

The Makefile provided with the project builds a static and dynamic library when executed on a supported platform with:

```
make
```

Installing the library files and headers is performed using:

```
make install
```

Building the test client and server is done by executing any of the following:

```
make test
make test-client
make test-server
```

Cleaning the project can be done using any of:

```
make clean
make clean-test
make clean-test-client
make clean-test-server
```

Dependencies

Nymph depends only on the POCO libraries[1]. One has to link against the Foundation, Net and Util libraries in this order:

1. libPocoNet
2. libPocoUtil
3. libPocoFoundation

Compilers

NymphRPC should compile with any reasonably modern (C++98+) compiler. It has been tested

1 http://pocoproject.org

with various compilers: ranging from 4.4.2 for QNX, 4.7.x Linux-ARMv7, 4.8 for Linux-x86, current GCC (10+) and MSVC (2017+) on Windows and Linux.

Getting started

After including one or both of the NymphRPC header files into the project, the remote client, remote server or both can be used. The following examples show how to create a simple client and server.

Client

The NymphRPC client wants to talk to a remote server, ergo it uses the remote server class. Since the `NymphRemoteServer` class is static, one does not require an instance, but can just call methods on it directly.

A basic example would be:

```
#include "nymph/nymph.h"

#include <iostream>
#include <vector>


void logFunction(int level, std::string logStr) {
 std::cout << level << " - " << logStr << std::endl;
}


// Callback to register with the server.
// This callback will be called once by the server and then discarded. This is
// useful for one-off events, but can also be used for callbacks during the
// life-time of the client.
void callbackFunction(uint32_t session, NymphMessage* msg, void* data) {
      std::cout << "Client callback function called.\n";

      // Remove the callback.
      NymphRemoteServer::removeCallback("helloCallbackFunction");

      msg->discard();
}


int main() {
      // Initialise the remote client instance.
      long timeout = 5000; // 5 seconds.
      NymphRemoteServer::init(logFunction, NYMPH_LOG_LEVEL_TRACE, timeout);

      // Connect to the remote server.
      uint32_t handle;
      std::string result;
      if (!NymphRemoteServer::connect("localhost", 4004, handle, 0, result)) {
            std::cout << "Connecting to remote server failed: " << result <<
std::endl;
```

```
            NymphRemoteServer::disconnect(handle, result);
            NymphRemoteServer::shutdown();
            return 1;
      }

      // Send message and wait for response.
      std::vector<NymphType*> values;
      std::string hello = "Hello World!";
      values.push_back(new NymphType(&hello));
      NymphType* returnValue = 0;
      if (!NymphRemoteServer::callMethod(handle, "helloFunction", values, 
returnValue, result)) {
            std::cout << "Error calling remote method: " << result << std::endl;
            NymphRemoteServer::disconnect(handle, result);
            NymphRemoteServer::shutdown();
            return 1;
      }

      std::string response(returnValue->getChar(), returnValue-
>string_length());

      std::cout << "Response string: " << response << std::endl;

      delete returnValue;

      // Register callback and send message with its ID to the server. Then wait
      // for the callback to be called.
      NymphRemoteServer::registerCallback("callbackFunction", callbackFunction, 
0);
      values.clear();
      std::string cbStr = "callbackFunction";
      values.push_back(new NymphType(&cbStr));
      returnValue = 0;
      if (!NymphRemoteServer::callMethod(handle, "helloCallbackFunction", 
values, returnValue, result)) {
            std::cout << "Error calling remote method: " << result << std::endl;
            NymphRemoteServer::disconnect(handle, result);
            NymphRemoteServer::shutdown();
            return 1;
      }

      if (!returnValue->getBool()) {
            std::cout << "Remote method returned false. " << result << 
std::endl;
            NymphRemoteServer::disconnect(handle, result);
            NymphRemoteServer::shutdown();
            return 1;
      }

      delete returnValue;

      // Shutdown.
      NymphRemoteServer::disconnect(handle, result);
      NymphRemoteServer::shutdown();
      return 0;
}
```

This example client connects to the remote server, calls a simple function on the server, then registers a callback method, calls a method on the server that calls this callback, followed by disconnecting from the remote server.

