RCF User Guide

Table of Contents

Version

This documentation applies to **RCF 2.0**.

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

Introduction

What is RCF?

RCF (Remote Call Framework) is a cross-platform interprocess communication framework for C++.

Unlike other communication frameworks, RCF doesn't use a separate IDL (Interface Definition Language). RCF interfaces are defined directly in C++, and serialization for user-defined data types likewise is implemented in C++. Instead of a separate IDL compiler tool, RCF uses the C++ compiler to generate client and server stubs.

RCF provides a broad range of interprocess communication features:

- **Oneway** and **twoway** messaging.
- **Batched oneway** messaging.
- **Publish/subscribe** style messaging.
- **Multicast** and **broadcast** messaging over UDP.
- **Asynchronous remote calls**.
- **Server-to-client** callback connections.
- **Multiple transports** (TCP, UDP, Windows named pipes and UNIX local domain sockets).
- **Tunneling** over HTTP and HTTPS.
- **Compression** (zlib) and **encryption** (Kerberos, NTLM, Schannel, and OpenSSL).
- Supports **IPv6**.
- Built in **serialization** framework.
- Built in **file transfer** capabilities.
- Robust **versioning** support.
- Support for Boost.Serialization.
- Support for Protocol Buffers.
- Support for JSON-RPC.
- **Portability** across a wide range of compilers, platforms and operating systems.
- **Scalability** acoss a wide range of applications, from simple parent-child IPC, right up to large scale distributed systems.
- **Efficiency**, with **zero-copy, zero-heap allocation** on critical paths, on both server and client.
- **No dependencies**, apart from Boost header files (1.35.0 or later). zlib and OpenSSL are optional.

To start learning about programming with RCF, go straight to the Tutorial.

Why should I use RCF?

You should consider using RCF if you are writing C++ components that need some form of interprocess communication. For communication between native C++ components, there is little benefit in using XML and XML schemas to describe messages, as native serialization formats are more descriptive and efficient. There are also few benefits to separately compiled IDL files. RCF's approach of describing interfaces in C++ code makes for a simpler build and more flexible development.

RCF is written in 100% standard C++, and is both portable and efficient. Basing your communication layer on RCF gives you portability across a wide range of compilers, operating systems, and platforms, and the ability to pick and choose from a wide range of transport mechanisms, threading models, and messaging paradigms.

RCF has been designed as a real world tool for real world applications. RCF has been in large-scale commercial use since 2006, and today powers networked applications around the world. Some of the scenarios RCF has been used in include:

- Client-server systems, across LAN's or WAN's. From industrial process control, to replacing DCOM in distributed desktop applications, to cross platform communication.
- Back end server components, typically in a homogeneous LAN environment.
- Communication between Windows services and their monitoring applications.
- Parent-child process communication.
- Generic local IPC, replacing COM.

Tutorial

Getting started

Hello World

Following programming language traditions, let's start with a "Hello World" example. We'll write a client that sends strings to a server, and a server that prints the strings to standard output.

Here is the server:

```
#include <RCF/RCF.hpp>

#include <iostream>


RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
RCF_END(I_HelloWorld)

class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }
};


int main()
{
    RCF::RcfInitDeinit rcfInit;

    HelloWorldImpl helloWorld;
    RCF::RcfServer server( RCF::TcpEndpoint(50001) );
    server.bind<I_HelloWorld>(helloWorld);
    server.start();

    std::cout << "Press Enter to exit..." << std::endl;
    std::cin.get();

    return 0;
}
```

And the client:

```
#include <iostream>

#include <RCF/RCF.hpp>


RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
RCF_END(I_HelloWorld)

int main()
{
    RCF::RcfInitDeinit rcfInit;
    std::cout << "Calling the I_HelloWorld Print() method." << std::endl;
    RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
    client.Print("Hello World");
    return 0;
}
```

The transport used is a TCP connection, with the server listening on port 50001 of the localhost (`127.0.0.1`) interface.

Before we can run this code, we need to build it. To build RCF, you simply need to compile the `src/RCF/RCF.cpp` source file into your application. RCF requires the Boost library to be available, so you will need to download Boost. Any version from 1.35.0 onwards will do. You do not need to build any Boost libraries, as RCF will in its default configuration only use header files from Boost.

Here is how you would build the server code above, with two common compiler toolsets.

How to build - Visual Studio 2010

- In the Visual Studio IDE, select `New Project -> Visual C++ -> Win32 -> Win32 Console Application`
- In the `Application Settings` dialog, check the `Empty Project` checkbox.
- In the `Project Properties` dialog for the new project, select `C/C++ -> General -> Additional Include Directories`.
- Add the include paths for Boost and RCF, e.g. `C:\boost_1_49_0` and `C:\RCF\include`.
- Add a file `Server.cpp` to the project, and copy-paste the code above into it.
- Add `RCF\src\RCF\RCF.cpp` to the project.
- Select `Build -> Build Solution`.

How to build - gcc

- Create a file `Server.cpp` and copy-paste the code above into it.
- From the same directory, run the following command:

```
g++ Server.cpp /path/to/RCF/src/RCF/RCF.cpp -I/path/to/boost_1_49_0 -I/path/to/RCF/include -lp↵
thread -ldl -oServer
```

, replacing `/path/to/boost_1_49_0` and `/path/to/RCF` with the actual paths on your system.

Running the server and client

Let's start the server:

```
c:\Projects\RcfSample\Debug>Server.exe
Press Enter to exit...
```

, and then the client:

```
c:\Projects\RcfSample\Debug>Client.exe
Calling the I_HelloWorld Print() method.
c:\Projects\RcfSample\Debug>
```

In the server window, you should now see:

```
c:\Projects\RcfSample\Debug>Server.exe
Press Enter to exit...
I_HelloWorld service: Hello World
```

To simplify the rest of the tutorial, we'll rewrite our Hello World example so that the client and server run in a single process:

```
#include <iostream>

#include <RCF/RCF.hpp>


RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
RCF_END(I_HelloWorld)

class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }
};


int main()
{
    RCF::RcfInitDeinit rcfInit;

    HelloWorldImpl helloWorld;
    RCF::RcfServer server( RCF::TcpEndpoint(50001) );

    server.bind<I_HelloWorld>(helloWorld);

    server.start();

    std::cout << "Calling the I_HelloWorld Print() method." << std::endl;

    RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
    client.Print("Hello World");

    return 0;
}
```

Running this program yields the output:

```
Calling the I_HelloWorld Print() method.
I_HelloWorld service: Hello World
```

The rest of the tutorial will build on this example, to illustrate some of the fundamental features of RCF.

Interfaces and implementations

RCF allows you to define remote interfaces, directly in code. In the example above, we defined the `I_HelloWorld` interface:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
RCF_END(I_HelloWorld)
```

`RCF_BEGIN()`, `RCF_METHOD_xx()`, and `RCF_END()` are macros that are used to generate client and server stubs for remote calls.

- `RCF_BEGIN()` starts the interface definition, and defines the compile time identifier of the interface (`I_HelloWorld`), and the runtime identifier of the interface (`"I_HelloWorld"`).
- `RCF_METHOD_xx()` - The `RCF_METHOD_xx()` macros define remote methods. `RCF_METHOD_V1()` defines a remote method taking one parameter (in this case `const std::string &`), and returning `void`. `RCF_METHOD_R2()` defines a remote method with two parameters and a non-`void` return value, and so on. The `RCF_METHOD_xx()` macros are defined for `void` and non-`void` remote calls taking up to 15 parameters.
- `RCF_END()` ends the interface definition.

On the server, we bind the interface to a servant object:

```
server.bind<I_HelloWorld>(helloWorld);
```

Let's add a few more remote methods to the `I_HelloWorld` interface. We'll add methods to print a list of strings, and return the number of characters printed:

```
// Serialization code for std::vector<>.
#include <SF/vector.hpp>


RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
    RCF_METHOD_R1(int,  Print, const std::vector<std::string> &)
    RCF_METHOD_V2(void, Print, const std::vector<std::string> &, int &)
RCF_END(I_HelloWorld)
```

Remote calls can return arguments either through a return value (as the second `Print()` method does), or through a non-const reference parameter (as the third `Print()` method does).

Having added these methods to the `I_HelloWorld` interface, we also need to implement them in the `HelloWorldImpl` servant object:

```
class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }

    int Print(const std::vector<std::string> & v)
    {
        int howManyChars = 0;
        for (std::size_t i=0; i<v.size(); ++i)
        {
            std::cout << "I_HelloWorld service: " << v[i] << std::endl;
            howManyChars += v[i].size();
        }
        return howManyChars;
    }

    void Print(const std::vector<std::string> & v, int & howManyChars)
    {
        howManyChars = 0;
        for (std::size_t i=0; i<v.size(); ++i)
        {
            std::cout << "I_HelloWorld service: " << v[i] << std::endl;
            howManyChars += v[i].size();
        }
    }
};
```

Notice that `HelloWorldImpl` is not a derived class. Instead of virtual functions, RCF uses template-based static polymorphism to bind interfaces to implementations.

Here is sample client code to call the new `Print()` methods:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );

std::vector<std::string> stringsToPrint;
stringsToPrint.push_back("AAA");
stringsToPrint.push_back("BBB");
stringsToPrint.push_back("CCC");

// Remote call returning argument through return value.
int howManyChars = client.Print(stringsToPrint);

// Remote call returning argument through non-const reference parameter.
client.Print(stringsToPrint, howManyChars);
```

And the output:

```
I_HelloWorld service: AAA
I_HelloWorld service: BBB
I_HelloWorld service: CCC
I_HelloWorld service: AAA
I_HelloWorld service: BBB
I_HelloWorld service: CCC
```

Error handling

If a remote call does not complete, RCF will throw an exception containing an error message describing the error condition. Let's put a `try/catch` wrapper around the remote call:

```
try
{
    RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
    client.Print("Hello World");
}
catch(const RCF::Exception & e)
{
    std::cout << "Error: " << e.getErrorString() << std::endl;
}
```

We can simulate the server being down, by commenting out the `RcfServer::start()` call:

```
HelloWorldImpl helloWorld;
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
server.bind<I_HelloWorld>(helloWorld);
//server.start();
```

Running the client, we now get this:

```
Error: Client connection to 127.0.0.1:50001 timed out after 2000 ms (server not started?).
```

All exceptions thrown by RCF are derived from `RCF::Exception`. `RCF::Exception` holds various descriptive and contextual information about the error. You can call `RCF::Exception::getErrorId()` to retrieve the error code, and `RCF::Exception::getErrorString()`, to retrieve an English translation of the error, including any error-specific arguments.

If the server implementation of a remote call throws an exception, it will be caught by the `RcfServer` and returned to the client, where it is thrown as a `RCF::RemoteException`. For example, if we start the server with the following servant implementation:

```
class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        throw std::runtime_error("Print() service is unavailable at this time.");
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }
};
```

, the client will now output:

```
Error: Server-side user exception. Exception type: class std::runtime_error. Exception message: ↵
"Print() service is unavailable at this time.".
```

Client stubs

Remote calls are always made through a client stub (`RCF::ClientStub`). You can access the client stub of a `RcfClient<>` by calling `RcfClient<>::getClientStub()`. The client stub contains a number of settings that affect how a remote call is executed.

Two of the most important settings are the connection timeout and the remote call timeout. The connection timeout determines how long RCF will wait while trying to establish a network connection to the server. The remote call timeout determines how long RCF will wait for a remote call response to return from the server.

To change these settings, call the `ClientStub::setConnectionTimeoutMs()` and `ClientStub::setRemoteCallTimeoutMs()` functions:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );

// 5 second timeout when establishing network connection.
client.getClientStub().setConnectTimeoutMs(5*1000);

// 60 second timeout when waiting for remote call response from the server.
client.getClientStub().setRemoteCallTimeoutMs(60*1000);

client.Print("Hello World");
```

See Remote calls - Client-side for more about client stubs.

Server sessions

On the server-side, RCF maintains a session (`RCF::RcfSession`) for every client connection to the server. The `RcfSession` of a client connection is available to server-side code through `RCF::getCurrentSession()`. You can use `RcfSession` to maintain application data specific to a particular client connection. Arbitrary C++ objects can be stored in a session by calling `RcfSession::createSessionObject<>()` or `RcfSession::getSessionObject<>()`.

For instance, to associate a `HelloWorldSession` object with each client connection to the `I_HelloWorld` interface, which tracks the numer of calls made on a connection:

```
class HelloWorldSession
{
public:
    HelloWorldSession() : mCallCount(0)
    {
        std::cout << "Created HelloWorldSession object." << std::endl;
    }

    ~HelloWorldSession()
    {
        std::cout << "Destroyed HelloWorldSession object." << std::endl;
    }

    std::size_t mCallCount;
};

class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        RCF::RcfSession & session = RCF::getCurrentRcfSession();

        // Creates the session object if it doesn't already exist.
        HelloWorldSession & hwSession = session.getSessionObject<HelloWorldSession>(true);

        ++hwSession.mCallCount;

        std::cout << "I_HelloWorld service: " << s << std::endl;
       std::cout << "I_HelloWorld service: " << "Total calls on this connection so far: " << hwSes↵
sion.mCallCount << std::endl;
    }
};
```

Distinct RcfClient<> instances will have distinct connections to the server. Here we are calling Print() three times from two RcfClient<> instances:

```
for (std::size_t i=0; i<2; ++i)
{
    RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
    client.Print("Hello World");
    client.Print("Hello World");
    client.Print("Hello World");
}

// Wait a little so the server has time to destroy the last session.
RCF::sleepMs(1000);
```

, resulting in the following output:

```
Created HelloWorldSession object.
I_HelloWorld service: Hello World
I_HelloWorld service: Total calls on this connection so far: 1
I_HelloWorld service: Hello World
I_HelloWorld service: Total calls on this connection so far: 2
I_HelloWorld service: Hello World
I_HelloWorld service: Total calls on this connection so far: 3
Destroyed HelloWorldSession object.
Created HelloWorldSession object.
I_HelloWorld service: Hello World
I_HelloWorld service: Total calls on this connection so far: 1
I_HelloWorld service: Hello World
I_HelloWorld service: Total calls on this connection so far: 2
I_HelloWorld service: Hello World
I_HelloWorld service: Total calls on this connection so far: 3
Destroyed HelloWorldSession object.
```

The server session and any associated session objects are destroyed when the client connection is closed.

See Remote calls - Server-side for more about server sessions.

Transports

RCF makes it easy to change the underlying transport of a remote call. The transport layer is determined by the endpoint parameters passed to RcfServer and RcfClient<>. So far we've been using TcpEndpoint, to specify a TCP transport.

By default, when you specify a TcpEndpoint with only a port number, RCF will use 127.0.0.1 as the IP address. So the following two snippets are equivalent:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
```

```
RCF::RcfServer server( RCF::TcpEndpoint("127.0.0.1", 50001) );
```

, as are the following:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
```

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint("127.0.0.1", 50001) );
```

`127.0.0.1` is the IPv4 loopback address. A server listening on `127.0.0.1` will only be available to clients on the same machine as the server. It's likely you'll want to run clients across the network, in which case the server will need to listen on an externally visible network address. The easiest way to do that is to specify `0.0.0.0` (for IPv4), or `::0` (for IPv6) instead, which will make the server listen on all available network interfaces:

```
// Server-side.
RCF::RcfServer server( RCF::TcpEndpoint("0.0.0.0", 50001) );

// Client-side.
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint("Server123", 50001) );
```

RCF supports a number of other endpoint types as well. To run the server and client over UDP, use `UdpEndpoint`:

```
// Server-side.
RCF::RcfServer server( RCF::UdpEndpoint("0.0.0.0", 50001) );

// Client-side.
RcfClient<I_HelloWorld> client( RCF::UdpEndpoint("Server123", 50001) );
```

To run the server and client over named pipes, use `NamedPipeEndpoint`:

```
// Server-side.
RCF::RcfServer server( RCF::Win32NamedPipeEndpoint("MyPipe") );

// Client-side.
RcfClient<I_HelloWorld> client( RCF::Win32NamedPipeEndpoint("MyPipe") );
```

`NamedPipeEndpoint` maps onto Windows named pipes on Windows systems, and UNIX local domain sockets on UNIX-based systems.

It's usually not practical to use UDP for two-way (request/response) messaging, as the unreliable semantics of UDP mean that messages may not be delivered, or may be delivered out of order. UDP is useful in one-way messaging scenarios, where the server application logic is resilient to messages being lost or arriving out of order.

RCF supports tunneling of remote calls over the HTTP and HTTPS protocols. `RCF::HttpEndpoint` is used to configure tunneling over HTTP. Here is an example of a client making remote calls to a server, through a third party HTTP proxy:

```
// Server-side.
HelloWorldImpl helloWorldImpl;
RCF::RcfServer server( RCF::HttpEndpoint("0.0.0.0", 80) );
server.bind<I_HelloWorld>(helloWorldImpl);
server.start();

// Client-side.
// This client will connect to server1.acme.com via the HTTP proxy at proxy.acme.com:8080.
RcfClient<I_HelloWorld> client( RCF::HttpEndpoint("server1.acme.com", 80) );
client.getClientStub().setHttpProxy("proxy.acme.com");
client.getClientStub().setHttpProxyPort(8080);
client.Print("Hello World");
```

Similarly, `RCF::HttpsEndpoint` can be used to configure tunneling over HTTPS:

```
// Server-side.
HelloWorldImpl helloWorldImpl;
RCF::RcfServer server( RCF::HttpsEndpoint("0.0.0.0", 443) );
server.bind<I_HelloWorld>(helloWorldImpl);
RCF::CertificatePtr serverCertPtr( new RCF::PfxCertificate("path/to/certificate.p12", "pass↵
word", "CertificateName") );
server.setCertificate(serverCertPtr);
server.start();

// Client-side.
// This client will connect to server1.acme.com via the HTTP proxy at proxy.acme.com:8080.
RcfClient<I_HelloWorld> client( RCF::HttpsEndpoint("server1.acme.com", 443) );
client.getClientStub().setHttpProxy("proxy.acme.com");
client.getClientStub().setHttpProxyPort(8080);
client.Print("Hello World");
```

Certificates and certificate validation for HTTPS are configured as for the SSL transport protocol (see Transport Protocols below).

