RCFProto User Guide

Table of Contents

Version

This documentation applies to **RCFProto 1.0**.

Please send any questions or comments to support@deltavsoft.com .

Introduction

What is RCFProto?

RCFProto is a cross platform RPC framework supporting 4 languages - C++, C#, Java and Python. RCFProto uses Protocol Buffers to define message types, which means you get automatically versioned, efficient and portable messages, based on a proven and widespread technology. RCFProto's networking subsystem is based on RCF, providing fast and scalable messaging, and a host of additional features such as publish/subscribe and callback connections.

To get started with RCFProto, download the relevant distribution (binaries for Windows, and source for non-Windows), and get started building the included demos.

Alternatively, read through the RCFProto User Guide first, to get a better understanding of what RCFProto is and what it can do for you.

Why should I use RCFProto?

RCFProto is a natural fit if you need a RPC solution involving any of the four languages that RCFProto supports. Protocol Buffers provides an elegant and widely adopted solution to the problem of versioning network messages, giving your application an easy upgrade path. Protocol Buffer messages are small and fast, giving you solid performance while still retaining a practical and convenient set of features.

Many backend systems today are written in C++, while needing to communicate with front-end systems written in languages such as C# and Java. RCFProto is a great fit for these situations. RCFProto gives you a powerful RPC system for front-end/back-end communication, while also allowing you to use RCF for high performance native C++ RPC calls within the back-end. A single RCFProto server can seamlessly serve RCFProto clients on one port, and native C++ RCF clients on another.

Tutorial

This tutorial provides an overview of writing RCFProto servers and clients.

For fully functional RCFProto demo servers and clients, in all four supported languages, complete with build instructions for common toolsets, please refer to the demo directories in the RCFProto distributions.

RCFProto client/server example

Let's start with an example of a RCFProto client communicating with a RCFProto server.

Defining a Protocol Buffers interface

The first thing we will need is a Protocol Buffers interface, to describe the interaction between the client and the server.

Protocol Buffers provides a simple language for defining message types and services. We won't cover the details of this language here, as it's features are well documented on the Protocol Buffers website. For the purposes of this tutorial, we want to define a service with a single method, that executes a search request and returns search hits.

Protocol Buffers messages and services are defined in .proto files. The following file, named `Tutorial.proto`, defines a service named `SearchService`, with associated `SearchRequest` and `SearchResponse` message types :

```
// Tutorial.proto

// Enable generation of Protocol Buffers RPC services.
option cc_generic_services = true;
option java_generic_services = true;
option py_generic_services = true;

// Defining a search service.

// Define a request.

message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3 [default = 10];
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  optional Corpus corpus = 4 [default = UNIVERSAL];
}

// Define a response

message SearchResponse {
  repeated group Result = 1 {
    required string url = 2;
    optional string title = 3;
    repeated string snippets = 4;
  }
}

// Define the service.

service SearchService
{
  rpc Search (SearchRequest) returns (SearchResponse);
}
```

To make use of this interface in our program, we first need to use the Protocol Buffers compiler, `protoc`, to generate the source code corresponding to the interface. This example will be in C++, so we instruct `protoc` to generate C++ source code:

```
protoc Tutorial.proto --cpp_out=.
```

This will produce two files, `Tutorial.pb.cc` and `Tutorial.pb.h`, in the same directory as `Tutorial.proto`. `Tutorial.pb.h` provides C++ definitions of message and service types, and will be included into our source code. `Tutorial.pb.cc` provides the corresponding source code for the message and service types, and needs to be compiled into both the server and the client.

C++ server

Here is the complete source code for a C++ RCFProto server, implementing the service we have defined:

```
#include <iostream>

#include <RCFProto.hpp>

// Include protoc-generated header.
#include "../bin/Tutorial.pb.h"

using namespace google::protobuf;

// SearchService implementation.
class SearchServiceImpl : public SearchService
{
public:

    // Search() method implementation.
    void Search(
        RpcController *                 controller,
        const SearchRequest *           request,
        SearchResponse *                response,
        Closure *                       done)
    {
        // Fill in the response.

        SearchResponse_Result * result = response->add_result();
        result->set_title("First result");
        result->set_url("http://acme.org/");
        result->add_snippets("A snippet from acme.org.");

        result = response->add_result();
        result->set_title("Second result");
        result->set_url("http://acme.org/abc");
        result->add_snippets("Another snippet from acme.org.");

        // Send response back to the client.
        done->Run();
    }

};

int main()
{
    try
    {
        // Initialize RCFProto.
        RCF::init();

        // Create server.
        RCF::RcfProtoServer server( RCF::TcpEndpoint("0.0.0.0", 50001) );

        // Bind Protobuf service.
        SearchServiceImpl searchServiceImpl;
        server.bindService(searchServiceImpl);

        // Start the server.
        server.start();

        // Wait for clients.
        std::cout << "Press Enter to exit." << std::endl;
        std::cin.get();
    }
    catch(const RCF::Exception & e)
    {
        std::cout << "RCF::Exception: " << e.getErrorString() << std::endl;
        return 1;
    }

    return 0;
}
```

Let's now step through this line by line.

We need to include RCFProto headers:

```
#include <RCFProto.hpp>
```

We need to include the generated headers for the Protocol Buffers message and service definitions:

```
// Include protoc-generated header.
#include "../bin/Tutorial.pb.h"
```

`SearchService` is an abstract interface class generated by Protocol Buffers. Our implementation must derive from this class, and implement the `Search()` method:

```
// SearchService implementation.
class SearchServiceImpl : public SearchService
{
public:

    // Search() method implementation.
    void Search(
        RpcController *                 controller,
        const SearchRequest *           request,
        SearchResponse *                response,
        Closure *                       done)
    {
        // Fill in the response.

        SearchResponse_Result * result = response->add_result();
        result->set_title("First result");
        result->set_url("http://acme.org/");
        result->add_snippets("A snippet from acme.org.");

        result = response->add_result();
        result->set_title("Second result");
        result->set_url("http://acme.org/abc");
        result->add_snippets("Another snippet from acme.org.");

        // Send response back to the client.
        done->Run();
    }

};
```

`RpcController` and `Closure` are classes from Protocol Buffers, that allow us to control the server-side execution of a remote call. For a simple implementation like this one, it's enough to call `Closure::Run()` at the bottom of the implementation function, to signal that the response is ready to be sent back to the client:

```
// Send response back to the client.
done->Run();
```

In `main()`, we need to initialize RCFProto:

```
// Initialize RCFProto.
RCF::init();
```

We create the server, and pass in the endpoint we want to the server to listen on:

```
// Create server.
RCF::RcfProtoServer server( RCF::TcpEndpoint("0.0.0.0", 50001) );
```

We have provided `0.0.0.0` as the IP address of the TCP endpoint, in order to listen on all available network interfaces of the local system.