Note: In a functional implementation, the client should wait for the callback to be called. This is beyond the scope of this example.

Exceptions

The response message from the server can contain an exception rather than a response. The client is responsible for checking whether an exception occurred, and obtaining the exception ID and value string from the response (a `NymphMessage` instance).

Server

Like the remote client implementation, the `NymphRemoteServer` is a static class and can be used without being instantiated:

```
#include “nymph/nymph.h”

#include <iostream>
#include <vector>
#include <csignal>


#include <Poco/Condition.h>
#include <Poco/Thread.h>


Poco::Condition gCon;
Poco::Mutex gMutex;


void signal_handler(int signal) {
      gCon.signal();
}


// --- LOG FUNCTION ---
void logFunction(int level, std::string logStr) {
      std::cout << level << " - " << logStr << std::endl;
}


// --- HELLO ---
// Callback for helloFunction.
NymphMessage* hello(int session, NymphMessage* msg, void* data) {
      std::cout << "Received message for session: " << session << ", msg ID: "
<< msg->getMessageId() << "\n";

      NymphType* nt = msg->parameters()[0];
      std::string* echoStr = new std::string(nt->getChar(), nt-
>string_length());
      std::cout << "Message string: " << *echoStr << "\n";

      NymphMessage* returnMsg = msg->getReplyMessage();
      NymphType* world = new NymphType(echoStr, true);
      returnMsg->setResultValue(world);
      msg->discard();
      return returnMsg;
}


// --- HELLO CALLBACK ---
// Callback for helloCallback. Called to register a client-side callback which
// we will call right afterwards.
NymphMessage* helloCallback(int session, NymphMessage* msg, void* data) {
      std::cout << "Received message for session: " << session << ", msg ID: "
<< msg->getMessageId() << "\n";

      NymphType* nt = msg->parameters()[0];
      std::string* echoStr = new std::string(nt->getChar(), nt-
>string_length());
      std::cout << "Client callback method name: " << *echoStr << "\n";

      // Register and call the callback method ('callbackFunction') on the
client.
      std::vector<NymphTypes> parameters;
      parameters.push_back(NYMPH_STRING);
      NymphMethod remoteMethod(*echoStr, parameters, NYMPH_NULL);
      remoteMethod.enableCallback();
      NymphRemoteClient::registerCallback(*echoStr, remoteMethod);

      std::vector<NymphType*> values;
      values.push_back(new NymphType(echoStr, true));
      std::string result;
      NymphMessage* returnMsg = msg->getReplyMessage();
      NymphType* world = new NymphType(true);
      if (!NymphRemoteClient::callCallback(session, *echoStr, values, result)) {
            std::cerr << "Calling callback failed: " << result << std::endl;
            world->setValue(false);
      }

      returnMsg->setResultValue(world);
      msg->discard();
      return returnMsg;
}


int main() {
      // Initialise the server instance.
      std::cout << "Initialising server..." << std::endl;
      long timeout = 5000; // 5 seconds.
      NymphRemoteClient::init(logFunction, NYMPH_LOG_LEVEL_TRACE, timeout);

      // Register methods to expose to the clients.
      std::cout << "Registering methods...\n";
      std::vector<NymphTypes> parameters;
      parameters.push_back(NYMPH_STRING);
      NymphMethod helloFunction("helloFunction", parameters, NYMPH_STRING,
hello);
      NymphRemoteClient::registerMethod("helloFunction", helloFunction);

      NymphMethod helloCallbackFunction("helloCallbackFunction", parameters,
NYMPH_BOOL, helloCallback);
      NymphRemoteClient::registerMethod("helloCallbackFunction",
helloCallbackFunction);

      // Install signal handler to terminate the server.
      signal(SIGINT, signal_handler);

      // Start server on port 4004.
      NymphRemoteClient::start(4004);

      // Loop until the SIGINT signal has been received.
      gMutex.lock();
      gCon.wait(gMutex);

      // Clean-up
      NymphRemoteClient::shutdown();

      // Wait before exiting, giving threads time to exit.
      Poco::Thread::sleep(2000); // 2 seconds.

      return 0;
}
```

This code will create a server, which will start listening on port 4004, using all interfaces. It'll handle new clients and match any incoming client requests matching one of the registered methods, calling the registered callback.