Multiple transports can be configured for a single `RcfServer`. For example, to configure a server that accepts both IPv4 and IPv6 connections on port 50001:

```
RCF::RcfServer server;
server.addEndpoint( RCF::TcpEndpoint("0.0.0.0", 50001) );
server.addEndpoint( RCF::TcpEndpoint("::0", 50001) );
server.start();
```

On some platforms, the underlying network stack will allow you to specify `::0` to listen on both IPv4 and IPv6 interfaces, in which case this would be sufficient:

```
RCF::RcfServer server;
server.addEndpoint( RCF::TcpEndpoint("::0", 50001) );
server.start();
```

Here is a server accepting connections over TCP, UDP and named pipes:

```
RCF::RcfServer server;
server.addEndpoint( RCF::TcpEndpoint("::0", 50001) );
server.addEndpoint( RCF::UdpEndpoint("::0", 50002) );
server.addEndpoint( RCF::Win32NamedPipeEndpoint("MyPipe") );
server.start();
```

See Transports for more information on transports.

Transport protocols

Transport protocols are layered on top of the transport, and provide authentication and encryption.

RCF supports the following transport protocols:

- NTLM
- Kerberos
- SSL

The NTLM and Kerberos transport protocols are only supported on Windows, while the SSL transport protocol is supported on all platforms.

An `RcfServer` can be configured to require certain transport protocols:

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

On the client-side, `ClientStub::setTransportProtocol()` is used to configure transport protocols. For instance, to use the NTLM protocol on a client connection:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
client.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
client.Print("Hello World");
```

In this example, the client will authenticate itself to the server and encrypt its connection using the NTLM protocol, using the implicit credentials of the logged on user.

To provide explicit credentials, use `ClientStub::setUsername()` and `ClientStub::setPassword()`:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
client.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
client.getClientStub().setUsername("SomeDomain\\Joe");
client.getClientStub().setPassword("JoesPassword");
client.Print("Hello World");
```

The Kerberos protocol can be configured similarly. The Kerberos protocol requires the client to supply the Service Principal Name (SPN) of the server. You can do this by calling `ClientStub::setKerberosSpn()`:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
client.getClientStub().setTransportProtocol(RCF::Tp_Kerberos);
client.getClientStub().setKerberosSpn("SomeDomain\\ServerAccount");
client.Print("Hello World");
```

If a client attempts to call `Print()` without configuring one of the transport protocols required by the server, they will get an error:

```
try
{
    RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
    client.Print("Hello World");
}
catch(const RCF::Exception & e)
{
    std::cout << "Error: " << e.getErrorString() << std::endl;
}
```

```
Error: Server requires one of the following transport protocols to be used: NTLM, Kerberos.
```

From within the server implementation of `Print()`, you can retrieve the transport protocol of the current session. If the transport protocol is NTLM or Kerberos, you can also retrieve the username of the client, as well as impersonate the client:

```
class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        RCF::RcfSession & session = RCF::getCurrentRcfSession();

        RCF::TransportProtocol protocol = session.getTransportProtocol();

        if ( protocol == RCF::Tp_Ntlm || protocol == RCF::Tp_Kerberos )
        {
            std::string clientUsername = session.getClientUsername();

            RCF::SspiImpersonator impersonator(session);

            // Now running under Windows credentials of client.
            // ...

            // Impersonation ends when we exit scope.
        }

        std::cout << s << std::endl;
    }
};
```

RCF also supports using SSL as the transport protocol. RCF provides two SSL implementations, one based on OpenSSL, and the other based on the Windows Schannel security package. The Windows Schannel implementation is used automatically on Windows, while the OpenSSL implementation is used if `RCF_USE_OPENSSL` is defined when building RCF (see Building).

SSL-enabled servers need to configure a SSL certificate. The mechanics of certificate configuration vary depending on which implementation is being used. Here is an example using the Schannel SSL implementation:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );

RCF::CertificatePtr serverCertificatePtr( new RCF::PfxCertificate(
    "C:\\ServerCert.p12",
    "Password",
    "CertificateName") );

server.setCertificate(serverCertificatePtr);

server.start();

RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
client.getClientStub().setTransportProtocol(RCF::Tp_Ssl);
client.getClientStub().setEnableSchannelCertificateValidation("localhost");
```

`RCF::PfxCertificate` is used to load PKCS #12 certificates, from .pfx and .p12 files. RCF also provides the `RCF::StoreCertificate` class, to load certificates from a Windows certificate store.

When using the OpenSSL-based SSL implementation, the `RCF::PemCertificate` class is used to load PEM certificates, from .pem files.

RCF also supports link level compression of remote calls. Compression is configured independently of transport protocols, using `ClientStub::setEnableCompression()`. The compression stage is applied immediately before the transport protocol stage.

Here is an example of a client connecting to a server through an HTTP proxy, using NTLM for authentication and encryption, and with compression enabled:

```
RcfClient<I_HelloWorld> client( RCF::HttpEndpoint("server123.com", 80) );
client.getClientStub().setHttpProxy("proxy.mycompany.com");
client.getClientStub().setHttpProxyPort(8080);
client.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
client.getClientStub().setEnableCompression(true);

client.Print("Hello World");
```

For more information see Transport protocols.

Server-side threading

By default a `RcfServer` uses a single thread to handle incoming remote calls, so remote calls are dispatched serially, one after the other.

This can be a problem if you have multiple clients and some of them are making calls that take significant time to complete. To improve responsiveness in these situations, a `RcfServer` can be configured to run multiple threads and dispatch remote calls in parallel.

To configure a multi-threaded `RcfServer`, use `RcfServer::setThreadPool()` to assign a thread pool to the server. RCF thread pools can be configured to use either a fixed number of threads:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );

// Thread pool with fixed number of threads (5).
RCF::ThreadPoolPtr tpPtr( new RCF::ThreadPool(5) );
server.setThreadPool(tpPtr);

server.start();
```

, or a dynamic number of threads, varying with server load:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );

// Thread pool with varying number of threads (1 to 25).
RCF::ThreadPoolPtr tpPtr( new RCF::ThreadPool(1, 25) );
server.setThreadPool(tpPtr);

server.start();
```

Thread pools configured through `RcfServer::setThreadPool()` are shared across all the transports of that `RcfServer`. It is also possible to use `I_ServerTransport::setThreadPool()` to configure thread pools for individual transports:

```
RCF::RcfServer server;

RCF::ServerTransport & tcpTransport = server.addEndpoint(RCF::TcpEndpoint(50001));
RCF::ServerTransport & pipeTransport = server.addEndpoint(RCF::Win32NamedPipeEndpoint("MyPipe"));

// Thread pool with up to 5 threads to serve TCP clients.
RCF::ThreadPoolPtr tcpThreadPoolPtr( new RCF::ThreadPool(1, 5) );
tcpTransport.setThreadPool(tcpThreadPoolPtr);

// Thread pool with single thread to serve named pipe clients.
RCF::ThreadPoolPtr pipeThreadPoolPtr( new RCF::ThreadPool(1) );
pipeTransport.setThreadPool(pipeThreadPoolPtr);

server.start();
```

Asynchronous remote calls

RCF supports client-side asynchronous remote call invocation, and server-side asynchronous remote call dispatching.

Asynchronous remote call invocation

On the client-side, asynchronous remote calls allow you to avoid blocking the thread that is making a remote call. Asynchronous remote calls complete on background threads, and the calling thread is notified at a later point, when the call completes.

Asynchronous remote call invocation is implemented in RCF using the `RCF::Future<>` template. Here is a simple example of waiting for an asynchronous call to complete:

```
HelloWorldImpl helloWorld;
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
server.bind<I_HelloWorld>(helloWorld);
server.start();

RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );

RCF::Future<int> fRet;

// Asynchronous remote call.
fRet = client.Print("Hello World");

// Wait for the call to complete.
while (!fRet.ready()) RCF::sleepMs(1000);

// Check for errors.
std::auto_ptr<RCF::Exception> ePtr = client.getClientStub().getAsyncException();
if (ePtr.get())
{
    // Error handling.
    // ...
}
else
{
    int howManyCharsPrinted = *fRet;
}
```

A remote call is performed asynchronously, if any of the parameters, or the return value, is of type `RCF::Future<>`, or if `RCF::AsyncOneway` or `RCF::AsyncTwoway` is specified as the calling semantic. Once a call is complete, `RCF::Future<>` instances can be dereferenced to retrieve the return values. If the call completed with an error, the error can be retrieved by calling `RCF::ClientStub::getAsyncException()` . The error will also be thrown if an attempt is made to dereference an associated `RCF::Future<>` instance.

Instead of waiting for the result on the calling thread, we can assign a completion callback to the remote call:

```
typedef boost::shared_ptr< RcfClient<I_HelloWorld> > HelloWorldPtr;

void onPrintCompleted(
    RCF::Future<int> fRet,
    HelloWorldPtr clientPtr)
{
    int howManyCharsPrinted = *fRet;
}
```

```
HelloWorldPtr clientPtr(
    new RcfClient<I_HelloWorld>(RCF::TcpEndpoint(50001)) );

RCF::Future<int> fRet;

// Asynchronous remote call, with completion callback.
fRet = clientPtr->Print(
    RCF::AsyncTwoway( boost::bind(&onPrintCompleted, fRet, clientPtr) ),
    "Hello World");
```

We've passed both the `Future<int>` return value and a reference counted client (`HelloWorldPtr`) to the callback. If we were to destroy the client on the main thread, the asynchronous call would be automatically canceled, and the callback would not be called at all.

Asynchronous remote calls are useful in many circumstances. For instance, the following code will call `Print()` once every 10 seconds on 50 different servers:

```
void onPrintCompleted(HelloWorldPtr clientPtr);
void onWaitCompleted(HelloWorldPtr clientPtr);

void onPrintCompleted(HelloWorldPtr clientPtr)
{
    // Print() call completed. Wait for 10 seconds.

    clientPtr->getClientStub().wait(
        boost::bind(&onWaitCompleted, clientPtr),
        10*1000);
}

void onWaitCompleted(HelloWorldPtr clientPtr)
{
    // 10 second wait completed. Make another Print() call.

    clientPtr->Print(
        RCF::AsyncTwoway( boost::bind(onPrintCompleted, clientPtr)),
        "Hello World");
}
```

```
// Addresses to 50 servers.
std::vector<RCF::TcpEndpoint> servers(50);
// ...

// Create a connection to each server, and make the first call.
std::vector<HelloWorldPtr> clients;
for (std::size_t i=0; i<50; ++i)
{
    HelloWorldPtr clientPtr( new RcfClient<I_HelloWorld>(servers[i]) );
    clients.push_back(clientPtr);

    // Asynchronous remote call, with completion callback.
    clientPtr->Print(
        RCF::AsyncTwoway( boost::bind(&onPrintCompleted, clientPtr)),
        "Hello World");
}

// All 50 servers are now being called once every 10 s.
// ...

// Upon leaving scope, the clients are all automatically destroyed. Any
// remote calls in progress are automatically canceled.
```

Instead of blocking 50 threads with synchronous remote calls, all 50 connections are managed by a single thread. The remote call sequences to the 50 servers are all handled on a background RCF thread, and the connections are automatically destroyed when the main thread leaves scope.

Asynchronous remote call dispatching

RCF also supports server-side asynchronous remote call dispatching. When a remote call arrives at a `RcfServer`, it is handled by one of the threads in the servers thread pool. To process the call asynchronously on another thread, you can use the `RCF::RemoteCallContext<>` template to capture the server-side context of the remote call, and queue it for later processing.

Here is an example of using `RemoteCallContext<>` in server-side code. Instead of responding to `Print()` calls in `HelloWorldImpl::Print()`, we are passing the remote call contexts to a background thread to be processed in bulk once every second. The background thread uses `RemoteCallContext::parameters()` to access the parameters of each remote call, and `RemoteCallContext::commit()` to send the remote call response back to the client.

```
#include <RCF/RemoteCallContext.hpp>


RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_R1(int, Print, const std::string &)
RCF_END(I_HelloWorld)

// I_HelloWorld servant object
class HelloWorldImpl
{
public:

    typedef RCF::RemoteCallContext<int, const std::string &> PrintCall;

    int Print(const std::string & s)
    {
        // Capture the remote call context and queue it in mPrintCalls.
        RCF::Lock lock(mPrintCallsMutex);
        mPrintCalls.push_back( PrintCall(RCF::getCurrentRcfSession()) );

        // Dummy return value.
        return 0;
    }

    HelloWorldImpl()
    {
        // Start the asynchronous printing thread.
        mStopFlag = false;

        mPrintThreadPtr.reset( new RCF::Thread( boost::bind(
            &HelloWorldImpl::processPrintCalls,
            this) ) );
    }

    ~HelloWorldImpl()
    {
        // Stop the asynchronous printing thread.
        mStopFlag = true;
        mPrintThreadPtr->join();
    }

private:

    // Queue of remote calls.
    RCF::Mutex              mPrintCallsMutex;
    std::deque<PrintCall>   mPrintCalls;

    // Asynchronous printing thread.
    RCF::ThreadPtr          mPrintThreadPtr;
    volatile bool           mStopFlag;

    void processPrintCalls()
    {
        // Once a second, process all queued Print() calls.
        while (!mStopFlag)
        {
            Sleep(1000);

            // Retrieve all queued print calls.
            std::deque<PrintCall> printCalls;
            {
                RCF::Lock lock(mPrintCallsMutex);
                printCalls.swap(mPrintCalls);
            }

            // Process them.
            for (std::size_t i=0; i<printCalls.size(); ++i)
            {
                PrintCall & printCall = printCalls[i];
                const std::string & stringToPrint = printCall.parameters().a1.get();
                std::cout << "I_HelloWorld service: " << stringToPrint << std::endl;
                printCall.parameters().r.set( stringToPrint.size() );
                printCall.commit();
            }
        }
    }
};
```

The client code is unchanged:

```
int main()
{
    RCF::RcfInitDeinit rcfInit;

    HelloWorldImpl helloWorld;
    RCF::RcfServer server( RCF::TcpEndpoint(50001) );
    server.bind<I_HelloWorld>(helloWorld);
    server.start();

    RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
    int charsPrinted = client.Print("Hello World");

    return 0;
}
```

For more information, see Asynchronous Remote Calls.

Publish/subscribe

RCF makes it easy to setup publish/subscribe feeds. For example, here is a publisher publishing `"Hello World"` once every second:

```
RCF::RcfServer publishingServer( RCF::TcpEndpoint(50001) );
publishingServer.start();

// Start publishing.
typedef boost::shared_ptr< RCF::Publisher<I_HelloWorld> > HelloWorldPublisherPtr;
HelloWorldPublisherPtr pubPtr = publishingServer.createPublisher<I_HelloWorld>();

while (shouldContinue())
{
    Sleep(1000);

    // Publish a Print() call to all currently connected subscribers.
    pubPtr->publish().Print("Hello World");
}

// Close the publisher.
pubPtr->close();
```

To create a subscription to the publisher:

```
// Start a subscriber.
RCF::RcfServer subscriptionServer(( RCF::TcpEndpoint() ));
subscriptionServer.start();

HelloWorldImpl helloWorld;

RCF::SubscriptionPtr subPtr = subscriptionServer.createSubscription<I_HelloWorld>(
    helloWorld,
    RCF::TcpEndpoint(50001));

// At this point Print() will be called on the helloWorld object once a second.
// ...

// Close the subscription.
subPtr->close();
```

Each call the publisher makes is sent as a oneway call to all subscribers.

For more information on publish/subscribe messaging, see Publish/subscribe.