We bind our `SearchService` implementation to the server:

```
// Bind Protobuf service.
SearchServiceImpl searchServiceImpl;
server.bindService(searchServiceImpl);
```

Finally, we start the server and wait for clients to call in:

```
// Start the server.
server.start();

// Wait for clients.
std::cout << "Press Enter to exit." << std::endl;
std::cin.get();
```

C++ client

Here is the complete C++ code for a RCFProto client calling the server:

```
#include <iostream>

#include <RCFProto.hpp>

// Include protoc-generated header.
#include "../bin/Tutorial.pb.h"

// TextFormat::PrintToString()
#include <google/protobuf/text_format.h>

using namespace google::protobuf;

int main()
{
    try
    {
        // Initialize RCFProto.
        RCF::init();

        // Create request object.
        SearchRequest request;
        request.set_query("something to search for");
        request.set_result_per_page(10);
        request.set_page_number(0);
        request.set_corpus(SearchRequest_Corpus_NEWS);

        // Create response object.
        SearchResponse response;

        // Create channel.
        RCF::RcfProtoChannel channel( RCF::TcpEndpoint("127.0.0.1", 50001) );

        // Create service stub.
        SearchService::Stub stub(&channel);

        // Print out request.
        std::string strRequest;
        TextFormat::PrintToString(request, &strRequest);
        std::cout << "Sending request:" << std::endl;
        std::cout << strRequest << std::endl;

        // Make a synchronous remote call to server.
        stub.Search(NULL, &request, &response, NULL);

        // Print out response.
        std::string strResponse;
        TextFormat::PrintToString(response, &strResponse);
        std::cout << "Received response:" << std::endl;
        std::cout << strResponse << std::endl;
    }
    catch(const RCF::Exception & e)
    {
        std::cout << "RCF::Exception: " << e.getErrorString() << std::endl;
        return 1;
    }

    return 0;
}
```

Let's step through this line by line.

We need to include the RCFProto headers:

```
#include <RCFProto.hpp>
```

We need to include the generated Protocol Buffers headers:

```
// Include protoc-generated header.
#include "../bin/Tutorial.pb.h"
```

We are also including the Protocol Buffers header for the `TextFormat` class, in order to be able to convert Protocol Buffers messages to text format, for log output:

```
// TextFormat::PrintToString()
#include <google/protobuf/text_format.h>
```

In `main()`, we start by initializing RCFProto:

```
// Initialize RCFProto.
RCF::init();
```

Protocol Buffers has defined the `SearchRequest` and `SearchResponse` classes for us. We create a `SearchRequest` object:

```
// Create request object.
SearchRequest request;
request.set_query("something to search for");
request.set_result_per_page(10);
request.set_page_number(0);
request.set_corpus(SearchRequest_Corpus_NEWS);
```

, as well as a `SearchResponse` object:

```
// Create response object.
SearchResponse response;
```

Before we can send the request, we need to create a `RcfProtoChannel` to send it over. In this case the server is listening on TCP port 50001 on the local network interface.

```
// Create channel.
RCF::RcfProtoChannel channel( RCF::TcpEndpoint("127.0.0.1", 50001) );
```

Protocol Buffers has generated a service stub type, which we instantiate:

```
// Create service stub.
SearchService::Stub stub(&channel);
```

Let's print the request:

```
// Print out request.
std::string strRequest;
TextFormat::PrintToString(request, &strRequest);
std::cout << "Sending request:" << std::endl;
std::cout << strRequest << std::endl;
```

Protocol Buffers generates a service stub that can function both synchronously and asynchronously. In this case we are making a synchronous remote call, so we call `Search()` with `NULL` as the first and fourth parameter:

```
// Make a synchronous remote call to server.
stub.Search(NULL, &request, &response, NULL);
```

When the call to `Search()` returns, the server response is in the `response` variable. Let's print it out:

```
// Print out response.
std::string strResponse;
TextFormat::PrintToString(response, &strResponse);
std::cout << "Received response:" << std::endl;
std::cout << strResponse << std::endl;
```

The RCFProto bindings for C#, Java and Python are very similar to the C++ bindings. Generally any RCFProto class or method available in C++, will be available in C#, Java and Python, with the same spelling, but possibly a different leading case.

The following sections provide line-by-line translations of the client and server example above, in C#, Java and Python.

C#

To use RCFProto in C#, you need to reference the `RCFProto_NET.dll` and `Google.ProtocolBuffers.dll` assemblies. RCFProto C# classes are defined in the `DeltaVSoft.RCFProto` namespace.

To generate C# code for your message and service types, use `ProtoGen.exe`:

```
ProtoGen.exe Tutorial.proto -service_generator_type=GENERIC
```

For more information on building C# applications, see Appendix - Building.

C# server

```
using DeltaVSoft.RCFProto;

// SearchService implementation.
class SearchServiceImpl : SearchService
{
    public override void Search(
        Google.ProtocolBuffers.IRpcController controller,
        SearchRequest request,
        System.Action<SearchResponse> done)
    {
        // Build the response.

        SearchResponse.Builder responseBuilder = SearchResponse.CreateBuilder();

        SearchResponse.Types.Result result = SearchResponse.Types.Result.CreateBuilder()
            .SetTitle("First result")
            .SetUrl("http://acme.org/")
            .Build();

        responseBuilder.AddResult(result);

        result = SearchResponse.Types.Result.CreateBuilder()
            .SetTitle("Second result")
            .SetUrl("http://acme.org/abc")
            .AddSnippets("Another snippet from acme.org.")
            .Build();

        responseBuilder.AddResult(result);

        SearchResponse response = responseBuilder.Build();

        // Send response back to the client.
        done(response);
    }

}

class TutorialServer
{
    static int Main(string[] args)
    {
        try
        {
            // Initialize RCFProto.
            RCFProto.Init();

            // Create server.
            RcfProtoServer server = new RcfProtoServer(new TcpEndpoint("0.0.0.0", 50001));

            // Bind Protobuf service.
            SearchServiceImpl searchServiceImpl = new SearchServiceImpl();
            server.BindService(searchServiceImpl);

            // Start the server.
            server.Start();

            // Wait for clients.
            System.Console.WriteLine("Press Enter to exit.");
            System.Console.ReadLine();
        }
        catch (System.Exception e)
        {
            System.Console.WriteLine("Exception: " + e.Message);
            return 1;
        }

        return 0;
    }
}
```