API

The NymphRPC API is split up into three sections: general, server and client. The former is used by both server and client, while the latter two are specific to their use cases (server or client).

General

As part of the NymphRPC library, a number of types are defined, each using the NymphType class.

Type enumeration

The enumeration for these types is:

```
enum NymphTypes {
      NYMPH_NULL  = 0,
      NYMPH_ARRAY,
      NYMPH_BOOL,
      NYMPH_UINT8,
      NYMPH_SINT8,
      NYMPH_UINT16,
      NYMPH_SINT16,
      NYMPH_UINT32,
      NYMPH_SINT32,
      NYMPH_UINT64,
      NYMPH_SINT64,
      NYMPH_FLOAT,
      NYMPH_DOUBLE,
      NYMPH_STRING,
      NYMPH_STRUCT,
      NYMPH_ANY
};
```

NymphType

This class is used for all NymphRPC values.

Class interface

The interface offered by the type class allows for instantiating in a variety of ways:

```
NymphType() { }
NymphType(bool v);
NymphType(uint8_t v);
NymphType(int8_t v);
NymphType(uint16_t v);
NymphType(int16_t v);
NymphType(uint32_t v);
NymphType(int32_t v);
NymphType(uint64_t v);
NymphType(int64_t v);
NymphType(float v);
NymphType(double v);
NymphType(char* v, uint32_t bytes, bool own = false);
NymphType(std::string* v, bool own = false);
NymphType(std::vector<NymphType*>* v, bool own = false);
NymphType(std::map<std::string, NymphPair>* v, bool own = false);
```

Values can be obtained in a variety of ways:

```
bool getBool(bool* v = 0);
uint8_t getUint8(uint8_t* v = 0);
int8_t getInt8(int8_t* v = 0);
uint16_t getUint16(uint16_t* v = 0);
int16_t getInt16(int16_t* v = 0);
uint32_t getUint32(uint32_t* v = 0);
int32_t getInt32(int32_t* v = 0);
uint64_t getUint64(uint64_t* v = 0);
int64_t getInt64(int64_t* v = 0);
float getFloat(float* v = 0);
double getDouble(double* v = 0);
const char* getChar(const char* v = 0);
std::vector<NymphType*>* getArray(std::vector<NymphType*>* v = 0);
std::map<std::string, NymphPair>* getStruct(std::map<std::string, NymphPair>* v
= 0);
```

Some utility functions are provided:

```
std::string getString();
bool getStructValue(std::string key, NymphType* &value);
```

Values can be set after instantiation as well:

```
void setValue(bool v);
void setValue(uint8_t v);
void setValue(int8_t v);
void setValue(uint16_t v);
void setValue(int16_t v);
void setValue(uint32_t v);
void setValue(int32_t v);
void setValue(uint64_t v);
void setValue(int64_t v);
void setValue(float v);
void setValue(double v);
void setValue(char* v, uint32_t bytes, bool own = false);
void setValue(std::string* v, bool own = false);
void setValue(std::vector<NymphType*>* v, bool own = false);
void setValue(std::map<std::string, NymphPair>* v, bool own = false);
```

In the case of a string type, access to the internal data pointer and length of the string is provided with:

```
uint64_t bytes();
uint32_t string_length();
```

The stored value type can be obtained with:

```
NymphTypes valuetype();
```

A message is serialized to binary format for transmission with:

```
void serialize(uint8_t* &index);
```

The following are internal functions, used with reference counting:

```
void linkWithMessage(NymphMessage* msg);
void triggerAddRC();
```

After being done with a message on the receiving end, the following method discards the message and cleans up any resources:

```
void discard();
```

NymphArray

Defined as type NYMPH_ARRAY. A NymphRPC array defines an array of values, which can be of any type, including other NymphRPC arrays. It is defined in the C++ implementation as an std::vector<NymphType*> instance.