Callback connections

So far we've seen clients making remote calls to servers. It's also possible for a server to make remote calls back to a client, once the client has established a connection. To do this, the client starts a `RcfServer` of its own, and calls `RcfServer::createCallback-Connection()`:

```
// Client-side
int main()
{
    RCF::RcfInitDeinit rcfInit;

    // Client needs a RcfServer to accept callback connections.
    RCF::RcfServer callbackServer(( RCF::TcpEndpoint() ));
    HelloWorldImpl helloWorld;
    callbackServer.bind<I_HelloWorld>(helloWorld);
    callbackServer.start();

    // Establish client connection to server.
    RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );

    // Create the callback connection.
    RCF::createCallbackConnection(client, callbackServer);

    // Server can now call Print() on the helloWorld object.
    // ...

    return 0;
}
```

On the server-side, `RcfServer::setCallbackConnectionCb()` is used to take control over the callback connection once it has been created by the client:

```
// Server-side

typedef boost::shared_ptr< RcfClient<I_HelloWorld> >    HelloWorldPtr;
RCF::Mutex                                              gCallbackClientsMutex;
std::vector< HelloWorldPtr >                            gCallbackClients;

void onCallbackConnectionCreated(
    RCF::RcfSessionPtr sessionPtr,
    RCF::ClientTransportAutoPtr transportAutoPtr)
{
    typedef boost::shared_ptr< RcfClient<I_HelloWorld> > HelloWorldPtr;
    HelloWorldPtr helloWorldPtr( new RcfClient<I_HelloWorld>(transportAutoPtr) );
    RCF::Lock lock(gCallbackClientsMutex);
    gCallbackClients.push_back( helloWorldPtr );
}


int main()
{
    RCF::RcfInitDeinit rcfInit;

    RCF::RcfServer server( RCF::TcpEndpoint(50001) );

    server.setOnCallbackConnectionCreated(
        boost::bind(&onCallbackConnectionCreated, _1, _2) );

    server.start();

    // Wait for clients to create callback connections.
    // ...

    // Retrieve all created callback connections.
    std::vector<HelloWorldPtr> clients;
    {
        RCF::Lock lock(gCallbackClientsMutex);
        clients.swap(gCallbackClients);
    }

    // Call Print() on them.
    for (std::size_t i=0; i<clients.size(); ++i)
    {
        HelloWorldPtr clientPtr = clients[i];
        clientPtr->Print("Hello World");
    }

    return 0;
}
```

Callback connections function just like regular connections, with the exception that callback connections cannot be reconnected. If a callback connection is lost, the client will need to connect again and call `RCF::createCallbackConnection()`, to re-establish connectivity.

For more information on callback connections, see Callback Connections.

File transfers

File downloads and uploads are common in distributed systems. RCF provides built-in support for file transfers, through the `RCF::FileDownload` and `RCF::FileUpload` classes. `RCF::FileDownload` and `RCF::FileUpload` are used as remote method arguments, to implement file transfers as part of a remote call.

On the server-side, file transfer functionality is by default disabled, and needs to be explicitly enabled:

```
HelloWorldImpl helloWorld;
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
server.bind<I_HelloWorld>(helloWorld);

server.start();
```

Here we've implemented a `PrintAndDownload()` method, that allows clients to download files:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
    RCF_METHOD_V2(void, PrintAndDownload, const std::string &, RCF::FileDownload)
RCF_END(I_HelloWorld)

class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }

    void PrintAndDownload(const std::string & s, RCF::FileDownload fileDownload)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
        fileDownload = RCF::FileDownload("path/to/download");
    }
};
```

, which can be called like this:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
RCF::FileDownload fileDownload("path/to/download/to");
client.PrintAndDownload("Hello World", fileDownload);

std::string pathToDownload = fileDownload.getLocalPath();
RCF::FileManifest & downloadManifest = fileDownload.getManifest();
std::cout << "Client-local path to upload: " << pathToDownload << std::endl;
std::cout << "Number of files uploaded: " << downloadManifest.mFiles.size() << std::endl;
```

Here we've implemented a `PrintAndUpload()` method, that allows clients to upload files:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
    RCF_METHOD_V2(void, PrintAndUpload, const std::string &, RCF::FileUpload)
RCF_END(I_HelloWorld)

class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }

    void PrintAndUpload(const std::string & s, RCF::FileUpload fileUpload)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
        std::string pathToUpload = fileUpload.getLocalPath();
        RCF::FileManifest & uploadManifest = fileUpload.getManifest();
        std::cout << "Server-local path to upload: " << pathToUpload << std::endl;
        std::cout << "Number of files uploaded: " << uploadManifest.mFiles.size() << std::endl;
    }
};
```

, which can be called like this:

```
// Upload files to server.
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
RCF::FileUpload fileUpload("path/to/files");
client.PrintAndUpload("Hello World", fileUpload);
```

File transfers can be monitored, paused, resumed, and canceled , and bandwidth throttles can be applied to file transfers to and from a given server.

For more information, see File transfers.

Protocol Buffers

RCF provides native integration with Protocol Buffers. Classes that are generated by the Protocol Buffers compiler can be used in RCF interfaces, and serialization and deserialization is performed through the relevant Protocol Buffers functions.

For example, running the `protoc` compiler over the following Protobuf interface:

```
// Person.proto

message Person {
    required int32 id = 1;
    required string name = 2;
    optional string email = 3;
}

message PbEmpty {
    optional string log = 1;
}
```

```
protoc Person.proto --cpp_out=.
```

, generates the C++ class `Person`, defined in `Person.pb.h`.

By including `Person.pb.h`, we can send `Person` instances to our `Print()` function:

```
#include <../test/protobuf/messages/cpp/Person.pb.h>


RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
    RCF_METHOD_V1(void, Print, const Person &)
RCF_END(I_HelloWorld)

class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << s << std::endl;
    }

    void Print(const Person & person)
    {
        std::cout << "Person name: " << person.name();
        std::cout << "Person email: " << person.email();
        std::cout << "Person id: " << person.id();
    }
};


int main()
{
    RCF::RcfInitDeinit rcfInit;

    HelloWorldImpl helloWorld;
    RCF::RcfServer server( RCF::TcpEndpoint(50001) );
    server.bind<I_HelloWorld>(helloWorld);
    server.start();

    RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );

    Person person;
    person.set_name("Bob");
    person.set_email("bob@acme.com");
    person.set_id(123);
    client.Print(person);

    return 0;
}
```

For more information on Protocol Buffers support, see Protocol Buffers.

JSON-RPC

RCF provides a built in JSON-RPC server, allowing access to C++ server functionality from JSON-RPC clients, such as Javascript code on web pages. RCF supports JSON-RPC over both HTTP and HTTPS.

RCF uses the JSON Spirit library to read and write JSON messages. You will need to download this library separately, and define `RCF_USE_JSON` when building RCF, to enable JSON-RPC support (see Appendix - Building).

To configure a JSON-RPC endpoint, call `I_ServerTransport::setRpcProtocol()` on the relevant server transport. To expose servant objects to JSON-RPC clients, use `RcfServer::bindJsonRpc()`.

Here is an example of a RCF server using multiple transports to accept both RCF requests and JSON-RPC requests:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
RCF_END(I_HelloWorld)

class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }

    void JsonPrint(
        const RCF::JsonRpcRequest & request,
        RCF::JsonRpcResponse & response)
    {
        // Print out all the strings passed in, and return the number of
        // characters printed.

        int charsPrinted = 0;

        const json_spirit::Array & params = request.getJsonParams();
        for (std::size_t i=0; i<params.size(); ++i)
        {
            const std::string & s = params[i].get_str();
            std::cout << "I_HelloWorld service: " << s << std::endl;
            charsPrinted += s.size();
        }

        // Return number of characters printed.
        json_spirit::mObject & responseObj = response.getJsonResponse();
        responseObj["result"] = charsPrinted;
    }
};


int main()
{
    RCF::RcfInitDeinit rcfInit;

    RCF::RcfServer server;

    // Accept RCF client requests on port 50001.
    HelloWorldImpl helloWorld;
    server.bind<I_HelloWorld>(helloWorld);
    server.addEndpoint( RCF::TcpEndpoint(50001) );

    // Accept JSON-RPC requests over HTTP on port 80.
    server.bindJsonRpc(
        boost::bind(&HelloWorldImpl::JsonPrint, &helloWorld, _1, _2),
        "JsonPrint");

    server.addEndpoint( RCF::HttpEndpoint(80) )
        .setRpcProtocol(RCF::Rp_JsonRpc);

    server.start();

    // RCF clients can call Print() on port 50001.
    // ...

    // JSON-RPC clients can call JsonPrint() over HTTP on port 80.
    // ...

    return 0;
}
```

For more information on JSON-RPC servers, see JSON-RPC.

Remote Calls - Client-side

The client-side of a remote call is controlled through `RCF::ClientStub`. `RCF::ClientStub` controls a single connection to the server. Every `RcfClient<>` instance contains a `RCF::ClientStub`, which can be accessed by calling `getClientStub()`.

```
RCF::ClientStub & clientStub = client.getClientStub();
```

Remote call semantics

You can use `ClientStub::setRemoteCallSemantics()` to set the calling semantics of a remote call.

```
client.getClientStub().setRemoteCallSemantics(RCF::Oneway);
client.Print("Hello World");

client.getClientStub().setRemoteCallSemantics(RCF::Twoway);
client.Print("Hello World");
```

- Twoway calls (`RCF::Twoway`) are the default. A twoway call does not complete until the client receives a reply from the server.
- A oneway call (`RCF::Oneway`) is complete as soon as the request has been sent to the server. The server will not send any response to a oneway call.

By default a twoway call will cause the calling thread to wait until a response is received from the server. It is also possible to receive the response asynchronously, thus freeing the calling thread to perform other tasks - see Asynchronous Remote Calls. Asynchronous semantics can also be used on oneway calls, to prevent the calling thread from blocking if the local network send buffers are full.

Calling semantics can also be provided on a call-by-call basis, as the first argument of the remote call:

```
client.Print(RCF::Oneway, "Hello World");
client.Print(RCF::Twoway, "Hello World");
```

RCF also provides batched oneway calls, as an optimization for oneway calls. Normally, when a `RcfClient<>` is configured to make oneway calls, a network message is sent to the server for each remote call that is made. If a large number of calls are being made, batched oneway calls can be configured, allowing multiple oneway calls to be coalesced into a single network message, and subsequently executed in sequence on the server.

To configure batched oneway calls, use the `ClientStub::enableBatching()`, `ClientStub::disableBatching()` and `ClientStub::flushBatch()` functions:

```
client.getClientStub().enableBatching();

// Automatically send batch when message size approaches 50kb.
client.getClientStub().setMaxBatchMessageLength(1024*50);
for (std::size_t i=0; i<100; ++i)
{
    client.Print("Hello World");
}

// Send final batch.
client.getClientStub().flushBatch();
```

Pinging

The `ClientStub::ping()` method can be used, to determine if the connection to the server is functional. A ping behaves exactly as a remote call with no in or out parameters, and is subject to the same timeouts.

```
// Ping the server.
client.getClientStub().ping();
```

Client progress callbacks

RCF can be configured to periodically report progress to a custom client progress callback function during a twoway call. You can use the progress callback function to display a progress bar, repaint the application window, or even cancel the call itself.

```
void onRemoteCallProgress()
{
    // To cancel the call, throw an exception.
    throw std::runtime_error("Canceling remote call.");
}
```

```
boost::uint32_t progressCallbackIntervalMs = 500;

client.getClientStub().setRemoteCallProgressCallback(
    boost::bind(&onRemoteCallProgress),
    progressCallbackIntervalMs);

// While the call is in progress, onRemoteCallProgress() will be called every 500ms.
client.Print("Hello World");
```

Per-request user data

Normally the data transferred from the client to the server in a remote call is contained in the arguments of the remote call. However, you can also supply extra information in the user data field of the remote call request. Similarly, the server can send extra information to the client by setting the the user data field of the remote call response.

```
class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;

        RCF::RcfSession & session = RCF::getCurrentRcfSession();

        std::string customRequestData = session.getRequestUserData();
        std::cout << "Custom request data: " << customRequestData << std::endl;

        std::string customResponseData = "f81d4fae-7dec-11d0-a765-00a0c91e6bf6";
        session.setResponseUserData(customResponseData);
    }
};
```

```
client.getClientStub().setRequestUserData( "e6a9bb54-da25-102b-9a03-2db401e887ec" );
client.Print("Hello World");
std::string customReponseData = client.getClientStub().getResponseUserData();
std::cout << "Custom response data: " << customReponseData << std::endl;
```

The user data fields can be used to implement custom authentication schemes where a token needs to be passed along as part of every remote call, but you don't want to have the token to be part of your remote call interface.

Access to underlying transport

You can use `ClientStub::getTransport()` to access the underlying transport of a ClientStub.

```
RCF::ClientTransport & transport = client.getClientStub().getTransport();
```

Transport level connection and disconnection from a server is normally handled automatically. However, you can also connect and disconnect manually by calling `ClientStub::connect()` and `ClientStub::disconnect()`.

```
// Connect to server.
client.getClientStub().connect();

// Disconnect from server.
client.getClientStub().disconnect();
```

You can use `ClientStub::releaseTransport()` and `ClientStub::setTransport()` to move a transport from one client stub to another:

```
RcfClient<I_AnotherInterface> client2( client.getClientStub().releaseTransport() );
client2.AnotherPrint("Hello World");
client.getClientStub().setTransport( client2.getClientStub().releaseTransport() );
client.Print("Hello World");
```

This is useful if you want to use the same connection to make calls on different interfaces.

Copy semantics

`RCF::RcfClient<>` objects can be copied, put in containers, and so on. However, each copy of a `RcfClient<>` object will establish its own network connection to the server. So the following code will establish 3 network connections to the server:

```
RcfClient<I_HelloWorld> client1(( RCF::TcpEndpoint(port) ));

RcfClient<I_HelloWorld> client2(client1);

RcfClient<I_HelloWorld> client3;
client3 = client1;

client1.Print("Hello World");
client2.Print("Hello World");
client3.Print("Hello World");
```

Remote Calls - Server-side

On the server-side, your application is responsible for dispatching remote calls from clients. Typically you will create a `RcfServer`:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
```

, bind a servant object to it:

```
HelloWorldImpl helloWorldImpl;
server.bind<I_HelloWorld>(helloWorldImpl);
```

, and start the server:

```
server.start();
```

When a client makes a remote call to the `I_HelloWorld` interface, the relevant member function of the servant object is executed. From within the servant object, you can access the current server side remote call session through `RCF::getCurrentSession()`:

```
class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;

        RCF::RcfSession & session = RCF::getCurrentRcfSession();
        // ...
    }
};
```

`RCF::getCurrentSession()` returns a `RcfSession` reference, through which server-side per-session configuration is done. A `RcfSession` is created for every connection to a RcfServer, and it's lifetime matches that of the underlying transport connection.

Configuring a server

Adding transports

A `RcfServer` listens on one or more transports, for remote calls from clients. Server transports can be configured by passing in an endpoint parameter to the `RcfServer` constructor:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
```

Alternatively, to configure multiple transports, use `RcfServer::addEndpoint()`:

```
RCF::RcfServer server;
server.addEndpoint( RCF::TcpEndpoint(50001) );
server.addEndpoint( RCF::UdpEndpoint(50002) );
```

Starting and stopping a server

A `RcfServer` instance will not start dispatching remote calls until `RcfServer::start()` is called. The server can be stopped manually by calling `RcfServer::stop()`:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
server.start();
// ...
server.stop();
```

You can also simply let the `RcfServer` object go out of scope, in which case the server is stopped automatically.

Binding servant objects

`RcfServer::bind<>()` is used to bind servant objects to RCF interfaces. Every bound servant object has a binding name which identifies it. The default servant binding name is the runtime name of the interface it is exposed through. So the following code:

```
server.bind<I_HelloWorld>(helloWorldImpl);
```

, creates a servant binding with the servant binding name `"I_HelloWorld"`, as that is the runtime identifier of the `I_HelloWorld` interface:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
RCF_END(I_HelloWorld)
```

The servant binding name can also be set explicitly:

```
server.bind<I_HelloWorld>(helloWorldImpl, "CustomBindingName");
```

Each remote call from a client specifies a servant binding name, which is used by the server to dispatch the remote call. The servant binding name supplied by a `RcfClient<>` defaults to the runtime name of the RCF interface it is using. So the following client code:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
client.Print("Hello World");
```

, will make a remote call to the servant object with the servant binding name `"I_HelloWorld"`.

The client can also set the servant binding name explicitly:

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001), "CustomBindingName" );
client.Print("Hello World");
```

, to get the call dispatched to the nominated servant binding.