C# client

```
using DeltaVSoft.RCFProto;

class TutorialClient
{
    static int Main(string[] args)
    {
        try
        {
            // Initialize RCFProto.
            RCFProto.Init();

            // Create request object.
            SearchRequest request = SearchRequest.CreateBuilder()
                .SetQuery("something to search for")
                .SetResultPerPage(10)
                .SetPageNumber(0)
                .SetCorpus(SearchRequest.Types.Corpus.NEWS)
                .Build();

            // Create channel.
          RcfProtoChannel channel = new RcfProtoChannel( new TcpEndpoint("127.0.0.1", 50001) );

            // Create service stub.
            SearchService.Stub stub = new SearchService.Stub(channel);

            // Print out request.
            System.Console.WriteLine("Sending request:");
            System.Console.WriteLine(request.ToString());

            // Make a synchronous remote call to server.
            stub.Search(null, request, null);
            SearchResponse response = (SearchResponse)channel.GetResponse();

            // Print out response.
            System.Console.WriteLine("Received response:");
            System.Console.WriteLine(response.ToString());
        }
        catch (System.Exception e)
        {
            System.Console.WriteLine("Exception: " + e.Message);
            return 1;
        }

        return 0;
    }
}
```

Java

To use RCFProto in Java, you need to have `RCFProto.jar` and the relevant Google Protocol Buffers `.jar` file, on your class path. RCFProto Java classes are defined in the `com.deltavsoft.rcfproto` namespace.

To generate Java code for your message and service types, use `protoc`:

```
protoc Tutorial.proto --java_out=.
```

For more information on building Java applications, see Appendix - Building.

Java server

```
import com.deltavsoft.rcfproto.*;

public class TutorialServer
{
    // SearchService implementation.
    static class SearchServiceImpl extends Tutorial.SearchService
    {
        public void search(
            com.google.protobuf.RpcController controller,
            Tutorial.SearchRequest request,
            com.google.protobuf.RpcCallback<Tutorial.SearchResponse> done)
        {
            // Build the response.

          Tutorial.SearchResponse.Builder responseBuilder = Tutorial.SearchResponse.newBuilder();

            Tutorial.SearchResponse.Result result = Tutorial.SearchResponse.Result.newBuilder()
                .setTitle("First result")
                .setUrl("http://acme.org/")
                .build();

            responseBuilder.addResult(result);

            result = Tutorial.SearchResponse.Result.newBuilder()
                .setTitle("Second result")
                .setUrl("http://acme.org/abc")
                .addSnippets("Another snippet from acme.org.")
                .build();

            responseBuilder.addResult(result);

            Tutorial.SearchResponse response = responseBuilder.build();

            // Send response back to the client.
            done.run(response);
        }

    }

    public static void main(String[] args)
    {
        try
        {
            // Initialize RCFProto.
            RCFProto.init();

            // Create server.
            RcfProtoServer server = new RcfProtoServer(new TcpEndpoint("0.0.0.0", 50001));
            // Bind Protobuf service.
            SearchServiceImpl searchServiceImpl = new SearchServiceImpl();
            server.bindService(searchServiceImpl);

            // Start the server.
            server.start();

            // Wait for clients.
            System.out.println("Press Enter to exit.");
            java.io.BufferedReader reader = new java.io.BufferedReader(new java.io.InputStream↵
Reader(System.in));
            reader.readLine();
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Java client

```
import com.deltavsoft.rcfproto.*;

public class TutorialClient
{
    public static void main(String[] args)
    {
        try
        {
            // Initialize RCFProto.
            RCFProto.init();

            // Create request object.
            Tutorial.SearchRequest request = Tutorial.SearchRequest.newBuilder()
                .setQuery("something to search for")
                .setResultPerPage(10)
                .setPageNumber(0)
                .setCorpus(Tutorial.SearchRequest.Corpus.NEWS)
                .build();

            // Create channel.
           RcfProtoChannel channel = new RcfProtoChannel( new TcpEndpoint("127.0.0.1", 50001) );

            // Create service stub.
            Tutorial.SearchService.Stub stub = Tutorial.SearchService.newStub(channel);

            // Print out request.
            System.out.println("Sending request:");
            System.out.println(request.toString());

            // Make a synchronous remote call to server.
            stub.search(null, request, null);
            Tutorial.SearchResponse response = (Tutorial.SearchResponse)channel.getResponse();

            // Print out response.
            System.out.println("Received response:");
            System.out.println(response.toString());
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e.getMessage());
        }

    }
}
```

Python

To use RCFProto in Python, you need to install the RCFProto and Google Protocol Buffers Python modules. RCFProto Python classes are defined in the `deltavsoft.rcfproto` package.

To generate Python code for your message and service types, use `protoc`:

```
protoc Tutorial.proto --python_out=.
```

For more information on building Python applications, see Appendix - Building.

Python server

```
import sys

# Import RCFProto.
from deltavsoft.rcfproto import *

# Import protoc-generated module with message and service types.
import Tutorial_pb2

# SearchService implementation.
class SearchServiceImpl(Tutorial_pb2.SearchService) :

    def Search(self, controller, request, done):

        # Build the response.
        response = Demo_pb2.SearchResponse()

        result = response.result.add()
        result.title = 'First result'
        result.url = 'http://acme.org'

        result = response.result.add()
        result.title = 'Second result'
        result.url = 'http://acme.org/abc'
        result.snippets.append('Another snippet from acme.org.')

        # Send response back to the client.
        done(response)

# Initialize RCFProto.
init()

# Create server.
server = RcfProtoServer( TcpEndpoint("0.0.0.0", 50001) );

# Bind Protobuf service.
svc = SearchServiceImpl();
server.BindService(svc);

# Start the server.
server.Start();

# Wait for clients.
print('Press Enter to exit.');
sys.stdin.readline()
```

Python client

```
# Import RCFProto.
from deltavsoft.rcfproto import *

# Import protoc-generated module with message and service types.
import Tutorial_pb2

# Initialize RCFProto.
init();

# Create request object.
request = Tutorial_pb2.SearchRequest()
request.query = 'Something to search for'
request.result_per_page = 10
request.page_number = 0
request.corpus = Tutorial_pb2.SearchRequest.NEWS