NymphMessage

This is a class which wraps NymphRPC messages, used by both the server and client side implementations.

```
NymphMessage()
NymphMessage(uint32_t methodId)
NymphMessage(uint8_t* binmsg, uint64_t bytes)
```

Constructor. Creates a new NymphMessage instance.		
methodId	uint32_t	Numeric method ID of the target method.
binmsg	uint8_t*	The binary NymphRPC message, minus the signature and length sections.
bytes	uint64_t	The size of the binary message in bytes.

```
bool addValue(NymphType* value)
bool addValues(std::vector<NymphType*> &values)
```

Adds a new value or values to the message.		
value	NymphType*	The NymphType value.
values	std::vector	A vector with NymphType* values.
Returns: a boolean value indicating success or failure.		

```
void serialize(string &output)
```

Generates the binary NymphRPC message and updates internal buffer.
Returns: nothing.

```
uint8_t* buffer()
```

Returns a pointer to the internal binary message buffer.
Returns: pointer to the binary message.

```
uint32_t buffer_size()
```

Returns the size in bytes of the internal binary message buffer.

Returns: the buffer size in bytes.

```
int getState()
```

Get the message state after parsing a binary NymphRPC message to check for errors.
Returns: an integer value with 0 indicating no error and a negative value that an error occurred.

```
bool isCorrupt()
```

Check the message's 'corrupt' flag, which indicates whether the parsed message was corrupt.
Returns: *true* if the parsed binary message was found to be corrupted.

```
bool isCallback()
```

Whether this message is a callback message.
Returns: *true* if this message targets a callback function.

```
void setInReplyTo(uint64_t msgId)
uint64_t getResponseId()
```

Set or get the message ID that this message is a response to.		
msgId	uint64_t	The message ID.
Returns: the getter method returns the message ID.		

```
uint64_T getMessageId()
```

Get the message ID.
Returns: the message ID for this message instance.

```
void setResultValue(NymphType* value)
```

Set the result value for a response message. This is generally used on the server side.		
value	NymphType*	The NymphType value instance.
Returns: nothing.		

```
NymphType* getResponse(bool take = false)
```

Obtain the response value in the message instance.		
take	bool	Set to *true* if taking ownership of the response value.
Returns: a pointer to an NymphType class instance containing the response value, or 0 if no response value was available.		

```
std::vector<NymphType*>& parameters()
```

Get the vector with function parameter values.
Returns: a vector containing pointers to the parameter values for the function that was called.

```
uint32_t getMethodId()
```

Obtain the method ID for this message.
Returns: The method ID.

```
NymphMessage* getReplyMessage()
```

Returns a new NymphMessage instance pointer with the response ID and new message ID filled in. This message instance is to be used as a response to the message instance it's being called on.
Returns: The newly constructed NymphMessage instance pointer.

```
NymphException getException()
```

Get the exception set for this message, if any.
Returns: The NymphException instance contained in the message.

```
bool setException(int exceptionId, std::string value)
```

Sets an exception on the message. This message will be interpreted as an exception by the receiver.		
exceptionId	int	The numeric ID of the exception.
value	std::string	A human-readable string detailing the exception.
Returns: a boolean value indicating whether setting the exception succeeded or not.		

```
bool isException()
```

Check if the message contains an exception.
Returns: *true* if the message contains an exception.

```
std::string getCallbackName()
```

If the message targets a callback, this returns the callback name.
Returns: the callback name or an empty string.

```
bool setCallback(std::string name)
```

Set the callback to call. Set by the server for a client.		
name	std::string	The callback name.
Returns: a boolean value indicating whether the action succeeded.		

```
bool isReply()
```

Check whether this message instance is a reply to a previous method call.
Returns: a boolean value indicating whether this is a reply or not.

```
void discard()
```

Discards the message and clean up allocated resources. Required to be called by the owner.
Returns: nothing.