Server threading

By default, a `RcfServer` will use a single thread to dispatch calls across all its transports. This behavior can be modified by explicitly assigning a thread pool to the `RcfServer`:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );

RCF::ThreadPoolPtr threadPoolPtr( new RCF::ThreadPool(1, 5) );
server.setThreadPool(threadPoolPtr);
```

`RCF::ThreadPool` can be configured to use a fixed number of threads, or a varying number of threads depending on server load:

```
// Thread pool with a fixed number of threads (5).
RCF::ThreadPoolPtr threadPoolPtr( new RCF::ThreadPool(5) );
server.setThreadPool(threadPoolPtr);
```

```
// Thread pool with a dynamically varying number of threads (1-25).
RCF::ThreadPoolPtr threadPoolPtr( new RCF::ThreadPool(1, 25) );
server.setThreadPool(threadPoolPtr);
```

A thread pool assigned to the `RcfServer` will be shared by all the transports of that `RcfServer`. It is also possible to assign thread pools to specific transports:

```
RCF::RcfServer server;

RCF::ThreadPoolPtr threadPool1( new RCF::ThreadPool(1) );
server.addEndpoint( RCF::TcpEndpoint(50001) ).setThreadPool(threadPool1);

RCF::ThreadPoolPtr threadPool2( new RCF::ThreadPool(1) );
server.addEndpoint( RCF::TcpEndpoint(50002) ).setThreadPool(threadPool2);
```

Server-side sessions

You can use `RcfSession` to find out a number of things about the current client connection, including:

- The transport being used:

```
RCF::RcfSession & session = RCF::getCurrentRcfSession();
RCF::TransportType transportType = session.getTransportType();
```

- The network address of the client:

```
const RCF::RemoteAddress & clientAddress = session.getClientAddress();
std::string strClientAddress = clientAddress.string();
```

- The current request header:

```
RCF::RemoteCallRequest request = session.getRemoteCallRequest();
```

- Any custom request data passed in by the client:

```
std::string customRequestData = session.getRequestUserData();
```

- You can create, retrieve and delete session objects:

```
session.createSessionObject<MySessionObj>();
MySessionObj & obj = session.getSessionObject<MySessionObj>();
MySessionObj * pObj = session.querySessionObject<MySessionObj>();
session.deleteSessionObject<MySessionObj>();
```

- How long the client has been connected:

```
// When the connection was established, as reported by CRT time() function.
time_t connectedAt = session.getConnectedAtTime();

// Connection duration in seconds.
boost::uint32_t connectionDurationS = session.getConnectionDuration();
```

- How many calls have been made on this connection:

```
boost::uint32_t callsMade = session.getRemoteCallCount();
```

- How many bytes have been sent and received on this connection:

```
boost::uint64_t totalBytesReceived = session.getTotalBytesReceived();
boost::uint64_t totalBytesSent = session.getTotalBytesSent();
```

Serialization

Serialization is a fundamental part of every remote call. Remote call arguments need to be serialized into binary form so they can be transmitted across the network, and once received, they need to be deserialized from binary form back into regular C++ objects.

RCF provides a serialization framework of its own, SF, which handles serialization of common C++ types automatically, and can be customized to serialize arbitrary user-defined C++ types.

Standard C++ types

Serialization of fundamental C++ types (`char`, `int`, `double`, etc) is handled automatically by RCF. Serialization of a number of other common and standard C++ types requires inclusion of the relevant header file:

Type	Which header to include
std::string, std::wstring, std::basic_string<>	#include <SF/string.hpp>
std::vector<>	#include <SF/vector.hpp>
std::list<>	#include <SF/list.hpp>
std::deque<>	#include <SF/deque.hpp>
std::set<>	#include <SF/set.hpp>
std::map<>	#include <SF/map.hpp>
std::pair<>	#include <SF/utility.hpp>
std::bitset<>	#include <SF/bitset.hpp>
std::auto_ptr<>	#include <SF/auto_ptr.hpp>
std::unique_ptr<>	#include <SF/unique_ptr.hpp>
boost::shared_ptr<>	#include <SF/shared_ptr.hpp>
std::tr1::shared_ptr<>	#include <SF/shared_ptr_tr1.hpp>
std::shared_ptr<>	#include <SF/shared_ptr_std.hpp>
boost::scoped_ptr<>	#include <SF/scoped_ptr.hpp>
boost::intrusive_ptr<>	#include <SF/intrusive_ptr.hpp>
boost::any	#include <SF/any.hpp>
boost::tuple<>	#include <SF/tuple.hpp>
std::tr1::tuple<>	#include <SF/tuple_tr1.hpp>
std::tuple<>	#include <SF/tuple_std.hpp>
boost::variant<>	#include <SF/variant.hpp>
boost::array<>	#include <SF/array.hpp>
std::tr1::array<>	#include <SF/array_tr1.hpp>
std::array<>	#include <SF/array_std.hpp>
std::tr1::unordered_map<>, std::tr1::unordered_multimap<>	#include <SF/unordered_map.hpp>
std::tr1::unordered_set<>, std::tr1::unordered_multiset<>	#include <SF/unordered_set.hpp>
stdext::hash_map<>, stdext::hash_multimap<>	#include <SF/hash_map.hpp>
stdext::hash_set<>, stdext::hash_multiset<>	#include <SF/hash_set.hpp>

In general, for a standard C++ type defined in `<xyz>`, or a Boost type defined in `<boost/xyz.hpp>`, the corresponding serialization definitions are located in `<SF/xyz.hpp>` .

C++ enums are serialized automatically, as integers. Serialization of C++11 enum classes requires the use of a helper macro:

```
// Legacy C++ enum. Automatically serialized as 'int'.
enum Suit
{
    Heart = 1,
    Diamond = 2,
    Club = 2,
    Spade = 2
};

// C++11 enum class with custom base type (8 bit integer).
enum class Colors : std::int8_t
{
    Red = 1,
    Green = 2,
    Blue = 3
};

// Use SF_SERIALIZE_ENUM_CLASS() to specify the base type of the enum class.
SF_SERIALIZE_ENUM_CLASS(Colors, std::int8_t)
```

By composing containers, you can build up structures of considerable complexity that can be used in RCF interfaces, without having to write any serialization code at all:

```
#include <boost/tuple/tuple.hpp>

#include <RCF/Idl.hpp>
#include <RCF/IpServerTransport.hpp>
#include <RCF/RcfServer.hpp>
#include <RCF/TcpEndpoint.hpp>

#include <SF/map.hpp>
#include <SF/list.hpp>
#include <SF/tuple.hpp>
#include <SF/vector.hpp>

typedef
std::map<
    std::string,
    std::list<
        std::pair<
            int,
            std::string> > >    MyMap;

typedef std::string                                                 Name;
typedef std::string                                                 Street;
typedef unsigned int                                                Zip;
typedef std::string                                                 Suburb;
typedef std::string                                                 State;
typedef std::string                                                 Country;

typedef boost::tuple<Name, Street, Zip, Suburb, State, Country> Address;
typedef std::vector<Address>                                        Addresses;


RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(MyMap, Echo, const MyMap &)
    RCF_METHOD_R1(Addresses, Echo, const Addresses &)
RCF_END(I_Echo)

class EchoImpl
{
public:
    MyMap Echo(const MyMap &myMap)
    {
        return myMap;
    }

    Addresses Echo(const Addresses &addresses)
    {
        return addresses;
    }
};


int main()
{
    EchoImpl echoImpl;
    RCF::RcfServer server( RCF::TcpEndpoint(0));
    server.bind<I_Echo>(echoImpl);
    server.start();

    int port = server.getIpServerTransport().getPort();

    MyMap myMap;
    myMap["1-3"].push_back( std::make_pair(1, "one"));
    myMap["1-3"].push_back( std::make_pair(2, "two"));
    myMap["1-3"].push_back( std::make_pair(3, "three"));
    myMap["4-6"].push_back( std::make_pair(4, "four"));
    myMap["4-6"].push_back( std::make_pair(5, "five"));
    myMap["4-6"].push_back( std::make_pair(6, "six"));

    RcfClient<I_Echo> client(( RCF::TcpEndpoint(port) ));

    // Echo a MyMap object.
    MyMap myMap2 = client.Echo(myMap);

    Addresses addresses;
    addresses.push_back( Address("", "", 123, "", "", ""));
    addresses.push_back( Address("", "", 456, "", "", ""));
    addresses.push_back( Address("", "", 789, "", "", ""));

    // Echo a Addresses object.
    Addresses addresses2 = client.Echo(addresses);

    return 0;
}
```

User-defined types

If you take a class of your own, and use it in an RCF interface:

```
class Point3D
{
public:
    double mX;
    double mY;
    double mZ;
};

RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(Point3D, Echo, const Point3D &)
RCF_END(I_Echo)
```

, you'll end up with an compiler error similar to this one:

```
..\..\..\..\..\include\SF\Serializer.hpp(324) : error C2039: 'serialize' : is not a member of ↵
'Point3D'
        C6.cpp(13) : see declaration of 'Point3D'
        ..\..\..\..\..\include\SF\Serializer.hpp(336) : see reference to function template in↵
stantiation 'void SF::serializeInternal<T>(SF::Archive &,T &)' being compiled
        with
        [
            T=U
        ]
        <snip>
```

The compiler is telling us that it couldn't find any serialization code for the class `Point3D`. We need to provide a `serialize()` function, either as a member function:

```
class Point3D
{
public:
    double mX;
    double mY;
    double mZ;

    // Internal serialization.
    void serialize(SF::Archive &ar)
    {
        ar & mX & mY & mZ;
    }
};
```

, or as a free function in either the same namespace as `Point3D`, or in the SF namespace:

```
// External serialization.
void serialize(SF::Archive &ar, Point3D &point)
{
    ar & point.mX & point.mY & point.mY;
}
```

The code in the `serialize()` function specifies which members to serialize.

The `serialize()` function is used both for serialization and deserialization. In some cases, you may want to use different logic, depending on whether you are serializing or deserializing. For example, the following snippet implements serialization of the `boost::gregorian::date` class, by representing it as a string:

```
// Serialization code for boost::gregorian::date can be placed either in
// the boost::gregorian::date namespace (where the path class is defined), or
// in the SF namespace. Here we've chosen the SF namespace.
namespace SF {

    void serialize(SF::Archive & ar, boost::gregorian::date & dt)
    {
        if (ar.isWrite())
        {
            // Code for serializing.
            std::ostringstream os;
            os << dt;
            std::string s = os.str();
            ar & s;
        }
        else
        {
            // Code for deserializing.
            std::string s;
            ar & s;
            std::istringstream is(s);
            is >> dt;
        }
    }

}


RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(boost::gregorian::date, Echo, const boost::gregorian::date &)
RCF_END(I_Echo)
```

Binary data

To send a chunk of binary data, you can use the `RCF::ByteBuffer` class:

```
RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(RCF::ByteBuffer, Echo, RCF::ByteBuffer)
RCF_END(I_Echo)

class EchoImpl
{
public:
    RCF::ByteBuffer Echo(RCF::ByteBuffer byteBuffer)
    {
        return byteBuffer;
    }
};


int main()
{
    // Set max message length to 600 kb.
    RCF::setDefaultMaxMessageLength(600*1024);

    EchoImpl echoImpl;
    RCF::RcfServer server( RCF::TcpEndpoint(0));
    server.bind<I_Echo>(echoImpl);
    server.start();

    int port = server.getIpServerTransport().getPort();

    // Create and fill a 500 kb byte buffer.
    RCF::ByteBuffer byteBuffer(500*1024);
    for (std::size_t i=0; i<byteBuffer.getLength(); ++i)
    {
        byteBuffer.getPtr()[i] = char(i % 256);
    }

    RcfClient<I_Echo> client(( RCF::TcpEndpoint(port) ));

    // Echo it.
    RCF::ByteBuffer byteBuffer2 = client.Echo(byteBuffer);

    return 0;
}
```

`std::string` or `std::vector<char>` could be used for the same purpose. However, serialization and marshaling of `RCF::ByteBuffer` is significantly more efficient (see Performance). In particular, with `RCF::ByteBuffer`, no copies at all will be made of the data, on either end of the wire.

Portability

Because C++ does not prescribe the sizes of its fundamental types, there is potential for serialization errors when servers and clients are deployed on different platforms. For example, consider the following interface:

```
RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(std::size_t, Echo, std::size_t)
RCF_END(I_Echo)
```

The `Echo()` method uses `std::size_t` as a parameter and return value. Unfortunately, `std::size_t` has different meanings on different platforms. For example, the 32 bit Visual C++ compiler considers `std::size_t` to be a 32 bit type, while the 64 bit

Visual C++ compiler considers `std::size_t` to be a 64 bit type. If a remote call is made from a 32 bit client to a 64 bit server, with `std::size_t` arguments, a runtime serialization error will be raised.

The same issue arises when using the type `long` with 32 and 64 bit versions of gcc. 32-bit gcc considers `long` to be a 32 bit type, while 64-bit gcc considers `long` to be a 64-bit type.

The correct approach in these situations is to use typedefs for arithmetic types whose bit sizes are guaranteed. In particular, `<boost/cstdint.hpp>` provides a number of useful typedefs, including the following.

Table 1. Portable integral types

Type	Description
`boost::int16_t`	16 bit signed integer
`boost::uint16_t`	16 bit unsigned integer
`boost::int32_t`	32 bit signed integer
`boost::uint32_t`	32 bit unsigned integer
`boost::int64_t`	64 bit signed integer
`boost::uint64_t`	64 bit unsigned integer

To ensure portability, the example with `std::size_t`, above, should be rewritten as:

```
RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(boost::uint32_t, Echo, boost::uint32_t)
    RCF_METHOD_R1(boost::uint64_t, Echo, boost::uint64_t)
RCF_END(I_Echo)
```

File-based serialization

SF can be used independently of RCF, for instance to serialize objects to and from files. To do so, use `SF::OBinaryStream` and `SF::IBinaryStream`:

```
std::string filename = "data.bin";

X x1;
X x2;

// Write x1 to a file.
{
    std::ofstream fout(filename.c_str(), std::ios::binary);
    SF::OBinaryStream os(fout);
    os << x1;
}

// Read x2 from a file.
{

    std::ifstream fin(filename.c_str(), std::ios::binary);
    SF::IBinaryStream is(fin);
    is >> x2;
}
```

For more advanced topics see Advanced Serialization.

Transports

Transport access

Server and client transports are responsible for the actual transmission and reception of network messages. A RcfServer will have one or more server transports, while a client will have a single client transport.

To access the server transports of a `RcfServer`, you need to capture the return value of `RcfServer::addEndpoint()`:

```
RCF::RcfServer server;

RCF::ServerTransport & serverTransportTcp = server.addEndpoint(
    RCF::TcpEndpoint(50001) );

RCF::ServerTransport & serverTransportUdp = server.addEndpoint(
    RCF::UdpEndpoint(50002) );

server.start();
```

Alternatively, if the `RcfServer` only has a single server transport, you can access it by calling `RcfServer::getServerTransport()`:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
RCF::ServerTransport & serverTransport = server.getServerTransport();
```

On the client side, the client transport is availablable through `ClientStub::getTransport()`:

```
RcfClient<I_Echo> client( RCF::TcpEndpoint(50001) );
RCF::ClientTransport & clientTransport = client.getClientStub().getTransport();
```

Transport configuration

Maximum incoming message lengths

For server-side transports, it is generally necessary to set an upper limit on the size of incoming network messages. Without an upper limit, it is possible for malformed requests to cause arbitrarily sized memory allocations on the server.

The maximum incoming message length setting of a RCF server transport defaults to 1 Mb, and can be changed by calling `I_ServerTransport::setMaxIncomingMessageLength()` :

```
RCF::RcfServer server( RCF::TcpEndpoint(0) );

// Set max message length to 5 Mb.
server.getServerTransport().setMaxIncomingMessageLength(5*1024*1024);
```

Similarly, there is a maximum incoming message length setting for client transports. It defaults to 1 Mb and can be changed by calling `I_ClientTransport::setMaxIncomingMessageLength()` :

```
RcfClient<I_Echo> client( RCF::TcpEndpoint(123) );

// Set max message length to 5 Mb.
client.getClientStub().getTransport().setMaxIncomingMessageLength(5*1024*1024);
```

As client transports only receive network messages from a peer they have connected to, the risk of malformed packets is not as great as for server transports.

It is possible to query a `RcfClient<>` for the sizes of the latest request and response messages sent:

```
RcfClient<I_Echo> client( RCF::TcpEndpoint(50001) );
client.Echo("1234");

// Retrieve request and response size of the previous call.
RCF::ClientTransport & transport = client.getClientStub().getTransport();

std::size_t requestSize = transport.getLastRequestSize();
std::size_t responseSize = transport.getLastResponseSize();
```

Connection limits

To set the maximum number of simultaneous connections to a RCF server transport:

```
RCF::RcfServer server( RCF::TcpEndpoint(0) );

// Allow at most 100 clients to be connected at any time.
server.getServerTransport().setConnectionLimit(100);
```

This setting is not applicable to the UDP server transport, as the UDP protocol doesn't support a connection concept.

Automatic IP port selection

For IP-based server transports, you can allow the local system to assign a port number automatically, by specifying 0 as the port number. When the server starts, the system will find a free port and assign it to the server. The port number can subseqently be retrieved through `I_IpServerTransport::getPort()`:

```
RCF::RcfServer server( RCF::TcpEndpoint(0) );
server.start();

int port = server.getIpServerTransport().getPort();
RcfClient<I_Echo> client(( RCF::TcpEndpoint(port) ));
```

IP-based access rules

For IP-based server transports, client access can be allowed or denied, based on the IP addresses of the clients.

To configure IP rules for allowing clients:

```
RCF::RcfServer server( RCF::TcpEndpoint(0) );

RCF::IpServerTransport & ipTransport =
    dynamic_cast<RCF::IpServerTransport &>(server.getServerTransport());

std::vector<RCF::IpRule> rules;

// Match 11.22.33.* (24 significant bits).
rules.push_back( RCF::IpRule( RCF::IpAddress("11.22.33.0"), 24) );

ipTransport.setAllowIps(rules);

server.start();

// Access will be granted to clients connecting from IP addresses matching 11.22.33.* .
// All other clients will be denied.
```

To configure IP rules for denying clients:

```
RCF::RcfServer server( RCF::TcpEndpoint(0) );

RCF::IpServerTransport & ipTransport =
    dynamic_cast<RCF::IpServerTransport &>(server.getServerTransport());

std::vector<RCF::IpRule> rules;

// Match 11.*.*.* (8 significant bits).
rules.push_back( RCF::IpRule( RCF::IpAddress("11.0.0.0"), 8) );

// Match 12.22.*.* (16 significant bits).
rules.push_back( RCF::IpRule( RCF::IpAddress("12.22.0.0"), 16) );

ipTransport.setDenyIps(rules);

server.start();

// Access will be denied to clients connecting from IP addresses matching 11.*.*.*  and 12.22.*.* .
// All other clients will be allowed.
```

IPv4/IPv6

RCF supports both IPv4 and IPv6. To enable IPv6 support in RCF, `RCF_USE_IPV6` must be defined (see Building).