# Create channel.
channel = RcfProtoChannel( TcpEndpoint("127.0.0.1", 50001) );

# Create service stub.
stub = Tutorial_pb2.SearchService_Stub(channel);

# Print out request.
print('Sending request:');
print( str(request) );

# Make a synchronous remote call to server.
stub.Search(None, request, None);
response = channel.GetResponse();

# Print out response.
print 'Received response:';
print( str(response) );
```

Client configuration

RCFProto client-side configuration is done using the `RcfProtoChannel` class.

For example, to set connect timeouts and remote call timeouts:

```
RCF::RcfProtoChannel channel( RCF::TcpEndpoint("127.0.0.1", 50001) );

// 5s connect timeout.
channel.setConnectTimeoutMs(5*1000);

// 10s remote call timeout.
channel.setRemoteCallTimeoutMs(10*1000);
```

You can configure transport protocols:

```
// Enable transport level compression.
channel.setEnableCompression(true);

// Configure NTLM transport protocol.
channel.setTransportProtocol(RCF::Tp_Ntlm);
```

For more information on client-side configuration, see the relevant reference documentation for `RcfProtoChannel`.

Server configuration

RCFProto server-side configuration is done using the `RcfProtoServer` and `RcfProtoSession` classes.

Configuring server endpoints:

```
// RcfProtoServer listening on single endpoint.
RCF::RcfProtoServer server( RCF::TcpEndpoint(50001) );
```

```
// RcfProtoServer listening on multiple endpoints.
RCF::RcfProtoServer server;
server.addEndpoint( RCF::TcpEndpoint(50001) );
server.addEndpoint( RCF::Win32NamedPipeEndpoint("MyPipe") );
```

Configuring server-side threading:

```
// Configure a thread pool with number of threads varying from 1 to 50.
RCF::ThreadPoolPtr threadPoolPtr( new RCF::ThreadPool(1, 50) );
server.setThreadPool(threadPoolPtr);
```

Configuring which transport protocols clients are required to use:

```
std::vector<RCF::TransportProtocol> protocols;
protocols.push_back(RCF::Tp_Ntlm);
protocols.push_back(RCF::Tp_Kerberos);
server.setSupportedTransportProtocols(protocols);
```

From the server-side implementation of a remote call, you have access to a `RcfProtoSession` object, representing the current client session. Here is how you would obtain the current `RcfProtoSession` in C++:

```
// Search() method implementation.
void Search(
    RpcController *                 controller,
    const SearchRequest *           request,
    SearchResponse *                response,
    Closure *                       done)
{

    RCF::RcfProtoController * rcfController =
        static_cast<RCF::RcfProtoController *>(controller);

    RCF::RcfProtoSession * pSession = rcfController->getSession();

    // Fill in the response.
    // ...

    // Send it back to the client.
    done->Run();
}
```

Here is how you obtain the current `RcfProtoSession` in C#:

```
// Search() method implemenation.
public override void Search(
    Google.ProtocolBuffers.IRpcController controller,
    SearchRequest request,
    System.Action<SearchResponse> done)
{
    // Obtain RcfProtoSession for current client connection.
    RcfProtoController rcfController = (RcfProtoController)controller;
    RcfProtoSession session = rcfController.GetSession();

    // Create response.
    SearchResponse response = SearchResponse.CreateBuilder().Build();

    // Send it back to the client.
    done(response);
}
```

Here is how you obtain the current `RcfProtoSession` in Java:

```
public void search(
    com.google.protobuf.RpcController controller,
    Tutorial.SearchRequest request,
    com.google.protobuf.RpcCallback<Tutorial.SearchResponse> done)
{
    // Obtain RcfProtoSession for current client connection.
    RcfProtoController rcfController = (RcfProtoController)controller;
    RcfProtoSession session = rcfController.getSession();

    // Create response.
    Tutorial.SearchResponse response = Tutorial.SearchResponse.newBuilder().build();

    // Send it back to the client.
    done.run(response);
}
```

Here is how you obtain the current `RcfProtoSession` in Python:

```
# Search() method implementation.
def Search(self, controller, request, done):

    # Obtain RcfProtoSession for current client connection.
    session = controller.GetSession();

    # Create response.
    response = Tutorial_pb2.SearchResponse()

    # Send it back to the client.
    done(response)
```

From `RcfProtoSession`, you can retrieve various details about the client.

You can determine the transport type and transport protocol in use:

```
RCF::TransportType transportType = pSession->getTransportType();
RCF::TransportProtocol protocol = pSession->getTransportProtocol();
```

If the transport protocol is NTLM or Kerberos, or if the transport is a Win32 named pipe, you can determine the Windows username of the client:

```
if (    (protocol == RCF::Tp_Ntlm || protocol == RCF::Tp_Kerberos)
    ||  transportType == RCF::Tt_Win32NamedPipe)
{
    std::string clientUsername = pSession->getClientUsername();
}
```

For more information on server-side configuration, see the relevant reference documentation for `RcfProtoServer` and `RcfProtoSession`.

Asynchronous remote calls

When you call remote methods on a client stub, RCFProto's default behavior is to make a synchronous call. This means that the method does not return until a response is received from the server, or an error or timeout occurs. Synchronous calls are conceptually simple and easy to program. However, they also have the drawback that during the call, your application does not have control of the calling thread. In some situations this may not be acceptable, in which case you will need to use asynchronous remote calls instead.

When making an asynchronous call, you need to pass in a completion handler that RCFProto will call when the call completes. The call is initiated on your own application thread, but the thread does not block and returns immediately to the application. Meanwhile the call completes on a background RCFProto thread, and when it does, your completion callback is called.

Here is an example of an asynchronous call in C++:

```
// Remote call completion handler.
void onCallCompletion(RCF::RcfProtoController * pController, SearchRequest * pRequest, Sear↵
chResponse * pResponse)
{
    // Check for any errors first.
    if (pController->Failed())
    {
        std::cout << "Error: " << pController->ErrorText() << std::endl;
    }
    else
    {
        std::string strResponse;
        TextFormat::PrintToString(*pResponse, &strResponse);
        std::cout << "Received response:" << std::endl;
        std::cout << strResponse << std::endl;
    }
}
```

```
// Executing an asynchronous remote call.
channel.setAsynchronousRpcMode(true);
RCF::RcfProtoController controller;
RCF::BoostBindClosure closure( boost::bind(&onCallCompletion, &controller, &request, &response) );
stub.Search(&controller, &request, &response, &closure);
```

`RcfProtoController` is used to monitor the asynchronous remote call while it is in progress:

```
// RcfProtoController is used to check on the progress of the asynchronous remote call, or to ↵
cancel it.