NymphMethod

The NymphMethod class handles the calling and validation of remote methods. It is also used on the server side to call the callback method registered with that method when called by a remote client.

It has these methods:

```
NymphMethod(std::string name, std::vector<NymphTypes> parameters, NymphTypes
retType)
NymphMethod(std::string name, std::vector<NymphTypes> parameters, NymphTypes
retType, NymphMethodCallback cb)
```

Constructor. Takes the method name and a definition of its parameters.		
name	string	The name of the method.
parameters	std::vector<NymphTypes>	Parameter definition for this method.
retType	NymphTypes	The return value type.
cb	NymphMethodCallback	The callback associated with this method.

```
void setCallback(NymphMethodCallback callback)
```

Server-side method. Set the callback method to call when a client calls this method.

This method has the following signature:

```
typedef NymphMessage* (*NymphMethodCallback)(int session, NymphMessage* msg, void* data);
```

callback	NymphMethodCallback	The callback function pointer.
Returns: void.		

```
NymphMessage* callCallback(int handle, NymphMessage* msg)
```

550

Server-side method. Call the callback for this method.		
handle	int	The session handle.
msg	NymphMessage	The message received from the client.
Returns: an NymphMessage instance containing the response from the callback, to be sent to the remote client.		

```
bool call(Net::StreamSocket* socket, NymphRequest* &request,
std::vector<NymphType*> &values, std::string &result)
```

Call the remote method registered with this method instance. This method is used internally. This method will validate the provided vector of values with the registered `NymphMethodParameter` instances.

The order, number of and type of parameters have to match exactly.

socket	sstd::tring	The name of the key.
request	Poco::Net::StreamSocket*	The socket to send the message on.
values	std::vector<NymphType*>	Vector with NymphType-derived values to be serialised.

result	std::string	Set to an error message on failure.
Returns: a boolean value indicating success or failure. On failure the result parameter will be set to the error message.		

```
bool call(NymphSession* session, std::ector<NymphType*> &values, std::string
&result)
```

Call the remote method registered with this method instance. This method is used internally. This method will validate the provided vector of values with the registered `NymphMethodParameter` instances. The order, number of and type of parameters have to match exactly.		
session	NymphSession*	The NymphSession instance to send the message on.
values	std::vector<NymphType*>	Vector with NymphType-derived values to be serialised.
result	std::string	Set to an error message on failure.
Returns: a boolean value indicating success or failure. On failure the result parameter will be set to the error message.		

NymphRemoteServer

This is a static class, used by the client side in order to communicate with the remote NymphRPC server. See the basic client example for a usage example.

This class has the following methods:

```
static bool init(logFnc logger, int level = NYMPH_LOG_LEVEL_TRACE, long timeout
= 3000)
```

Initialise the NymphRemoteServer static class. This allows one to set the time-out period for message calls.		
logger	logFnc	The logger function. See **setLogger()**.
level	int	The level to log at. See **setLogger()**.
timeout	long int	The time-out for a remote method call (in milliseconds). Default is 3 s.
Returns: a boolean indicating success or failure.		

`static void **setLogger**(logFnc logger, int level)`

Sets the logger method which the NymphRPC runtime will use to write log messages to, as well as the log level to use.
 	
This method is supposed to have the following format:
 	
`typedef void (*logFnc)(int, std::string);`
 	
`Log levels:`
 	
`NYMPH_LOG_LEVEL_FATAL = 1,`
`NYMPH_LOG_LEVEL_CRITICAL,`
`NYMPH_LOG_LEVEL_ERROR,`
`NYMPH_LOG_LEVEL_WARNING,`
`NYMPH_LOG_LEVEL_NOTICE,`
`NYMPH_LOG_LEVEL_INFO,`
`NYMPH_LOG_LEVEL_DEBUG,`
`NYMPH_LOG_LEVEL_TRACE`		

logger	logFnc	Function pointer to the logging method.
level	int	The log level (see summary).
Returns: void.		