For example, to run a server and client over a loopback IPv4 connection:

```
// Specifying an explicit IPv4 address to bind to.
RCF::RcfServer server( RCF::TcpEndpoint("127.0.0.1", 50001) );
server.start();

// Specifying an explicit IPv4 address to bind to.
RcfClient<I_Echo> client( RCF::TcpEndpoint("127.0.0.1", 50001) );
```

To run a server and client over a loopback IPv6 connection instead, specify `::1` instead of `127.0.0.1`:

```
// Specifying an explicit IPv6 address to bind to.
RCF::RcfServer server( RCF::TcpEndpoint("::1", 50001) );
server.start();

// Specifying an explicit IPv6 address to bind to.
RcfClient<I_Echo> client( RCF::TcpEndpoint("::1", 50001) );
```

RCF uses the POSIX `getaddrinfo()` function to resolve IP addresses. `getaddrinfo()` can return either IPv4 or IPv6 addresses, depending on the configuration of the local system and network. So the following client will use either IPv4 or IPv6, depending on how the local system and network have been configured:

```
// Will resolve to either IPv4 or IPv6, depending on what the system
// resolves machine.domain to.

RcfClient<I_Echo> client( RCF::TcpEndpoint("machine.domain", 50001) );
```

You can force IPv4 or IPv6 resolution by using the `IpAddressV4` and `IpAddressV6` classes:

```
// Force IPv4 address resolution.
RCF::IpAddressV4 addr_4("machine.domain", 50001);
RcfClient<I_Echo> client_4(( RCF::TcpEndpoint(addr_4) ));

// Force IPv6 address resolution.
RCF::IpAddressV6 addr_6("machine.domain", 50001);
RcfClient<I_Echo> client_6(( RCF::TcpEndpoint(addr_6) ));
```

On machines with dual IPv4/IPv6 stacks, you will probably want your server to listen on both IPv4 and IPv6 addresses. You can do this portably by listening on both `0.0.0.0` and `::0`:

```
// Listen on port 50001, on both IPv4 and IPv6.
RCF::RcfServer server;
server.addEndpoint( RCF::TcpEndpoint("0.0.0.0", 50001) );
server.addEndpoint( RCF::TcpEndpoint("::0", 50001) );
server.start();
```

On some platforms, it is sufficient to listen only on `::0`, as the system will translate incoming IPv4 connections into IPv6 connections with a special class of IPv6 addresses.

Local address bindings for clients

When a `RcfClient<>` connects to a server over an IP-based transport, the default behavior is to allow the system to decide which local network interface and port to use. In some circumstances, you may want to explicitly set the local network interface a client should bind to. This is done by calling `I_IpServerTransport::setLocalIp()`, before connecting:

```
RcfClient<I_Echo> client( RCF::TcpEndpoint("127.0.0.1", 50001) );

RCF::IpClientTransport & ipTransport =
    client.getClientStub().getIpTransport();

// Force client to bind to a particular local network interface (127.0.0.1).
ipTransport.setLocalIp( RCF::IpAddress("127.0.0.1", 0) );

client.getClientStub().connect();
```

After a `RcfClient<>` has connected, you can determine which local network interface and port it is bound to, by calling `I_IpServerTransport::getAssignedLocalIp()`:

```
RcfClient<I_Echo> client( RCF::TcpEndpoint("127.0.0.1", 50001) );

RCF::IpClientTransport & ipTransport =
    client.getClientStub().getIpTransport();

client.getClientStub().connect();

// Find out which local network interface the client is bound to.
RCF::IpAddress localIp = ipTransport.getAssignedLocalIp();
std::string localInterface = localIp.getIp();
int localPort = localIp.getPort();
```

Socket level access

RCF provides access to the underlying OS primitives (sockets, handles) of client and server transports. For example:

```
// Client-side.

RcfClient<I_Echo> client( RCF::TcpEndpoint("127.0.0.1", 50001) );
client.getClientStub().connect();

RCF::TcpClientTransport & tcpClientTransport =
    dynamic_cast<RCF::TcpClientTransport &>(
        client.getClientStub().getTransport() );

// Obtain client socket handle.
int sock = tcpClientTransport.getNativeHandle();
```

```
// Server-side.

RCF::SessionState & sessionState = RCF::getCurrentRcfSession().getSessionState();

RCF::TcpAsioSessionState & tcpSessionState =
    dynamic_cast<RCF::TcpAsioSessionState &>(sessionState);

// Obtain server socket handle.
int sock = tcpSessionState.getNativeHandle();
```

This can be useful if you need to set custom socket options, for instance.

Transport implementations

TCP

TCP endpoints are represented in RCF by the `RCF::TcpEndpoint` class, constructed from an IP address and a port number.

```
RCF::RcfServer server( RCF::TcpEndpoint("0.0.0.0", 50001) );
server.start();

RcfClient<I_Echo> client( RCF::TcpEndpoint("127.0.0.1", 50001) );
```

Server transports interpret the IP address as the local network interface to listen on. So for example `"0.0.0.0"` should be specified in order to listen on all available IPv4 network interfaces, and `"127.0.0.1"` should be specified to listen only on the loopback IPv4 interface. If no IP address is specified, `"127.0.0.1"` is assumed.

UDP

Like `RCF::TcpEndpoint`, `RCF::UdpEndpoint` is constructed from an IP address and a port.

```
RCF::RcfServer server( RCF::UdpEndpoint("0.0.0.0", 50001) );
server.start();

RcfClient<I_Echo> client( RCF::UdpEndpoint("127.0.0.1", 50001) );
```

However, `RCF::UdpEndpoint` also contains some extra functionality, to deal with multicasting and broadcasting.

Multicasting

`RCF::UdpEndpoint` can be configured to listen on a multicast IP address:

```
// Listen on multicast address 232.5.5.5, on port 50001, on all network interfaces.
RCF::RcfServer server(
    RCF::UdpEndpoint("0.0.0.0", 50001).listenOnMulticast("232.5.5.5"));

server.start();
```

Note that the server still needs to specify a local network interface to listen on.

To send multicast messages, specify a multicast IP address and port when creating the client:

```
RcfClient<I_Echo> client( RCF::UdpEndpoint("232.5.5.5", 50001) );
client.Echo(RCF::Oneway, "ping");
```

Broadcasting

To send broadcast messages, specify a broadcast IP address and port:

```
RcfClient<I_Echo> client( RCF::UdpEndpoint("255.255.255.255", 50001) );
client.Echo(RCF::Oneway, "ping");
```

Address sharing

RCF's UDP server transport can be configured to share its address binding, so that multiple `RcfServer`'s can listen on the same port of the same interface. This is enabled by default when listening on multicast addresses, but can also be enabled when listening on non-multicast addresses. This can be useful if multiple processes on the same machine need to listen to the same broadcasts:

```
EchoImpl echoImpl;

RCF::RcfServer server1(
    RCF::UdpEndpoint("0.0.0.0", 50001).enableSharedAddressBinding() );

server1.bind<I_Echo>(echoImpl);
server1.start();

RCF::RcfServer server2(
    RCF::UdpEndpoint("0.0.0.0", 50001).enableSharedAddressBinding());

server2.bind<I_Echo>(echoImpl);
server2.start();

// This broadcast message will be received by both servers.
RcfClient<I_Echo> client( RCF::UdpEndpoint("255.255.255.255", 50001) );
client.Echo(RCF::Oneway, "ping");
```

Server discovery

In situations where servers are started on dynamically assigned ports, multicasting and broadcasting can be a useful means of communicating server IP addresses and ports to clients. For example:

```
// Interface for broadcasting port number of a TCP server.
RCF_BEGIN(I_Broadcast, "I_Broadcast")
    RCF_METHOD_V1(void, ServerIsRunningOnPort, int)
RCF_END(I_Broadcast)

// Implementation class for receiving I_Broadcast messages.
class BroadcastImpl
{
public:
    BroadcastImpl() : mPort()
    {}
    void ServerIsRunningOnPort(int port)
    {
        mPort = port;
    }
    int mPort;
};

// A server thread runs this function, to broadcast the server location once
// per second.
void broadcastThread(int port, const std::string &multicastIp, int multicastPort)
{
    RcfClient<I_Broadcast> client(
        RCF::UdpEndpoint(multicastIp, multicastPort) );

    client.getClientStub().setRemoteCallSemantics(RCF::Oneway);

    // Broadcast 1 message per second.
    while (true)
    {
        client.ServerIsRunningOnPort(port);
        RCF::sleepMs(1000);
    }
}
```

```
// ***** Server side ****

// Start a server on a dynamically assigned port.
EchoImpl echoImpl;
RCF::RcfServer server( RCF::TcpEndpoint(0));
server.bind<I_Echo>(echoImpl);
server.start();

// Retrieve the port number.
int port = server.getIpServerTransport().getPort();

// Start broadcasting the port number.
RCF::ThreadPtr broadcastThreadPtr( new RCF::Thread(
    boost::bind(&broadcastThread, port, "232.5.5.5", 50001)));

// ***** Client side ****

// Clients will listen for the broadcasts before doing anything else.
BroadcastImpl broadcastImpl;
RCF::RcfServer clientSideBroadcastListener(
    RCF::UdpEndpoint("0.0.0.0", 50001).listenOnMulticast("232.5.5.5"));

clientSideBroadcastListener.bind<I_Broadcast>(broadcastImpl);
clientSideBroadcastListener.start();

// Wait for a broadcast message.
while (!broadcastImpl.mPort)
{
    RCF::sleepMs(1000);
}

// Once the clients know the port number, they can connect.
RcfClient<I_Echo> client( RCF::TcpEndpoint(broadcastImpl.mPort));
client.Echo("asdf");
```

Note that here we are actually using a multicast address to broadcast information to clients. If multicasting had been unavailable on this particular network, we could also have used an IP broadcast address instead of an IP multicast address.

Win32 named pipes

RCF supports Win32 named pipe transports. `RCF::Win32NamedPipeEndpoint` takes one constructor parameter, which is the name of the named pipe, with or without the leading `\\.\pipe\` prefix.

```
RCF::RcfServer server( RCF::Win32NamedPipeEndpoint("MyPipe") );
server.start();

RcfClient<I_Echo> client( RCF::Win32NamedPipeEndpoint("MyPipe") );
```

An advantage of using Win32 named pipes, is that they allow easy authentication of clients. A server using a Win32 named pipe server transport can authenticate its clients through the `RCF::Win32NamedPipeImpersonator` class, which uses the Windows API function `ImpersonateNamedPipeClient()` to impersonate the client:

```
RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(std::string, Echo, const std::string &)
RCF_END(I_Echo)

class EchoImpl
{
public:
    std::string Echo(const std::string & s)
    {
        // Impersonate client.
        RCF::Win32NamedPipeImpersonator impersonator;
        std::cout << "Client user name: " << RCF::getMyUserName();
        return s;
    }
};
```

If we had used a TCP connection to `127.0.0.1` instead, we would have needed to enable Kerberos or NTLM authentication to securely determine the clients user name (see Transport protocols).

UNIX domain sockets

UNIX domain sockets are the UNIX analogue of Win32 named pipes, and allow efficient communication between servers and clients on the same machine. `RCF::UnixLocalEndpoint` takes one parameter, which is the name of the UNIX domain socket. The name must be a valid filesystem path. For servers, the program must have sufficient privilege to create the given path, and the file must not already exist. For clients, the program must have sufficient privilege to access the given path.

Here is a simple example:

```
RCF::RcfServer server( RCF::UnixLocalEndpoint("/home/xyz/MySocket"));
server.start();

RcfClient<I_Echo> client( RCF::UnixLocalEndpoint("/home/xyz/MySocket"));
```

Transport Protocols

Transport protocols are used to transform the data passing over a transport. RCF uses transport protocols to provide authentication, encryption and compression for remote calls.

RCF currently supports NTLM, Kerberos, Negotiate and SSL transport protocols. NTLM, Kerberos and Negotiate are only supported on Windows platforms, while SSL is supported on all platforms.

In addition, RCF also supports zlib-based compression of remote calls.

Transport protocols are assigned to a client connection by calling `ClientStub::setTransportProtocol()`:

```
RcfClient<I_HelloWorld> client(( RCF::TcpEndpoint(port) ));
client.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
```

From within the server session of a client connection, you can call `RcfSession::getTransportProtocol()` to determine the transport prococol the client is using:

```
RCF::RcfSession & session = RCF::getCurrentRcfSession();
RCF::TransportProtocol tp = session.getTransportProtocol();
```

NTLM

To configure NTLM on a client connection:

```
RcfClient<I_HelloWorld> client(( RCF::TcpEndpoint(port) ));
client.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
client.Print("Hello World");
```

On the server-side, you can determine the Windows username of the client, and impersonate them:

```
std::string clientUsername = session.getClientUsername();

RCF::SspiImpersonator impersonator(session);
// We are now impersonating the client, until impersonator goes out of scope.
// ...
```

Kerberos

To configure Kerberos on a client connection:

```
client.getClientStub().setTransportProtocol(RCF::Tp_Kerberos);
client.getClientStub().setKerberosSpn("Domain\\ServerAccount");
client.Print("Hello World");
```

Notice that the client needs to call `ClientStub::setKerberosSpn()` to specify the username it expects the server to be running under. This is known as the SPN (Service Principal Name) of the server, and is used by Kerberos to implement mutual authentication. If the server is not running under this account, the connection will fail.

On the server-side, you can determine the Windows username of the client, and impersonate them:

```
std::string clientUsername = session.getClientUsername();

RCF::SspiImpersonator impersonator(session);
// We are now impersonating the client, until impersonator goes out of scope.
// ...
```

Negotiate

Negotiate is a negotiation protocol between the NTLM and Kerberos protocols. It will resolve to Kerberos if possible, and otherwise fall back to NTLM.

To configure Negotiate on a client connection:

```
client.getClientStub().setTransportProtocol(RCF::Tp_Negotiate);
client.getClientStub().setKerberosSpn("Domain\\ServerAccount");
client.Print("Hello World");
```

As with the Kerberos transport protocol, you need to supply a SPN for the server.

On the server-side, you can determine the Windows username of the client, and impersonate them:

```
std::string clientUsername = session.getClientUsername();

RCF::SspiImpersonator impersonator(session);
// We are now impersonating the client, until impersonator goes out of scope.
// ...
```

SSL

RCF offers two SSL transport protocol implementations. One is based on the cross-platform OpenSSL library, and the other is based on the Windows-only Schannel package.

OpenSSL support is only available if `RCF_USE_OPENSSL` has been defined. Schannel support is only available in Windows builds.

If you define `RCF_USE_OPENSSL` in a Windows build, RCF will use OpenSSL rather than Schannel. Should you want to use Schannel, despite defining `RCF_USE_OPENSSL`, you can set the SSL implementation for individual servers and clients using the `RcfServer::setSslImplementation()` and `ClientStub::setSslImplementation()` functions, or for the whole RCF runtime using the `RCF::setSslImplementation()` function.

The SSL protocol uses certificates to authenticate servers to clients (and optionally clients to servers). Certificate and certificate validation functionality is handled differently by the two SSL transport protocol implemenations, as described below.

Schannel

To configure a server to accept SSL connections, you need to provide a server certificate. RCF provides the `RCF::PfxCertificate` class, to load certificates from .pfx and .p12 files:

```
RCF::CertificatePtr serverCertPtr( new RCF::PfxCertificate(
    "C:\\serverCert.p12",
    "password",
    "CertificateName") );

server.setCertificate(serverCertPtr);
```

RCF also provides the `RCF::StoreCertificate` class, to load certificates from Windows certificate stores:

```
RCF::CertificatePtr serverCertPtr( new RCF::StoreCertificate(
    RCF::Cl_LocalMachine,
    RCF::Cs_My,
    "CertificateName") );

server.setCertificate(serverCertPtr);
```

On the client, you need to provide a means of validating the certificate presented by the server. There are several ways of doing this. You can let the Schannel package apply its own internal validation logic, which will defer to the locally installed certificate authorities:

```
client.getClientStub().setTransportProtocol(RCF::Tp_Ssl);
client.getClientStub().setEnableSchannelCertificateValidation("CertificateName");
client.Print("Hello World");
```

You can also provide a particular certificate authority yourself, which will be used to validate the server certificate:

```
RCF::CertificatePtr caCertPtr( new RCF::PfxCertificate(
    "C:\\clientCaCertificate.p12",
    "password",
    "CaCertificatename"));

client.getClientStub().setCaCertificate(caCertPtr);
```

You can also write your own custom certificate validation logic:

```
bool schannelValidateCert(RCF::Certificate * pCert)
{
    RCF::Win32Certificate * pWin32Cert = static_cast<RCF::Win32Certificate *>(pCert);
    if (pWin32Cert)
    {
        RCF::tstring certName = pWin32Cert->getCertificateName();
        RCF::tstring issuerName = pWin32Cert->getIssuerName();

        PCCERT_CONTEXT pContext = pWin32Cert->getWin32Context();

        // Custom code to inspect and validate certificate.
        // ...
    }

    // Return true if the certificate is considered valid. Otherwise, return false,
    // or throw an exception.
    return true;
}
```

```
client.getClientStub().setCertificateValidationCallback(&schannelValidateCert);
```

RCF clients can also be configured to present a certificate to the server:

```
RCF::CertificatePtr clientCertPtr( new RCF::PfxCertificate(
    "C:\\clientCert.p12",
    "password",
    "CertificateName") );

client.getClientStub().setCertificate(clientCertPtr);
```

Server-side certificate validation is done in the same way as client-side certificate validation, but using the `RcfServer::setSchannelDefaultCertificateValidation()`, `RcfServer::setSchannelCertificateValidationCb()`, and `RcfServer::setSslCaCertificate()` functions.