// Poll for completion...
while (!controller.Completed())
{
    Sleep(500);
}

// ... or cancel the remote call.
controller.StartCancel();

// After the remote call completes, check if it failed.
if (controller.Failed())
{
    std::string errorMsg = controller.ErrorText();
}
```

Here is an example of an asynchronous remote call in C#:

```
// Remote call completion handler.
static void onCallCompletion(RcfProtoController controller, SearchRequest request, SearchResponse ↵
response)
{
    // Check for any errors first.
    if (controller.Failed)
    {
        System.Console.WriteLine( "Error: " + controller.ErrorText );
    }
    else
    {
        System.Console.WriteLine( "Received response:" );
        System.Console.WriteLine( response.ToString() );
    }
}
```

```
// Executing an asynchronous remote call.
channel.SetAsynchronousRpcMode(true);
RcfProtoController controller = new RcfProtoController();
System.Action<SearchResponse> done = delegate(SearchResponse response)
{
    onCallCompletion(controller, request, response);
};
stub.Search(controller, request, done);
```

```
// RcfProtoController is used to check on the progress of the asynchronous remote call, or to ↵
cancel it.

// Poll for completion...
while ( !controller.Completed )
{
    System.Threading.Thread.Sleep(500);
}

// ... or cancel the remote call.
controller.StartCancel();

// After the remote call completes, check if it failed.
if (controller.Failed)
{
    string errorMsg = controller.ErrorText;
}
```

Here is an example of an asynchronous remote call in Java:

```
// Remote call completion handler.
static class OnRemoteCallCompleted implements RpcCallback<Tutorial.SearchResponse>
{
    OnRemoteCallCompleted(RcfProtoController controller)
    {
        mController = controller;
    }

    public void run(Tutorial.SearchResponse response)
    {
        if ( mController.failed() )
        {
            System.out.println( "Error: " + mController.errorText() );
        }
        else
        {
            System.out.println("Received response:");
            System.out.println(response.toString());
        }
    }

    private RcfProtoController mController;
}
```

```
// Executing an asynchronous remote call.
channel.setAsynchronousRpcMode(true);
RcfProtoController controller = new RcfProtoController();
OnRemoteCallCompleted done = new OnRemoteCallCompleted(controller);
stub.search(controller, request, done);
```

```
// RcfProtoController is used to check on the progress of the asynchronous remote call, or to ↲
cancel it.

// Poll for completion...
while ( !controller.completed() )
{
    Thread.sleep(500);
}

// ... or cancel the remote call.
controller.startCancel();

// After the remote call completes, check if it failed.
if (controller.failed())
{
    String errorMsg = controller.errorText();
}
```

Here is an example of an asynchronous remote call in Python:

```
# Remote call completion handler.
def OnRemoteCallCompleted(controller, response):
    if controller.Failed():
        print "Error: " + controller.ErrorText()
    else:
        print( 'Received response:' )
        print( str(response) )

# Executing an asynchronous remote call.
channel.SetAsynchronousRpcMode(True)
controller = RcfProtoController()
stub.Search(controller, request, lambda response: OnRemoteCallCompleted(controller, response))
```

```
# RcfProtoController is used to check on the progress of the asynchronous remote call, or to can↲
cel it.

# Poll for completion...
while not controller.Completed():
    time.sleep(.5);


# ... or cancel the remote call.
controller.StartCancel();

# After the remote call completes, check if it failed.
if controller.Failed():
    errorMsg = controller.ErrorText()
```

Transports

This section provides an overview of the various transports that RCFProto supports.

TCP

`TcpEndpoint` represents a TCP endpoint. It is important to remember that if you construct a `TcpEndpoint` without passing in an IP address, the loopback address `127.0.0.1` is assumed. For servers, this means that the server will only be accessible from the local machine. To create a TCP endpoint that is accessible from across the network, use `0.0.0.0` (for IPV4), or `::0` (for IPV6).

Here are some examples of servers and clients using `TcpEndpoint`.

```
// Listen on port 50001 on all available IPv4 network interfaces.
RCF::RcfProtoServer server( RCF::TcpEndpoint("0.0.0.0", 50001) );
```

```
// Listen on port 50001 on all available IPv6 network interfaces.
RCF::RcfProtoServer server( RCF::TcpEndpoint("::0", 50001) );
```

```
// Listen on loopback IPv4 network interface.
RCF::RcfProtoServer server( RCF::TcpEndpoint("127.0.0.1", 50001) );
```

```
// Listen on loopback IPv4 network interface.
RCF::RcfProtoServer server( RCF::TcpEndpoint(50001) );
```

```
// Listen on dynamically assigned port on loopback IPv4 network interface.
RCF::RcfProtoServer server( RCF::TcpEndpoint(0) );
server.start();
int port = server.getIpServerTransport().getPort();
```

```
// Connect to server on port 50001 of server.corp.com .
RCF::RcfProtoChannel channel( RCF::TcpEndpoint("server.corp.com", 50001) );
```

HTTP/HTTPS

RCFProto provides HTTP and HTTPS based transports, in case you need to tunnel through an HTTP proxy to reach your server.

Here are some examples of servers and clients using `HttpEndpoint` and `HttpsEndpoint`.

```
// Listen on port 80 of all available IPv4 network interfaces.
RCF::RcfProtoServer server( RCF::HttpEndpoint("0.0.0.0", 80) );
```

```
// Connect to port 80 of server.corp.com.
RCF::RcfProtoChannel channel( RCF::HttpEndpoint("server.corp.com", 80) );

// Configure HTTP proxy.
channel.setHttpProxy("proxy.corp.com");
channel.setHttpProxyPort(8080);
```

```
// Listen on port 443 of all available IPv4 network interfaces.
RCF::RcfProtoServer server( RCF::HttpsEndpoint("0.0.0.0", 443) );

// Configure certificate for OpenSSL-based SSL.
RCF::CertificatePtr certPtr( new RCF::PemCertificate("/path/to/certificate.pem", "password") );
server.setCertificate(certPtr);
```

```
// Connect to port 443 of server.corp.com.
RCF::RcfProtoChannel channel( RCF::HttpEndpoint("server.corp.com", 443) );

// Configure HTTP proxy.
channel.setHttpProxy("proxy.corp.com");
channel.setHttpProxyPort(8080);
```

Certificate configuration for HTTPS is done in the same way as for SSL transports (see further down).

Named pipes

For communication between processes on a single machine, RCF provides Win32 named pipe transport (for Windows), and UNIX local domain socket transports (for UNIX):

```
// Listen on Win32 named pipe named "MyPipe".
RCF::RcfProtoServer server( RCF::Win32NamedPipeEndpoint("MyPipe") );
```

```
// Connect to Win32 named pipe named "MyPipe".
RCF::RcfProtoChannel channel( RCF::Win32NamedPipeEndpoint("MyPipe") );
```

```
// Listen on UNIX local socket named "MyLocalSocket".
RCF::RcfProtoServer server( RCF::UnixLocalEndpoint("MyLocalSocket") );
```

```
// Connect to UNIX local socket named "MyLocalSocket".
RCF::RcfProtoChannel channel( RCF::UnixLocalEndpoint("MyLocalSocket") );
```

Transport protocols

RCFProto supports several different transport protocols, for purposes of compression, encryption and authentication.

Compression

Compression can be enabled for a `RCFProtoChannel`. The compression algorithm is stateful and compresses data across multiple calls for better compression.