`static bool **shutdown**()`

Shuts down the NymphRemoteServer's features. This will terminate any existing connections with the remote Nymph server.
Returns: a boolean value indicating success or failure.

580

static bool **connect**(std::string host, int port, uint32_t &handle, void* data, std::string &result)
static bool **connect**(std::string url, uint32_t &handle, void* data, std::string &result)
static bool **connect**(Poco::Net::SocketAddress sa, int &handle, void* data, std::string &result)

Establish a connection with the remote Nymph server.		
host	std::string	The host name, as IP or DNS name.
port	int	The remote port.
url	std::string	Host name and port in <IP/DNS>:<port> format.
sa	Poco::Net::SocketAddress	Remote address in Poco SocketAddress format.
handle	uint32_t	Unique handle for the new connection.
result	std::string	Error messages in text format.
Returns: a boolean value indicating success or failure. On success the handle parameter will contain the new value, on failure the result string will contain the error message.		

```
static bool disconnect(uint32_t handle, string &result)
```

Disconnect the specified connection.		
handle	`uint32`	The connection handle.
result	string	Error messages in text format.
Returns: a boolean value indicating success or failure. On failure the result parameter will contain the error message.		

```
static bool registerMethod(string name, NymphMethod method)
```

Register a NymphRPC method.		
name	string	The name of the method.
method	NymphMethod	The NymphMethod instance to register.
Returns: a boolean value indicating success or failure.		

```
static bool callMethod(uint32_t handle, std::string name,
std::vector<NymphType*> &values, NymphType* &returnvalue, std::string &result)
```

Call a method by its name. This will send a message containing the provided values on the connection identified by the handle (obtained via a call to connect). The return value is provided as a parameter if available.		
handle	`uint32_t`	The connection handle.
name	std::string	The name of the method.
values	vector<NymphType*>	Vector of values to set in the message.
returnValue	NymphType*	Method return value as NymphType-derived value. Owned by caller.
result	std::string	Set to an error message on failure.
Returns: a boolean indicating success or failure.		

```
static bool callMethodId(uint32_t handle, uint32_t id, std::vector<NymphType*>
&values, NymphType* &returnvalue, std::string &result);
```

Call a method by its NymphRPC method id. This will send a message containing the provided values on the connection identified by the handle (obtained via a call to connect). The return value is provided as a parameter if available.		
handle	uint32_t	The connection handle.

id	uint32_t	The ID of the method name.
values	std::vector<NymphType*>	Vector of values to set in the message.
returnValue	NymphType*	Method return value as NymphType value. Owned by caller.
result	std::string	Set to an error message on failure.
Returns: a boolean indicating success or failure. On success the returnValue parameter will be set to the method's return value. On failure the result parameter will be set to an error message.		

```
static bool removeMethod(uint32_t handle, std::string name)
```

Remove the specified method.		
handle	uint32_t	The session handle.
name	std::string	The name of the method.
Returns: a bool indicating success or failure.		

```
static bool registerCallback(std::string name, NymphCallbackMethod method, void*
data)
```

Client-side method. Register a callback to be called when the specified method is called. The signature for the NymphCallbackMethod is: `typedef void (*NymphCallbackMethod)(NymphMessage* msg, void* data);`		
name	std::string	The name of the callback method.
method	NymphCallbackMethod	The callback function pointer.
data	void*	Optional. Data to be passed to the callback method.
Returns: a boolean value indicating success or failure.		

```
static bool removeCallback(std::string name)
```

Remove the specified callback.		
name	std::string	The name of the callback method.
Returns: a pointer to an NymphType class instance containing the value, or 0 if not found.		