OpenSSL

When using the OpenSSL-based SSL transport protocol, certificates need to be loaded from .pem files, using the `RCF::PemCertificate` class. Here is an example of providing a server certificate:

```
RCF::CertificatePtr serverCertPtr( new RCF::PemCertificate(
    "C:\\serverCert.pem",
    "password") );

server.setCertificate(serverCertPtr);
```

The client can validate the server certificate in two ways. It can provide a certificate authority certificate:

```
RCF::CertificatePtr caCertPtr( new RCF::PemCertificate(
    "C:\\clientCaCertificate.pem",
    "password"));

client.getClientStub().setCaCertificate(caCertPtr);
```

, or it can provide custom validation logic in a callback function:

```
bool opensslValidateCert(RCF::Certificate * pCert)
{
    RCF::X509Certificate * pX509Cert = static_cast<RCF::X509Certificate *>(pCert);

    if (pX509Cert)
    {
        std::string certName = pX509Cert->getCertificateName();
        std::string issuerName = pX509Cert->getIssuerName();

        X509 * pX509 = pX509Cert->getX509();

        // Custom code to inspect and validate certificate.
        // ...
    }

    // Return true if valid, false if not.
    return true;
}
```

```
client.getClientStub().setCertificateValidationCallback(&opensslValidateCert);
```

The client can also provide a certificate of its own, to present to the server:

```
RCF::CertificatePtr clientCertPtr( new RCF::PemCertificate(
    "C:\\clientCert.pem",
    "password") );

client.getClientStub().setCertificate(clientCertPtr);
```

Server-side certificate validation is done in the same way as client-side certificate validation, but using the `RcfServer::setOpenSslCertificateValidationCb()`, and `RcfServer::setSslCaCertificate()` functions.

Compression

RCF supports compression of remote calls. To build RCF with support for compression, define `RCF_USE_ZLIB` (see Building).

Compression is configured independently of other transport protocols, using `ClientStub::setEnableCompression()`:

```
RcfClient<I_HelloWorld> client(( RCF::TcpEndpoint(port) ));
client.getClientStub().setEnableCompression(true);
client.Print("Hello World");
```

RCF applies the compression stage before the transport protocol stage. For example, if you configure compression and NTLM on the same connection:

```
RcfClient<I_HelloWorld> client(( RCF::TcpEndpoint(port) ));
client.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
client.getClientStub().setEnableCompression(true);
client.Print("Hello World");
```

, the remote call data will be compressed first, before being encrypted and sent across the wire.

HTTP/HTTPS Tunneling

RCF supports tunneling remote calls over the HTTP and HTTPS protocols. In particular, remote calls can be directed through HTTP and HTTPS proxies.

HTTPS is essentially the HTTP protocol layered on top of the SSL protocol. As such, configuration of the SSL aspects of HTTPS, is done in the same way as for the SSL transport protocol (see Transport protocols).

Server-side

To setup a server with an HTTP endpoint, use `RCF::HttpEndpoint`:

```
RCF::RcfServer server( RCF::HttpEndpoint("0.0.0.0", 80) );
HelloWorldImpl helloWorldImpl;
server.bind<I_HelloWorld>(helloWorldImpl);
server.start();
```

Similarly, for an HTTPS endpoint, use `RCF::HttpsEndpoint`:

```
RCF::RcfServer server( RCF::HttpsEndpoint("0.0.0.0", 443) );
HelloWorldImpl helloWorldImpl;
server.bind<I_HelloWorld>(helloWorldImpl);

server.setCertificate( RCF::CertificatePtr( new RCF::PfxCertificate(
    "C:\\serverCert.p12",
    "password",
    "CertificateName") ) );

server.start();
```

Client-side

Client side configuration is similar, using `HttpEndpoint` for a HTTP client:

```
RcfClient<I_HelloWorld> client( RCF::HttpEndpoint("server.acme.com", 80) );
client.Print("Hello World");
```

, and `HttpsEndpoint` for a HTTPS client:

```
RcfClient<I_HelloWorld> client( RCF::HttpsEndpoint("server.acme.com", 443) );
client.getClientStub().setCertificateValidationCallback(&schannelValidate);
client.Print("Hello World");
```

Finally, to direct remote calls through a HTTP or HTTPS proxy, use the `ClientStub::setHttpProxy()` and `ClientStub::setHttpProxyPort()` functions:

```
client.getClientStub().setHttpProxy("proxy.acme.com");
client.getClientStub().setHttpProxyPort(8080);
client.Print("Hello World");
```

Server-side Access Controls

Servant binding access controls

RCF allows you to apply access controls to individual servant bindings on your server. The access control is implemented as a user-defined callback function, in which you can apply application-specific logic to determine whether a client connection should be allowed to access a particular servant binding.

For example:

```
bool onHelloWorldAccess(int dispatchId)
{
    // Return true to allow access, and false to deny access.
    // ...

    return true;
}
```

```
HelloWorldImpl helloWorldImpl;
RCF::RcfServer server( RCF::TcpEndpoint(50001) );

RCF::ServerBindingPtr bindingPtr = server.bind<I_HelloWorld>(helloWorldImpl);
bindingPtr->setAccessControl( boost::bind(&onHelloWorldAccess, _1) );

server.start();
```

The access control callback will be invoked by the `RcfServer` each time a client tries to call a method on that servant. From the access control callback you can inspect the current session and determine whether it should be granted access to the servant. Once authentication is granted, you will probably want to store the authentication state in a session object, so it can be easily reused on subsequent calls:

```
// App-specific authentication state.
class HelloWorldAuthenticationState
{
public:
    HelloWorldAuthenticationState() : mAuthenticated(false)
    {
    }

    bool            mAuthenticated;
    std::string     mClientUsername;
};

// Servant binding access control.
bool onHelloWorldAccess(int dispatchId)
{
    RCF::RcfSession & session = RCF::getCurrentRcfSession();
    HelloWorldAuthenticationState & authState = session.getSessionObject<HelloWorldAuthentica↲
tionState>(true);
    if (!authState.mAuthenticated)
    {
        // Here we are checking that the client is using either NTLM or Kerberos.
        RCF::TransportProtocol tp =  session.getTransportProtocol();
        if (tp == RCF::Tp_Ntlm || tp == RCF::Tp_Kerberos)
        {
            authState.mAuthenticated = true;
            authState.mClientUsername = session.getClientUsername();
        }
    }
    return authState.mAuthenticated;
}
```

As the previous example indicates, typically you would use the access control callback to inspect the transport protocol the client is using, and use that to determine the identity of the client.

In some situations you may want the client to provide extra authentication information, beyond what is available though the transport protocol. This typically means having the equivalent of a `Login()` method on the interface, that needs to be called before any other method on the interface. The access control callback can be used to verify that `Login()` is called before any other method:

```
// App-specific login info.
class LoginInfo
{
public:
    // ...

    void serialize(SF::Archive & ar)
    {}
};

RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Login, const LoginInfo &)
    RCF_METHOD_V1(void, Print, const std::string &)
RCF_END(I_HelloWorld)

// App-specific authentication state.
class HelloWorldAuthenticationState
{
public:
    HelloWorldAuthenticationState() : mAuthenticated(false)
    {
    }
    bool                mAuthenticated;
    std::string         mClientUsername;
    LoginInfo           mLoginInfo;
};

// Servant object.
class HelloWorldImpl
{
public:

    void Login(const LoginInfo & loginInfo)
    {
        RCF::RcfSession & session = RCF::getCurrentRcfSession();
       HelloWorldAuthenticationState & authState = session.getSessionObject<HelloWorldAuthentic↵
ationState>(true);
        if (!authState.mAuthenticated)
        {
            RCF::TransportProtocol tp =  session.getTransportProtocol();
            if (tp == RCF::Tp_Ntlm || tp == RCF::Tp_Kerberos)
            {
                authState.mAuthenticated = true;
                authState.mClientUsername = session.getClientUsername();
                authState.mLoginInfo = loginInfo;
            }
        }
    }

    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }
};

// Servant binding access control.
bool onHelloWorldAccess(int dispatchId)
{
    if (dispatchId == 0)
    {
        // Calls to Login() are always allowed through.
        return true;
    }
    else
    {
        RCF::RcfSession & session = RCF::getCurrentRcfSession();
       HelloWorldAuthenticationState & authState = session.getSessionObject<HelloWorldAuthentic↵
ationState>(true);
        return authState.mAuthenticated;
    }
}
```

In this case, the access control callback allows calls to `Login()` to go straight through, while for any other method on the `I_HelloWorld` interface, it checks for the existence of the authentication state that the `Login()` call creates.

The `Login()` method is identified in the access control callback by its dispatch ID - in this case 0, as it is the first method on the interface. Dispatch ID's are assigned in incremental order, from 0, so if for example `Login()` had been the third method on the interface, it would have had a dispatch ID of 2.

Asynchronous Remote Calls

Asynchronous remote call invocation

Asynchronous remote calls allow you to initate remote calls on one thread, and have them complete asynchronously on another thread.

Asynchronous remote calls are implemeted in RCF using the `RCF::Future<>` template. Given an RCF interface:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")

    // Returns number of characters printed.
    RCF_METHOD_R1(int, Print, const std::string &)

RCF_END(I_HelloWorld)
```

, an asynchronous call is made by specifying one or more `RCF::Future<>` arguments. For example:

```
RcfClient<I_HelloWorld> client(( RCF::TcpEndpoint(port) ));
RCF::Future<int> fRet = client.Print("Hello World");
```

The thread that executes this code will return immediately. It can subsequently poll for completion:

```
// Poll until remote call completes.
while (!fRet.ready())
{
    RCF::sleepMs(500);
}
```

, or it can wait for completion:

```
// Wait for remote call to complete.
fRet.wait();
```

Once the call is complete, the return values can be recovered from the relevant `Future<>` instances:

```
int charsPrinted = *fRet;
```

If the call resulted in an error, the error will be thrown when the `Future<>` instance is dereferenced, as above. Alternatively, the error can be retrieved by calling `Future<>::getAsyncException()`:

```
std::auto_ptr<RCF::Exception> ePtr = fRet.getAsyncException();
```

Instead of polling or waiting, the thread that initiates an asynchronous remote call can provide a completion callback that will be called by the RCF runtime, on a background thread, when the call completes:

```
typedef boost::shared_ptr< RcfClient<I_HelloWorld> > HelloWorldPtr;
void onCallCompleted(HelloWorldPtr client, RCF::Future<int> fRet)
{
    std::auto_ptr<RCF::Exception> ePtr = fRet.getAsyncException();
    if (ePtr.get())
    {
        // Deal with any exception.
        // ...
    }
    else
    {
        int charsPrinted = *fRet;
        // ...
    }
}
```

```
RCF::Future<int> fRet;
HelloWorldPtr client( new RcfClient<I_HelloWorld>(RCF::TcpEndpoint(port)) );
fRet = client->Print(
    RCF::AsyncTwoway( boost::bind(&onCallCompleted, client, fRet)),
    "Hello World");
```

Notice that the `Future<>` arguments are passed as arguments to the completion callback function. `Future<>` objects are internally reference counted, and can be copied freely, while still referring to the same underlying value.

An asynchronous call in progress can be canceled by disconnecting the client:

```
client.getClientStub().disconnect();
```

If a `RcfClient` is destroyed while an asynchronous call is in progress, the call is automatically disconnected and any asynchronous operations are canceled.

Asynchronous remote call dispatching

On the server-side, RCF will normally dispatch a remote call on the same server thread that receives the remote call request from the transport. Asynchronous dispatching allows you to instead transfer the remote call over to other threads, freeing up the RCF thread to process other remote calls.

The `RCF::RemoteCallContext<>` class is used to capture the server-side context of a remote call. `RemoteCallContext<>` objects can be copied into queues and stored for later execution on arbitrary application threads.

`RemoteCallContext<>` objects are created from within the corresponding servant implemenation method. Here is a non-asynchronously dispatched `Print()` method:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")

    // Returns number of characters printed.
    RCF_METHOD_R1(int, Print, const std::string &)

RCF_END(I_HelloWorld)

class HelloWorldImpl
{
public:
    int Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
        return s.length();
    }
};
```

To instead dispatch the call asynchronously, a `RemoteCallContext<>` object is created in `Print()`, with template parameters corresponding to the method signature:

```
class HelloWorldImpl
{
public:

    typedef RCF::RemoteCallContext<int, const std::string&> PrintContext;

    int Print(const std::string & s)
    {
        // Capture current remote call context.
        PrintContext printContext(RCF::getCurrentRcfSession());

        // Create a new thread to dispatch the remote call.
        RCF::ThreadPtr threadPtr( new RCF::Thread( boost::bind(
            &HelloWorldImpl::threadFunc,
            this,
            printContext) ) );

        return 0;
    }

    void threadFunc(PrintContext printContext)
    {
        const std::string & s = printContext.parameters().a1.get();
        std::cout << "I_HelloWorld service: " << s << std::endl;
        printContext.parameters().r.set( s.length() );
        printContext.commit();
    }
};
```

Once created, the `RemoteCallContext<>` can be stored and copied like any other C++ object. When the `Print()` function returns, RCF will not send a response to the client. The response will only be sent when `RemoteCallContext<>::commit()` is called.

`RemoteCallContext::parameters()` provides access to all the parameters of the remote call, including the return value. We can access in parameters:

```
const std::string & s = printContext.parameters().a1.get();
```

, and set out parameters (in this case the return value):

```
printContext.parameters().r.set( s.length() );
```

Publish/Subscribe

RCF provides a built in publish/subscribe implementation. Publishers can publish remote calls to a group of peer components (subscribers). Publishing servers can publish on any number of topics, and subscribers can pick which topics to subscribe to.

Publish/subscribe can be used in the presence of firewalls and NAT's. The only network topology requirement is that the subscriber must be able to initiate a network connection to the publisher. Publishers will never attempt to establish network connections to their subscribers.

Publishers

To create a publisher, use `RcfServer::createPublisher<>()`:

```
RCF::RcfServer pubServer( RCF::TcpEndpoint(50001) );
pubServer.start();

typedef RCF::Publisher<I_HelloWorld> HelloWorldPublisher;
typedef boost::shared_ptr< HelloWorldPublisher > HelloWorldPublisherPtr;
HelloWorldPublisherPtr publisherPtr = pubServer.createPublisher<I_HelloWorld>();
```

This will create a publisher with a default topic name. The default topic name is the runtime name of the RCF interface passed to `RcfServer::createPublisher<>()` (in this case `"I_HelloWorld"`).

The topic name can also be set manually. For example, to create two distinct publishers, using the same RCF interface:

```
RCF::PublisherParms pubParms;
pubParms.setTopicName("HelloWorld_Topic_1");
HelloWorldPublisherPtr publisher1Ptr = pubServer.createPublisher<I_HelloWorld>(pubParms);

pubParms.setTopicName("HelloWorld_Topic_2");
HelloWorldPublisherPtr publisher2Ptr = pubServer.createPublisher<I_HelloWorld>(pubParms);
```

The `Publisher<>` object returned by `RcfServer::createPublisher<>()` is used to publish remote calls. Published remote calls always have one-way semantics, and are received by all subscribers currently subscribing to that publishing topic:

```
publisherPtr->publish().Print("First published message.");
publisherPtr->publish().Print("Second published message.");
```

The publishing topic is closed, and all of its subscribers disconnected, when the `Publisher<>` object is destroyed, or when `Publisher<>::close()` is called.

```
// Close the publishing topic. All subscribers will be disconnected.
publisherPtr->close();
```

Subscribers

To subscribe to a publisher, use `RcfServer::createSubscription<>()`:

```
RCF::RcfServer subServer( RCF::TcpEndpoint(-1) );
subServer.start();

HelloWorldImpl helloWorldImpl;
RCF::SubscriptionParms subParms;
subParms.setPublisherEndpoint( RCF::TcpEndpoint(50001) );

RCF::SubscriptionPtr subscriptionPtr = subServer.createSubscription<I_HelloWorld>(
    helloWorldImpl,
    subParms);
```

The topic name defaults to the runtime name of the RCF interface (in this case `"I_HelloWorld"`). The topic name can also be specified manually:

```
HelloWorldImpl helloWorldImpl;

RCF::SubscriptionParms subParms;
subParms.setPublisherEndpoint( RCF::TcpEndpoint(50001) );
subParms.setTopicName("HelloWorld_Topic1");

RCF::SubscriptionPtr subscription1Ptr = subServer.createSubscription<I_HelloWorld>(
    helloWorldImpl,
    subParms);

subParms.setTopicName("HelloWorld_Topic2");

RCF::SubscriptionPtr subscription2Ptr = subServer.createSubscription<I_HelloWorld>(
    helloWorldImpl,
    subParms);
```

The first parameter passed to `createSubscription<>()` is the object which will receive the published messages. It is the applications responsibility to make sure this object is not destroyed while the subscription is still connected.