```
channel.setEnableCompression(true);
```

RCFProto uses the zlib library to implement compression.

NTLM

On Windows, RCFProto supports NTLM encryption and authentication:

```
channel.setTransportProtocol(RCF::Tp_Ntlm);

// Leave username and password blank, to connect using the logged on users credentials:
channel.connect();

// Or specify username and password explicitly:
channel.disconnect();
channel.setUsername("SomeDomain\\SomeUser");
channel.setPassword("SomePassword");
channel.connect();
```

When the NTLM transport protocol is in use, server-side code is able to securely determine the Windows username of the client:

```
// When using NTLM or Kerberos transport protocols, the client username is available to server-
side code.
RCF::RcfProtoSession * session = controller.getSession();
std::string clientUsername = session->getClientUsername();
```

Kerberos

RCFProto also supports Kerberos encryption and authentication:

```
channel.setTransportProtocol(RCF::Tp_Kerberos);

// Kerberos requires the client to specify the SPN (Server Principal Name) of the server.
// The Kerberos protocol will verify that the server is running as this account.
channel.setKerberosSpn("SomeDomain\\ServerAccount");

// As for NTML, username and password are optional. If not specified, the current users creden↲
tials are used.
channel.setUsername("SomeDomain\\SomeUser");
channel.setPassword("SomePassword");
```

When the Kerberos transport protocol is in use, server-side code is able to securely determine the Windows username of the client:

```
// When using NTLM or Kerberos transport protocols, the client username is available to server-
side code.
RCF::RcfProtoSession * session = controller.getSession();
std::string clientUsername = session->getClientUsername();
```

SSL

RCFProto provides two separate SSL implementations. One is based on the Schannel SSPI package in Windows, and the other is based on OpenSSL.

On Windows, it's a good idea to explicitly specify which SSL implementation you want to use. This can be done globally:

```
RCF::setDefaultSslImplementation(RCF::Si_Schannel);
```

, or for individual servers and clients:

```
RCF::RcfProtoServer server( RCF::TcpEndpoint(50001) );
server.setSslImplementation(RCF::Si_Schannel);
```

```
RCF::RcfProtoChannel channel( RCF::TcpEndpoint(50001) );
channel.setSslImplementation(RCF::Si_Schannel);
```

Certificate validation can be done through custom validation callbacks, or through OpenSSL-specific or Schannel-specific means.

Here is an example of a custom validation callback:

```
// Custom certificate validation callback.
bool onValidateCertificate(RCF::Certificate * pCert)
{
    RCF::Win32Certificate * pWin32Cert = dynamic_cast<RCF::Win32Certificate *>(pCert);
    if (pWin32Cert)
    {
        // Schannel based certificate.
       std::cout << "Server certificate details: " << RCF::toAstring(pWin32Cert->getCertificate↵
Name()) << std::endl;
        std::cout << "Server certificate issuer details: " << RCF::toAstring(pWin32Cert->getIs↵
suerName()) << std::endl;
    }

    RCF::X509Certificate * pX509Cert = dynamic_cast<RCF::X509Certificate *>(pCert);
    if (pX509Cert)
    {
        // OpenSSL based certificate.
        std::cout << "Server certificate details: " << pX509Cert->getCertificate↵
Name() << std::endl;
        std::cout << "Server certificate issuer details: " << pX509Cert->getIssuer↵
Name() << std::endl;
    }

    // Return false or throw an exception to indicate that you don't consider the certificate ↵
valid.
    //return false;
    //throw std::runtime_error("Invalid server certificate. Reason: ???");

    // Return true to indicate that you consider the certificate valid.
    return true;
}
```

A number of methods are available on `RcfProtoChannel` and `RcfProtoServer` to configure certificate and certificate validation. Here is an example of certificate configuration using OpenSSL:

Server-side OpenSSL configuration:

```
// Server-side configuration.
RCF::RcfProtoServer server( RCF::TcpEndpoint(50001) );

server.setSslImplementation(RCF::Si_OpenSsl);

RCF::CertificatePtr certPtr( new RCF::PemCertificate(
    "/path/to/certificate.pem",
    "password") );

// Set the certificate that this server will present to clients.
server.setCertificate(certPtr);
```

Client-side OpenSSL configuration:

```
// Client-side configuration.
RCF::RcfProtoChannel channel( RCF::TcpEndpoint(50001) );
channel.setSslImplementation(RCF::Si_OpenSsl);

// Certificate validation - provide certificate authority certificate. OpenSSL will verify
// the server certificate against the provided CA certificate.
RCF::CertificatePtr caCertPtr( new RCF::PemCertificate("/path/to/caCertificate.pem", "") );
channel.setCaCertificate(caCertPtr);

// Certificate validation - custom certificate validation callback.
channel.setCertificateValidationCallback(onValidateCertificate);
```

Here is an example of certificate configuration using Schannel:

Server-side Schannel configuration:

```
// Server-side configuration.
RCF::RcfProtoServer server( RCF::TcpEndpoint(50001) );

server.setSslImplementation(RCF::Si_Schannel);

RCF::CertificatePtr certPtr( new RCF::PfxCertificate(
    "/path/to/certificate.pfx",
    "password",
    "certificate name") );

// Set the certificate that this server will present to clients.
server.setCertificate(certPtr);
```

Client-side Schannel configuration:

```
// Client-side configuration.
RCF::RcfProtoChannel channel( RCF::TcpEndpoint(50001) );
channel.setSslImplementation(RCF::Si_Schannel);

// Certificate validation - use default Schannel validation. Schannel will check that
// the server certificate matches the provided name, and that a corresponding certificate
// authority is present in the Windows "Trusted Root Certification Authorities"
// certificate store.
channel.setEnableSchannelCertificateValidation(RCF::toTstring("certificate name"));