NymphRemoteClient

This is a static class used by NymphRPC server-side applications. See the example code for a usage example.

This class implements the following methods:

```
static bool init(logFnc logger, int level = NYMPH_LOG_LEVEL_TRACE, long timeout
= 3000)
```

Initialise the NymphRemoteClient static class. This allows one to set the time-out period for message calls.

logger	logFnc	The logger function. See **setLogger()**.
level	int	The log level. See **setLogger()**.
timeout	long int	The time-out for a remote method call (in milliseconds). Default is 3 s.

Returns: a boolean indicating success or failure.

```
static void setLogger(logFnc logger, int level)
```

Sets the logger method which the Nymph runtime will use to write log messages to, as well as the log level to use. **Note:** This method must be set or a segmentation fault will occur.

This method is supposed to have the following format:

```
typedef void (*logFnc)(int, std::string);
```

```
Log levels:

     NYMPH_LOG_LEVEL_FATAL = 1,
     NYMPH_LOG_LEVEL_CRITICAL,
     NYMPH_LOG_LEVEL_ERROR,
     NYMPH_LOG_LEVEL_WARNING,
     NYMPH_LOG_LEVEL_NOTICE,
     NYMPH_LOG_LEVEL_INFO,
     NYMPH_LOG_LEVEL_DEBUG,
     NYMPH_LOG_LEVEL_TRACE
```

logger	logFnc	Function pointer to the logging method.
level	int	The log level (see summary).

Returns: void.

```
static bool start(int port = 4004)
```

Start the listening socket for this server.

port	int	Port to listen on.
Returns: a boolean value indicating success or failure.		

```
static bool shutdown()
```

Shuts down the runtime. Stops the listening socket and disconnects any clients.
Returns: a boolean value indicating success or failure.

```
static bool registerMethod(std::string name, NymphMethod method)
```

Register a NymphRPC method. The callback has to be set on this NymphMethod instance.		
name	std::string	The name of the method.
method	NymphMethod	The NymphMethod instance to register.
Returns: a boolean value indicating success or failure.		

```
static bool callMethodCallback(int handle, uint32_t methodId, NymphMessage* msg,
NymphMessage* &response)
```

Call the callback registered for this method with the provided NymphMessage instance.		
handle	uint32_t	The NymphRPC session handle.
methodId	uint32_t	The ID for the method.
msg	NymphMessage*	The received message to be passed to the callback.
response	NymphMessage*	Response message from the callback method.
Returns: a boolean value indicating success or failure. On success the response parameter will be set to the callback response value.		

```
static bool removeMethod(std::string name)
```

Remove the specified method.		
name	std::string	The name of the method.
Returns: a bool indicating success or failure.		

```
static bool registerCallback(std::string name, NymphMethod method)
```

Register a NymphRPC callback method.

name	std::string	The name of the method.
method	NymphMethod	The NymphMethod instance to register.
Returns: a boolean value indicating success or failure.		

```
static bool callCallback(uint32_t handle, std::string name,
std::vector<NymphType*> &values, std::string &result)
```

Call the method for the specified callback name with the provided values.		
handle	uint32_t	The NymphRPC session handle.
name	std::string	The name of the method to call.
values	vector<NymphType*>	Values to send to the client with the method call.
result	std::string	Set to an error message if not successful.
Returns: a boolean value indicating success or failure. On failure the result parameter will be set to an error message.		

```
static bool removeCallback(std::string name)
```

Remove the specified callback method.		
name	std::string	The name of the method.
Returns: a bool indicating success or failure.		

```
static bool addSession(int handle, NymphSession* session)
```

Remove the specified callback method.		
handle	int	The handle for the NymphRPC session.
session	NymphSession*	Pointer to the NymphSession instance.
Returns: a bool indicating success or failure.		

```
static bool removeSession(int handle)
```

Remove the specified callback method.		
handle	int	Handle of the NymphRPC session to remove.
Returns: a bool indicating success or failure.		