A disconnect callback can be provided, which will be called if the subscriber is disconnected:

```
void onSubscriptionDisconnected(RCF::RcfSession & session)
{
    // Handle subscription disconnection here.
    // ...
}
```

```
HelloWorldImpl helloWorldImpl;

RCF::SubscriptionParms subParms;
subParms.setPublisherEndpoint( RCF::TcpEndpoint(50001) );
subParms.setOnSubscriptionDisconnect(&onSubscriptionDisconnected);

RCF::SubscriptionPtr subscriptionPtr = subServer.createSubscription<I_HelloWorld>(
    helloWorldImpl,
    subParms);
```

To terminate a subscription, destroy the `Subscription` object, or call `Subscription::close()`:

```
subscriptionPtr->close();
```

Access control

Access controls can be applied to publishers, in the form of an access control callback which will be called for each subscriber attempting to subscribe to the publisher. Similarly to servant binding access controls, publisher access controls can be used to inspect the `RcfSession` of the subscriber connections for any relevant authentication information:

```
bool onSubscriberConnect(RCF::RcfSession & session, const std::string & topicName)
{
    // Return true to allow access, false otherwise.
    // ...

    return true;
}

void onSubscriberDisconnect(RCF::RcfSession & session, const std::string & topicName)
{
    // ...
}
```

```
RCF::PublisherParms pubParms;
pubParms.setOnSubscriberConnect(onSubscriberConnect);
pubParms.setOnSubscriberDisconnect(onSubscriberDisconnect);
HelloWorldPublisherPtr publisher1Ptr = pubServer.createPublisher<I_HelloWorld>(pubParms);
```

Callback Connections

Creating callback connections

Usually remote calls are made from a client to a server. RCF also implements a callback connection concept, allowing the server to take over a connection and use it to make remote calls back to the client.

Remote calls over callback connections function just like remote calls over regular connections, except that callback connections cannot be automatically re-connected.

On the client-side, to create a callback connection to receive calls on, use `RCF::createCallbackConnection()`:

```
// Client-side - start a callback server.
RCF::RcfServer callbackServer( RCF::TcpEndpoint(0) );
HelloWorldImpl helloWorldImpl;
callbackServer.bind<I_HelloWorld>(helloWorldImpl);
callbackServer.start();

// Client-side - create callback connection.
RcfClient<I_HelloWorld> client(( RCF::TcpEndpoint(port) ));
RCF::createCallbackConnection(client, callbackServer);
```

Note that you need a callback server to receive calls through, and a `RcfClient<>` instance with which to connect to the remote server.

On the server-side, use `RcfServer::setOnCallbackConnectionCreated()` to receive notification when a callback connection is created, and take control of the callback client transport:

```
// Server-side - taking control of callback connections.
RCF::Mutex gCallbackTransportMutex;
RCF::ClientTransportAutoPtr gCallbackTransportPtr;

void onCallbackConnectionCreated(
    RCF::RcfSessionPtr sessionPtr,
    RCF::ClientTransportAutoPtr clientTransportPtr)
{
    // Store the callback client transport in a global variable for later use.
    RCF::Lock lock(gCallbackTransportMutex);
    gCallbackTransportPtr = clientTransportPtr;
}
```

```
RCF::RcfServer server( RCF::TcpEndpoint(0) );
server.setOnCallbackConnectionCreated(onCallbackConnectionCreated);
server.start();
```

Once you have the callback client transport, you can use it to construct an appropriate `RcfClient<>` instance, and subsequently use that to make calls back to the client:

```
// Server-side - wait for callback connection to be created.
while (true)
{
    RCF::sleepMs(1000);
    RCF::Lock lock(gCallbackTransportMutex);
    if (gCallbackTransportPtr.get())
    {
        RcfClient<I_HelloWorld> callbackClient( gCallbackTransportPtr );
        callbackClient.Print("Hello World");
        break;
    }
}
```

Identifying clients

Typically you will want a server to maintain a list of callback connections to clients, and make calls back to specific clients. To do this, the server needs a mechanism by which to identify the callback connections. The best way to do this is to have the client set some session-specific data, before it calls `RCF::createCallbackConnection()`.

Let's assume you want to identify your clients with a string - for example a name. Before the client calls `RCF::createCallback-Connection()`, it should call an application-defined remote method on the server, to set the name of the client:

```
RCF_BEGIN(I_IdentifyClient, "I_IdentifyClient")
    RCF_METHOD_V1(void, SetClientName, const std::string&)
RCF_END(I_IdentifyClient)
```

```
// Client-side - create callback connection.
RcfClient<I_IdentifyClient> client(( RCF::TcpEndpoint(port) ));
client.SetClientName("ABC-123");
RCF::createCallbackConnection(client, callbackServer);
```

The server implementation of `SetClientName()` creates a session object in the `RcfSession` of the connection, with the client name:

```
class IdentifyClientImpl
{
public:
    void SetClientName(const std::string& clientName)
    {
        RCF::RcfSession & session = RCF::getCurrentRcfSession();
        std::string & sessionClientName = session.createSessionObject<std::string>();
        sessionClientName = clientName;
    }
};
```

When the client subsequently calls `RCF::createCallbackConnection()`, the `RcfServer` can then retrieve the name of the client and associate the callback connection with that name:

```
// Server-side - taking control of callback connections.
RCF::Mutex                                          gCallbackClientMapMutex;

typedef RcfClient<I_HelloWorld>                     HelloWorldClient;
typedef boost::shared_ptr<HelloWorldClient>         HelloWorldClientPtr;
std::map<std::string, HelloWorldClientPtr>          gCallbackClientMap;

void onCallbackConnectionCreated(
    RCF::RcfSessionPtr sessionPtr,
    RCF::ClientTransportAutoPtr clientTransportPtr)
{
    std::string & clientName = sessionPtr->getSessionObject<std::string>();

    // Store the callback client transport in a global variable for later use.
    RCF::Lock lock(gCallbackClientMapMutex);
    HelloWorldClientPtr hwClientPtr( new HelloWorldClient(clientTransportPtr) );
    gCallbackClientMap[clientName] = hwClientPtr;
}
```

Transport protocols

Transport protocols configured on the original client-to-server connection are not carried over to the callback connection. If you want to use a transport protocol on the callback connection, it needs to be configured explicitly:

```
RcfClient<I_HelloWorld> callbackClient( gCallbackTransportPtr );
callbackClient.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
callbackClient.Print("Hello World");
```

File Transfers

RCF has built-in support for transferring files and directories. While smaller files can be transferred easily in a single call (for example using `RCF::ByteBuffer` arguments), this technique fails once the size of the file exceeds the maximum message size of the connection.

To reliably transfer files of arbitrary size, the files need to be split into chunks and transmitted in separate calls, with the recipient reassembling the chunks to form the destination file. RCF automates this chunking logic in a secure and efficient way, using the `RCF::FileUpload` and `RCF::FileDownload` classes.

File uploads

RCF clients can upload files or directories to a server by including a `FileUpload` object as a remote call parameter:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
    RCF_METHOD_V1(void, UploadFiles, RCF::FileUpload)
RCF_END(I_HelloWorld)
```

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );
RCF::FileUpload fileUpload("C:\\FileToUpload.zip");
client.UploadFiles(fileUpload);
```

The serialization implementation of the `FileUpload` class takes care of transferring the nominated files to the server. On the server-side, the servant object that the remote call is dispatched to, uses the `FileUpload` object to locate the newly uploaded files on the local system:

```
class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }

    void UploadFiles(RCF::FileUpload fileUpload)
    {
        // Where are the uploaded files.
        std::string uploadPath = fileUpload.getLocalPath();

        // Which files have been uploaded.
        RCF::FileManifest & manifest = fileUpload.getManifest();
    }
};
```

If the connection to the server is lost during the transfer, the upload can be resumed by reusing the same `FileUpload` object:

```
// Call UploadFiles with the same arguments again, to resume an interrupted upload.
client.UploadFiles(fileUpload);
```

To monitor the progress of the upload, use `ClientStub::setFileProgressCallback()`:

```
void onFileTransferProgress(const RCF::FileTransferProgress & progress)
{
    std::cout << "Total bytes to transfer: " << progress.mBytesTotalToTransfer << std::endl;
    std::cout << "Bytes transferred so far: " << progress.mBytesTransferredSoFar << std::endl;
   std::cout << "Server-side bandwidth limit (if any): " << progress.mServerLimitBps << std::endl;
}
```

```
client.getClientStub().setFileProgressCallback(&onFileTransferProgress);
```

To encrypt the file upload, set the transport protocol of the `RcfClient<>` object, before commencing the upload. For instance, to use NTLM:

```
client.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
```

On the server, the default upload location is a subdirectory of the current working directory. You can use `RcfServer::setFileUploadDirectory()` to set a different location for file uploads:

```
server.setFileUploadDirectory("C:\\RCF-Uploads");
```

Once files are uploaded to the server, the lifetime of the uploaded files becomes the responsibility of the server application.

The server can also monitor the progress of all file uploads, as well as cancel file uploads:

```
void onServerFileUpload(RCF::RcfSession & session, const RCF::FileUploadInfo & uploadInfo)
{
    // Which files are being uploaded.
    const RCF::FileManifest& manifest = uploadInfo.mManifest;

    // How far has the upload progressed.
    boost::uint32_t currentFile = uploadInfo.mCurrentFile;
    boost::uint64_t currentFilePos = uploadInfo.mCurrentPos;

    // To cancel the upload, throw an exception.
    // ...
}
```

```
server.setOnFileUploadProgress(&onServerFileUpload);
```

File downloads

File downloads are implemented in RCF in a similary way to file uploads. RCF clients can download files or directories from a server by including a `FileDownload` object as a remote call parameter:

```
RCF_BEGIN(I_HelloWorld, "I_HelloWorld")
    RCF_METHOD_V1(void, Print, const std::string &)
    RCF_METHOD_V1(void, DownloadFiles, RCF::FileDownload)
RCF_END(I_HelloWorld)
```

```
RcfClient<I_HelloWorld> client( RCF::TcpEndpoint(50001) );

RCF::FileDownload fileDownload;
client.DownloadFiles(fileDownload);

std::string localFilePath = fileDownload.getLocalPath();
RCF::FileManifest & fileManifest = fileDownload.getManifest();
```

The servant object that the remote call is dispatched to, uses the `FileDownload` object to set the files that will be downloaded to the client:

```
class HelloWorldImpl
{
public:
    void Print(const std::string & s)
    {
        std::cout << "I_HelloWorld service: " << s << std::endl;
    }

    void DownloadFiles(RCF::FileDownload fileDownload)
    {
        fileDownload = RCF::FileDownload("C:\\FileToDownload.zip");
    }
};
```

Interrupted downloads can be resumed by specifying the same download location as for the previous download attempt:

```
// Either reuse the FileDownload object from the previous call:
client.DownloadFiles(fileDownload);

// , or call FileDownload::setDownloadToPath() on a new FileDownload:
RCF::FileDownload secondFileDownload;
std::string previousDownloadToPath = fileDownload.getDownloadToPath();
secondFileDownload.setDownloadToPath(previousDownloadToPath);
client.DownloadFiles(secondFileDownload);
```

To monitor the progess of a file download, use `ClientStub::setFileProgressCallback()`:

```
void onFileTransferProgress(const RCF::FileTransferProgress & progress)
{
    std::cout << "Total bytes to transfer: " << progress.mBytesTotalToTransfer << std::endl;
    std::cout << "Bytes transferred so far: " << progress.mBytesTransferredSoFar << std::endl;
   std::cout << "Server-side bandwidth limit (if any): " << progress.mServerLimitBps << std::endl;
}
```

```
client.getClientStub().setFileProgressCallback(&onFileTransferProgress);
```

To encrypt the download, set an appropriate transport protocol on the `RcfClient<>` object:

```
client.getClientStub().setTransportProtocol(RCF::Tp_Ntlm);
```

Similarly to file uploads, the server can monitor and optionally cancel any file downloads that are in progress:

```
void onServerFileDownload(RCF::RcfSession & session, const RCF::FileDownloadInfo & uploadInfo)
{
    // Which files are being downloaded.
    const RCF::FileManifest& manifest = uploadInfo.mManifest;

    // How far has the download progressed.
    boost::uint32_t currentFile = uploadInfo.mCurrentFile;
    boost::uint64_t currentFilePos = uploadInfo.mCurrentPos;

    // To cancel the download, throw an exception.
    // ...
}
```

```
server.setOnFileDownloadProgress(&onServerFileDownload);
```

Bandwidth limits

RCF file transfers will automatically run as fast as the network allows them to run. However, in some cases you may want to limit bandwidth usage and run file transfers at a restricted pace. RCF allows you set the maximum bandwidth consumed by all file uploads, or all file downloads, on a server:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );

// 1 Mbps upload limit.
boost::uint32_t serverUploadLimitMbps = 1;

// Convert to bytes/second.
boost::uint32_t serverUploadLimitBps = serverUploadLimitMbps*1000*1000/8;
server.setFileUploadBandwidthLimit(serverUploadLimitBps);

// 5 Mbps upload limit.
boost::uint32_t serverDownloadLimitMbps = 5;

// Convert to bytes/second.
boost::uint32_t serverDownloadLimitBps = serverDownloadLimitMbps*1000*1000/8;
server.setFileUploadBandwidthLimit(serverDownloadLimitBps);
```

In this example, if three clients are uploading files simultaneously, the total bandwidth consumed by all three clients will not be allowed to exceed 1 Mbps. Similarly, if there are three clients downloading files simultaneously, the total bandwidth consumed by the downloads will not be allowed to exceed 5 Mbps.

RCF also supports setting bandwidth limits on a more granular level, with application-defined subsets of clients sharing a particular bandwidth quota. For example, on a server connected to multiple networks with varying bandwidth characteristics, you may want to limit file transfer bandwidth based on which network the file transfer is taking place on. To do this, use `RcfServer::setFileUploadCustomBandwidthLimit()`/`RcfServer::setFileDownloadCustomBandwidthLimit()` to provide a callback function for the `RcfServer` to call, whenever a file transfer commences. From the callback funtion you can assign a relevant bandwidth quota.

For example, imagine we want to implement the following bandwidth limits:

1. 1 Mbps upload bandwidth for TCP connections from `192.168.*.*`.

2. 56 Kbps upload bandwidth for TCP connections from `15.146.*.*`.

3. Unlimited upload bandwidth for all other connections.

We'll need three separate `RCF::BandwidthQuota` objects for the three bandwidth quotas described above, and then we'll use `RcfServer::setFileUploadCustomBandwidthLimit()` to assign bandwidth quotas based on the IP address of the client:

```
// 1 Mbps quota bucket.
RCF::BandwidthQuotaPtr quota_1_Mbps( new RCF::BandwidthQuota(1*1000*1000/8) );

// 56 Kbps quota bucket.
RCF::BandwidthQuotaPtr quota_56_Kbps( new RCF::BandwidthQuota(56*1000/8) );

// Unlimited quota bucket.
RCF::BandwidthQuotaPtr quota_unlimited( new RCF::BandwidthQuota(0) );

RCF::BandwidthQuotaPtr uploadBandwidthQuotaCb(RCF::RcfSession & session)
{
    // Use clients IP address to determine which quota to allocate from.

    const RCF::RemoteAddress & clientAddr = session.getClientAddress();
    const RCF::IpAddress & clientIpAddr = dynamic_cast<const RCF::IpAddress &>(clientAddr);
    if ( clientIpAddr.matches( RCF::IpAddress("192.168.0.0", 16) ) )
    {
        return quota_1_Mbps;
    }
    else if ( clientIpAddr.matches( RCF::IpAddress("15.146.0.0", 16) ) )
    {
        return quota_56_Kbps;
    }
    else
    {
        return quota_unlimited;
    }
}
```

```
// Assign a custom file upload bandwidth limit.
server.setFileUploadCustomBandwidthLimit(&uploadBandwidthQuotaCb);
```

Versioning

Versioning, in this context, is the subject of upgrading distributed components (clients or servers) without breaking runtime compatibility with previously deployed components (clients or servers).

If you write components which only communicate with peer components of the same version, you won't need to worry about versioning. For instance, if all your components are built from the same codebase and deployed together, versioning won't be an issue.

However, if you, for example, have clients which need to communicate with servers that were built from an older version of the same codebase, and have already been deployed, then you will need to be aware of how RCF deals with versioning.

Interface versioning

Adding or removing functions

Each method in an RCF interface has a dispatch ID associated with it, which is used by clients to identify that particular method. The first method on an interface has a dispatch ID of 0, the next one a dispatch ID of 1, and so on.

Inserting a method at the beginning, or in the middle, of an RCF interface, changes the existing dispatch ID's and hence breaks compatibility with existing clients and servers. To preserve compatibility, methods need to be added at the end of the RCF interface:

```
// Version 1
RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_R2(double, add, double, double)
    RCF_METHOD_R2(double, subtract, double, double)
RCF_END(I_Calculator)
```

```
// Version 2.

// * Clients compiled against this interface will be able to call add() and
//   subtract() on servers compiled against the old interface.
//
// * Servers compiled against this interface will be able to take add() and
//   subtract() calls from clients compiled against the old interface.

RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_R2(double, add, double, double)
    RCF_METHOD_R2(double, subtract, double, double)
    RCF_METHOD_R2(double, multiply, double, double)
RCF_END(I_Calculator)
```

Removing methods can be done as well, as long as a place holder is left in the interface, in order to preserve the dispatch ID's of the remaining methods in the interface.