// Certificate validation - custom certificate validation callback.
channel.setCertificateValidationCallback(onValidateCertificate);
```

Appendix - Building RCFProto

This section provides details on building RCFProto and RCFProto applications, in C++, C#, Java and Python.

Building RCFProto from source is generally only necessary if you are developing on non-Windows platforms. If you are developing on a Windows platform, you can use the RCFProto Windows binary distribution instead, which comes with pre-packaged C#, Java and Python binaries for x86 and x64 architectures.

Building RCFProto bindings (C#, Java and Python)

Dependencies

To build RCFProto from source, you will need the following dependencies:

- C++ compiler - any reasonably recent compiler will do (gcc 4.x, Visual Studio 2008 or newer, or similar)
- cmake (2.8 or later)
- Boost (1.35.0 or later)
- zlib
- OpenSSL

Additional dependencies are needed, depending on which language binding you are building.

For C# bindings:

- .NET SDK
- Protocol Buffers C# library (Google.ProtocolBuffers.dll). This is a C# port of Google's Protocol Buffer implementation.

For Java bindings:

- JDK
- Protocol Buffers Java library. The Protocol Buffers distribution contains relevant build instructions.

For Python bindings:

- Python interpreter and development libraries.
- Protocol Buffers Python library. The Protocol Buffers distribution contains relevant build instructions.

Running cmake

RCFProto uses a `cmake`-based build system. `cmake` is used to generate a native build system (either Unix makefiles or Visual C++ `nmake` makefiles). The native build system is then used to build the executables.

To build on Unix-like systems, use `cmake` with the `"Unix Makefiles"` generator. From the root directory of the RCFProto source distribution:

```
mkdir bin
cd bin
cmake -G "Unix Makefiles" ..
```

This will generate a makefile, and to build RCFProto you then run `make`:

```
make
```

On Windows systems, use `cmake` with the Visual C++ `nmake` generator:

```
mkdir bin
cd bin
cmake -G "NMake Makefiles" ..
```

This will generate a `nmake` makefile, which is then built by running `nmake` from a Visual C++ command prompt:

```
nmake
```

When `cmake` runs, it attempts to automatically locate all dependencies and prints a summary of the dependencies it found. For example, a build on Linux may produce this output:

```
...
************************************************
*** RCFProto build: Third party dependency summary ***
RCF include path: /home/user/Development/RCF/RCFProto/cpp/src/RCF/include
Boost include path: /home/user/Development/boost_1_44_0
Protobuf include path: /usr/include
Protobuf libs: /usr/lib/libprotobuf.so
Protobuf debug lib:
Protobuf Java class path: /usr/share/java/protobuf.jar
JNI include paths: /usr/lib/jvm/java-6-openjdk/include/usr/lib/jvm/java-6-openjdk/in↲
clude/usr/lib/jvm/java-6-openjdk/include
Java compiler: /usr/bin/javac
Java runtime: /usr/bin/java
Java archiver: /usr/bin/jar
Python include paths: /usr/include/python2.7
Python libs: /usr/lib/libpython2.7.so
ZLIB include path: /usr/include
OpenSSL include path: /usr/include
************************************************
...
```

If you have a dependency installed to a custom location that `cmake` is unable to locate automatically, you can set the relevant `cmake` variable:

```
# Custom path to Protocol Buffers Java library (protobuf.jar)
cmake -G "Unix Makefiles" -DPROTOBUF_JAVA_CLASS_PATH=path/to/protobuf.jar ..
```

Note that if you are building e.g. Python bindings, you do not need Java-related dependencies (JDK, Protobuf Java class path), and vice versa. The RCFProto build will only build those bindings for which dependencies are present.

Binaries

C#

The C# build will copy the built binaries to the `/csharp/bin` folder in the source distribution.

Here is a list of the binaries on a Windows system:

- `csharp/bin/Google.ProtocolBuffers.dll` - Protocol Buffers C# assembly.
- `csharp/bin/protoc.exe` - required for `ProtoGen.exe`.

- `csharp/bin/ProtoGen.exe` - Protocol Buffers C# code generator.
- `csharp/bin/RCFProto_NET.dll` - RCFProto .NET assembly, built with AnyCPU architecture.
- `csharp/bin/x86/RCFProto_NET_impl.dll` - RCFProto implementation DLL for x86.
- `csharp/bin/x64/RCFProto_NET_impl.dll` - RCFProto implementation DLL for x64.

Java

The Java build will copy the built binaries to the `/java/bin` folder in the source distribution.

Here is a list of the binaries on a Windows system:

- `java/bin/Google.ProtocolBuffers.jar` - Protocol Buffers Java library.
- `java/bin/protoc.exe` - Protocol Buffers Java code generator.
- `java/bin/RCFProto.jar` - RCFProto Java library.
- `java/bin/x86/RCFProto_Java_impl.dll` - RCFProto implementation DLL for x86.
- `java/bin/x64/RCFProto_Java_impl.dll` - RCFProto implementation DLL for x64.

Here is a list of the binaries on a Unix x86 system:

- `java/bin/protobuf.jar` - Protocol Buffers Java library.
- `java/bin/protoc` - Protocol Buffers Java code generator.
- `java/bin/RCFProto.jar` - RCFProto Java library.
- `java/bin/x86/libRCFProto_Java_impl.so` - RCFProto implementation library for x86.

Python

The Python build will copy the built binaries to the `/python/bin` folder in the source distribution.

Here is a list of the binaries on a Windows system:

- `python/bin/protoc.exe` - Protocol Buffers Python code generator.
- `python/bin/x86/_rcfproto.pyd` - RCFProto Python extension DLL for x86.
- `python/bin/x64/_rcfproto.pyd` - RCFProto Python extension DLL for x64.

Here is a list of the binaries on a Unix x86 system:

- `python/bin/protoc` - Protocol Buffers Python code generator.
- `python/bin/x86/_rcfproto.so` - RCFProto Python extension library for x86.

Once the RCFProto Python bindings are built, you will need to install RCFProto into your Python interpreter. To do so, go to the `python/src` folder and run `setup.py`:

```
python setup.py install
```

At this point you will be able to open a Python interpreter and instantiate RCFProto classes:

```
from deltavsoft.rcfproto import *
init()
server = RcfProtoServer( TcpEndpoint('0.0.0.0', 50001 ) )
server.Start()
```

Building RCFProto C++ applications

Building

To build C++ applications using RCFProto, you only need to compile `RCF.cpp` and `RCFProto.cpp` into your application. If you want to build RCFProto as a DLL or shared library, you will need to define `RCF_BUILD_DLL`.

There are several other defines that can be used to configure RCFProto:

- `RCF_USE_ZLIB` - Build RCFProto with compression support.
- `RCF_USE_OPENSSL` - Build RCFProto with OpenSSL-based SSL support.
- `RCF_USE_IPV6` - Build RCFProto with IPV6 support.

You will also need to link to the Protocol Buffers C++ library. The Protocol Buffers package contains instructions on building this library (`libprotobuf.dll` / `libprotobuf.so`) .

To generate Protocol Buffers C++ code for your application, use `protoc` with the `--cpp_out` option:

```
protoc Tutorial.proto --cpp_out=.
```

Runtime requirements

The RCFProto C++ bindings have no runtime requirements. Your compiler may have its own runtime requirements (for example, Visual C++-based builds with dynamic runtime linking will require the Visual C++ compiler runtimes to be available).

Building RCFProto C# applications

Building

The stock Protocol Buffers implementation from Google does not provide a C# implementation. However, a third party implementation, protobuf-csharp-port, is available which provides a C# implementation matching fairly closely with the Java implementation from Google. RCFProto's C# support is based on this Protocol Buffers implementation.

To build C# applications using RCFProto, you will need to reference the `RCFProto_NET.dll` assembly, along with the `Google.ProtocolBuffers.dll` assembly.

To generate Protocol Buffers C# code for your application, use `ProtoGen.exe`, with the `-service_generator_type=GENERIC` option:

```
ProtoGen.exe Tutorial.proto -service_generator_type=GENERIC -output_directory=.
```

Without the `-service_generator_type=GENERIC` option, `ProtoGen.exe` will only generate code for message types, and not for RPC services.

Runtime requirements

At runtime, your C# application will load `RCFProto_NET.dll` . When your application calls `RCFProto.Init()` to initialize RCFProto, RCFProto will then load a native DLL (`RCFProto_NET_impl.dll`), which contains the RCFProto implementation.

This means that at runtime, `RCFProto_NET_impl.dll` must be present in a location searched by the Windows DLL loader. Typically this means that `RCFProto_NET_impl.dll` should be in the same location as the loading executable.

In some cases this may not be possible (for example if you are writing plugins for a hosting application such as IIS). If so, you will need to call the `RCFProto.SetNativeDllPath()` method, prior to `RCFProto.Init()`, to explicitly specify the full path from which `RCFProto_NET_impl.dll` should be loaded.

`RCFProto_NET_impl.dll` is a native DLL and is provided in both x86 and x64 versions. You will need to make sure you select the right version for your application, or the DLL will not be able to be loaded.

Building RCFProto Java applications

Building

Java applications need to include the paths to `RCFProto.jar` and Protocol Buffers (e.g. `Protocol.Buffers.jar`), as part of the class path.

To generate Protocol Buffers Java code, use `protoc` with the `--java_out` option:

```
protoc Tutorial.proto --java_out=.
```

Runtime requirements

Java applications need to include the paths to `RCFProto.jar` and Protocol Buffers (e.g. `Protocol.Buffers.jar`), as part of the class path.

When `RCFProto.init()` is called, RCFProto will attempt to load the native RCFProto JNI library (`RCFProto_Java_impl.dll` on Windows, or `libRCFProto_Java_impl.so` on Unix). For this to succeed, the `java.lib.path` variable needs to be configured with the path to the directory where the native JNI library is to be loaded from.

`java.lib.path` is normally set when the JVM is started. If you find yourself needing to modify `java.lib.path` after the JVM has started, the following trick may be useful:

```
// This hack enables us to modify java.lib.path at runtime.
// http://blog.cedarsoft.com/2010/11/setting-java-library-path-programmatically/

String javaLibPath = System.getProperty( "java.library.path" );

// Modify javaLibPath.
// ...

// Set it back again.
System.setProperty( "java.library.path", javaLibPath );
Field fieldSysPath = ClassLoader.class.getDeclaredField( "sys_paths" );
fieldSysPath.setAccessible( true );
fieldSysPath.set( null, null );
```

Building RCFProto Python applications

Building

Python applications need to import the `deltavsoft.rcfproto` package.

To generate Protocol Buffers Python code, use `protoc` with the `--python_out` option:

```
protoc Tutorial.proto --python_out=.
```

Runtime requirements

The RCFProto and Protocol Buffers modules need to be installed into your Python interpreter.

Appendix - Logging

RCFProto provides a configurable logging subsystem with 4 log levels and several log output options.

Logging is disabled by default. To enable it, call `enableLogging()`:

```
RCF::enableLogging( RCF::LogToFile("path/to/rcfproto.log"), 4, "");
```

In this case we are setting the log level to 4 (highest), and directing log output to the given log file.

To disable logging, call `disableLogging()`:

```
RCF::disableLogging();
```

Logging can also be directed to the debug window of Visual Studio (`LogToDebugWindow`), the Windows event log (`LogToEventLog`), or standard output (`LogToStdout`). For more information, see the reference documentation for these classes.

Appendix - Release Notes

RCFProto 1.0.0.3 - 2014-09-06

- Add `RcfProtoSession::getRcfSession()` to allow access to `RcfSession` from RCFProto C++ servers.
- Fix RCFProto server-side memory leak in C#, Java and Python bindings.
- Downloads:
  - RCFProto-src-1.0.0.3.zip
  - RCFProto-win32-1.0.0.3.zip

RCFProto 1.0.0.1 - 2014-04-24

- Fix for C++ client not properly deserializing response messages from server.
- Downloads:
  - RCFProto-src-1.0.0.1.zip
  - RCFProto-win32-1.0.0.1.zip

RCFProto 0.9.0.6 - 2013-08-21

- Fix UNIX local domain socket names in demos, so they match across all languages.
- Fix memory leak in C++ bindings.
- Add RCFProto User Guide. The User Guide is available online, as well as being part of the distributions.
- Downloads:
  - RCFProto-src-0.9.0.6.zip
  - RCFProto-win32-0.9.0.6.zip

RCFProto 0.9.0.5 - 2013-07-17

- Regenerated demo SSL certificates. Certificates in 0.9.0.4 had expired after release.
- Downloads:
  - RCFProto-src-0.9.0.5.zip
  - RCFProto-win32-0.9.0.5.zip

RCFProto 0.9.0.4 - 2013-06-01

- First release.
- Downloads:
  - RCFProto-src-0.9.0.4.zip
  - RCFProto-win32-0.9.0.4.zip

Appendix - FAQ