```
// Version 1
RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_R2(double, add, double, double)
    RCF_METHOD_R2(double, subtract, double, double)
RCF_END(I_Calculator)
```

```
// Version 2.

// * Clients compiled against this interface will be able to call subtract()
//   on servers compiled against the old interface.
//
// * Servers compiled against this interface will be able to take subtract()
//   calls from clients compiled against the old interface (but not add() calls).

RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_PLACEHOLDER()
    RCF_METHOD_R2(double, subtract, double, double)
RCF_END(I_Calculator)
```

Adding or removing parameters

Parameters can be added to a method, or removed from a method, without breaking compatibility. RCF servers and clients ignore any extra (redundant) parameters that are passed in a remote call, and if an expected parameter is not supplied, it is default initialized.

```
// Version 1
RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_R2(double, add, double, double)
RCF_END(I_Calculator)
```

```
// Version 2

// * Clients compiled against this interface will be able to call add()
//   on servers compiled against the old interface (the server will ignore
//   the third parameter).
//
// * Servers compiled against this interface will be able to take add()
//   calls from clients compiled against the old interface (the third parameter
//   will be default initialized to zero).

RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_R3(double, add, double, double, double)
RCF_END(I_Calculator)
```

Likewise, parameters can be removed:

```
// Version 1
RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_R2(double, add, double, double)
RCF_END(I_Calculator)
```

```
// Version 2

// * Clients compiled against this interface will be able to call add()
//   on servers compiled against the old interface (the server will assume the
//   second parameter is zero).
//
// * Servers compiled against this interface will be able to take add()
//   calls from clients compiled against the old interface (the second parameter
//   from the client will be ignored).

RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_R1(double, add, double)
RCF_END(I_Calculator)
```

Note that RCF marshals in-parameters and out-parameters in the order that they appear in the `RCF_METHOD_XX()` declaration. Any added (or removed) parameters must be the last to be marshalled, otherwise compatibility will be broken.

Renaming interfaces

RCF interfaces are identified by their runtime name, as specified in the second parameter of the `RCF_BEGIN()` macro. As long as this name is preserved, the compile time name of the interface can be changed, without breaking compatibility.

```
// Version 1
RCF_BEGIN(I_Calculator, "I_Calculator")
    RCF_METHOD_R2(double, add, double, double)
RCF_END(I_Calculator)
```

```
// Version 2

// * Clients compiled against this interface will be able to call add()
//   on servers compiled against the old interface.
//
// * Servers compiled against this interface will be able to take add()
//   calls from clients compiled against the old interface.

RCF_BEGIN(I_CalculatorService, "I_Calculator")
    RCF_METHOD_R2(double, add, double, double)
RCF_END(I_CalculatorService)
```

Archive versioning

The application-specific data types passed in a remote call, are likely to change over time. To assist applications in maintaining backwards compatibility, RCF provides an archive version number, which is automatically passed from the client to the server on each call. The archive version number is a 32 bit unsigned integer, and is set to zero by default. The archive version number can be retrieved from any serialization function, by calling SF::Archive::getArchiveVersion():

```
void serialize(SF::Archive & ar, MyClass & m)
{
    boost::uint32_t archiveVersion = ar.getArchiveVersion();
    // ...
}
```

The archive version number in use on a client-server connection is determined by an automatic version negotiation step that takes place when the client connects. The version negotiation step ensures that the connection uses the greatest archive version number that both components support.

When breaking changes are made to serialization functions, the archive version number should be updated to reflect the change. Typically you would increment the archive version number once for each release of the application. The serialization functions then use the value of the archive version number, to determine which members to serialize.

For example, assume that the first version of your application contains this code:

```
class X
{
public:
    X() : mN(0)
    {}

    int mN;

    void serialize(SF::Archive & ar)
    {
        ar & mN;
    }
};

RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(X, echo, const X &)
RCF_END(I_Echo)

class Echo
{
public:
    X echo(const X & x) { return x; }
};
```

```
//----------------------------------------------------------------------------
// Accepting calls from other processes...
Echo echo;
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
server.bind<I_Echo>(echo);
server.start();

//----------------------------------------------------------------------------
// ... or making calls to other processes.
RcfClient<I_Echo> client( RCF::TcpEndpoint(50002) );

X x1;
X x2;
x2 = client.echo(x1);
```

Once this version has been released, a new version is prepared, with a new member added to the `X` class:

```
class X
{
public:
    X() : mN(0)
    {}

    int mN;
    std::string mS;

    void serialize(SF::Archive & ar)
    {
        // Retrieve archive version, to determine which members to serialize.
        boost::uint32_t version = ar.getArchiveVersion();

        if (version == 0)
        {
            ar & mN;
        }
        else if (version == 1)
        {
            ar & mN;
            ar & mS;
        }
        else
        {
            // Unsupported version
            throw std::runtime_error("Unsupported version.");
        }
    }
};

class Echo
{
public:
    X echo(const X & x) { return x; }
};
```

Notice that the serialization code of `X` now uses the archive version number to determine whether it should serialize the new `mS` member.

With these changes, new servers are able to process calls from both old and new clients, and new clients are able to call either old or new servers:

```
// The default archive version should be the latest archive version this process supports.
RCF::setDefaultArchiveVersion(1);

//----------------------------------------------------------------------------
// Accepting calls from other processes...

// This server can take calls from either new or old clients. Archive version
// will be 0 when old clients call in, and 1 when new clients call in.
Echo echo;
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
server.bind<I_Echo>(echo);
server.start();

//----------------------------------------------------------------------------
// ... or making calls to other processes.

// This client can call either new or old servers.
RcfClient<I_Echo> client( RCF::TcpEndpoint(50002) );

X x1;
X x2;

// If the server on port 50002 is old, this call will have archive version set to 0.
// If the server on port 50002 is new, this call will have archive version set to 1.
x2 = client.echo(x1);
```

Runtime versioning

RCF maintains runtime compatibility with itself, for all releases dating back to and including RCF 0.9c, released in July 2007.

To implement runtime compatibility between RCF releases, RCF maintains a runtime version number, which is incremented for each RCF release. The runtime version number is passed in the request header for each remote call, and allows old and new RCF releases to interoperate.

RCF's automatic client-server version negotiation handles runtime versioning, as well as archive versioning. In most circumstances, you won't need to know about runtime version numbers - you can mix and match RCF releases, and at runtime, an appropriate runtime version is negotiated for each client-server connection.

Custom version negotiation

RCF's automatic version negotiation does require an extra round trip between the client and the server, in the case of a version mismatch. In some situations, the client may already know the runtime version and archive version of the server it is about to call, in which case it can disable automatic versioning and set the version numbers explicitly:

```
RcfClient<I_Echo> client( RCF::TcpEndpoint(50001) );

// Turn off automatic version negotiation.
client.getClientStub().setAutoVersioning(false);

// Assume that the server is running RCF 1.1, with archive version 2.
client.getClientStub().setRuntimeVersion(5); // RCF 1.1
client.getClientStub().setArchiveVersion(2);

// If the server doesn't support the requested version numbers, an exception will be thrown.
X x1;
X x2 = client.echo(x1);
```

Automatic version negotiation is not supported for oneway calls. In particular, the oneway calls made by a publisher to its subscribers, are not automatically versioned. If you have subscribers with varying runtime and archive version numbers, the publisher will need to explicitly set version numbers on the publishing `RcfClient<>` object, to match that of the oldest subscriber:

```
RCF::RcfServer server( RCF::TcpEndpoint(50001) );
server.start();

typedef RCF::Publisher<I_HelloWorld> HelloWorldPublisher;
typedef boost::shared_ptr< HelloWorldPublisher > HelloWorldPublisherPtr;
HelloWorldPublisherPtr pubPtr = server.createPublisher<I_HelloWorld>();

// Explicitly set version numbers to support older subscribers.
pubPtr->publish().getClientStub().setRuntimeVersion(3); // RCF runtime version 3 (RCF 0.9d).
pubPtr->publish().getClientStub().setArchiveVersion(5); // Application archive version 5.
```

For reference, here is a table of runtime version numbers for each RCF release.

RCF release	Runtime version number
0.9c	2
0.9d	3
1.0	4
1.1	5
1.2	6
1.3	8
1.3.1	9
2.0	10

Protocol Buffers

For applications with backwards compatibility requirements and short or continuous release cycles, archive versioning can become difficult to manage. Each increment of the archive version number involves adding new execution paths to serialization functions, and may lead to complicated serialization code.

RCF also supports Protocol Buffers, which provides an alternative approach to versioning. Rather than manually writing serialization code for C++ objects, Protocol Buffers can be used to generate C++ classes with built-in serialization code, which deals automatically with versioning differences (see Protocol Buffers).

Performance

RCF provides two basic performance guarantees for remote calls: zero copy and zero allocation.

Zero copy

RCF makes no internal copies of data while sending or receiving, either on the server or the client. However, serialization often forces copies to be made. For instance, deserializing a `std::string` can't be done without making a copy of the string contents, because `std::string` always allocates its own storage. The same goes for `std::vector<>`. RCF provides a workaround for this issue, via the `RCF::ByteBuffer` class. The contents of a `ByteBuffer` are not copied upon serialization or deserialization, instead RCF uses scatter/gather style semantics to send and receive the contents directly.

To improve performance, you should use `RCF::ByteBuffer` whenever transferring large chunks of untyped data. On typical hardware, transferring multiple megabytes of data in a single call is not a problem.

Zero allocation

Zero allocation means that, whether on server or client, RCF will not make any heap allocations unless processing the first call on a connection, or processing a call with larger amounts of data than on the previous call of the same connection. In particular, if a remote call is made twice with identical parameters, on the same connection, RCF guarantees that it will not make any heap allocations on the second call, either on the client or on the server:

```
bool gExpectAllocations = true;

// Override global operator new so we can intercept heap allocations.
void *operator new(size_t bytes)
{
    if (!gExpectAllocations)
    {
        throw std::runtime_error("Unexpected heap allocation!");
    }
    return malloc(bytes);
}

void operator delete (void *pv) throw()
{
    free(pv);
}

// Override global operator new[] so we can intercept heap allocations.
void *operator new [](size_t bytes)
{
    if (!gExpectAllocations)
    {
        throw std::runtime_error("Unexpected heap allocation!");
    }
    return malloc(bytes);
}

void operator delete [](void *pv) throw()
{
    free(pv);
}
```

```
RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(RCF::ByteBuffer, echo, RCF::ByteBuffer)
RCF_END(I_Echo)

class Echo
{
public:
    RCF::ByteBuffer echo(RCF::ByteBuffer byteBuffer)
    {
        return byteBuffer;
    }
};
```

```
RcfClient<I_Echo> client(( RCF::TcpEndpoint(port)));

// First call will trigger some heap allocations.
gExpectAllocations = true;
client.echo(byteBuffer);

// These calls won't trigger any client-side or server-side heap allocations.
gExpectAllocations = false;
for (std::size_t i=0; i<10; ++i)
{
    RCF::ByteBuffer byteBuffer2 = client.echo(byteBuffer);
}
```

However, serialization functions of the data types involved in the remote call, may make heap allocations of their own. To eliminate allocations associated with deserialization, RCF provides server-side object caching.

Server-side object caching

Serialization and deserialization of remote call parameters can become a performance bottleneck. In particular, deserialization of a complex datatype involves not only creating the object to begin with, but also a number of memory allocations (and CPU cycles) when deserializing all the fields and subfields of the object.

To improve performance in these circumstances, RCF provides a server-side cache of objects used during remote calls. Objects used as parameters in one remote call, can be transparently reused in subsequent calls. This means that construction overhead and memory allocations due to deserialization, can be eliminated in subsequent calls.

For example:

```
RCF_BEGIN(I_Echo, "I_Echo")
RCF_METHOD_R1(std::string, echo, std::string)
RCF_END(I_Echo)

class Echo
{
public:
    std::string echo(const std::string & s)
    {
        return s;
    }
};
```

```
Echo echo;
RCF::RcfServer server( RCF::TcpEndpoint(0));
server.bind<I_Echo>(echo);
server.start();

int port = server.getIpServerTransport().getPort();

RCF::ObjectPool & cache = RCF::getObjectPool();

// Enable caching for std::string.
// * Don't cache more than 10 std::string objects.
// * Call std::string::clear() before putting a string into the cache.
cache.enableCaching<std::string>(10, boost::bind(&std::string::clear, _1));

std::string s1 = "123456789012345678901234567890";
std::string s2;

RcfClient<I_Echo> client(( RCF::TcpEndpoint(port) ));

// First call.
s2 = client.echo(s1);

// Subsequent calls - no memory allocations at all, in RCF runtime, or
// in std::string serialization/deserialization, on client or server.

for (std::size_t i=0; i<100; ++i)
{
    s2 = client.echo(s1);
}

// Disable caching for std::string.
cache.disableCaching<std::string>();
```

In this example, the first call to `echo()` will cause several server-side deserialization-related memory allocations - one to construct a `std::string`, and another to expand the internal buffer of the string, to fit the incoming data.

With server-side object caching enabled, after the call returns, the server-side string is cleared and then held in a cache, rather than being destroyed. On the next call, instead of constructing a new `std::string`, RCF reuses the `std::string` in the cache. Upon deserialization, `std::string::resize()` is called, to fit the incoming data. As this particular string object has already held the requested amount of data earlier, the `resize()` request does not result in any memory allocation.

The server-side object cache is configured on a per-type basis, using the `ObjectPool::enableCaching<>()` and `ObjectPool::disableCaching<>()` functions. For each cached datatype, you can specify the maximum number of objects to cache, and which function to call, to put the objects in a reusable state.

Scalability

The zero copy and zero allocation guarantees affect the execution of a single remote call. Another performance factor is the maximum number of concurrent clients a server can support. RCF's server transport implementation is based on Boost.Asio, which leverages native network API's when they are avaialable (I/O completion ports on Windows, `epoll()` on Linux, `/dev/poll` on Solaris, `kqueue()` on FreeBSD), and less performant API's when they aren't (BSD sockets). The number of clients a server can support is essentially determined by how much memory the server has, and the application-specific resources required by each client.

RCF has been designed to yield minimal performance overhead, with network intensive, high throughput applications in mind. Keep in mind that bottlenecks in distributed systems tend to be determined by the overall design of the distributed system - a poorly designed distributed system will have its performance cut off well before the communications layer reaches its limit.

Advanced Serialization

User-defined C++ objects can be quite complex to serialize. This section describes some of the more advanced features of RCF's internal serialization framework, SF.

Polymorphic serialization

SF will automatically serialize polymorphic pointers and references, as fully derived types. However, to do this, SF needs to be configured with two pieces of information about the polymorphic type being serialized. First, SF needs a runtime identifier string for the polymorphic type. Second, it needs to know which base classes the polymorphic type will be serialized through.

As an example, consider the following polymorphic class hierarchy.

```
class A
{
public:
    A() : mA() {}
    virtual ~A() {}

    void serialize(SF::Archive &ar)
    {
        ar & mA;
    }

    int mA;
};

class B : public A
{
public:
    B() : mB() {}

    void serialize(SF::Archive &ar)
    {
        SF::serializeParent<A>(ar, *this);
        ar & mB;
    }

    int mB;
};

class C : public B
{
public:
    C() : mC() {}

    void serialize(SF::Archive &ar)
    {
        SF::serializeParent<B>(ar, *this);
        ar & mC;
    }

    int mC;
};
```

Note that `SF::serializeParent<>()` is used to invoke base class serialization code. If you try to serialize the parent directly, e.g. by calling `ar & static_cast<A&>(*this)`, SF will detect that the parent class is actually a derived class, and will try to serialize the derived class once again.

We want to implement polymorphic serialization of `A *` pointers, for use in the `X` class:

```
class X
{
public:
    X() : mpA()
    {}

    void serialize(SF::Archive &ar)
    {
        ar & mpA;
    }

    A * mpA;
};

RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(X, Echo, X)
RCF_END(I_Echo)
```

First we need to suppply runtime identifiers for the `B` and `C` classes.

```
SF::registerType<B>("B");
SF::registerType<C>("C");
```

The runtime identifiers will be included in serialized archives when `B` and `C` objects are serialized, and will allow the deserialization code to construct objects of the appropriate type.

We also need to specify which base types the `B` and `C` classes will be serialized through. In this case, `B` and `C` objects will be serialized through an `A` pointer.

```
SF::registerBaseAndDerived<A, B>();
SF::registerBaseAndDerived<A, C>();
```

With the SF runtime now configured, we can run this code:

```
RcfClient<I_Echo> client(( RCF::TcpEndpoint(port)));

X x1;
x1.mpA = new B();
X x2 = client.Echo(x1);

x1.mpA = new C();
x2 = client.Echo(x1);
```

The polymorphic `A` pointers contained in the `X` objects will be serialized and deserialized, as fully derived polymorphic types.

Finally, it's the applications responsiblity to delete the `X::mpA` pointer that SF creates upon deserialization. The easiest way to ensure this happens, is to use a C++ smart pointer rather than a raw pointer. SF supports a number of smart pointers, including:

- `std::auto_ptr<>`
- `boost::scoped_ptr<>`
- `boost::shared_ptr<>`

So for example, we can write `X` as:

```
class X
{
public:
    X() : mpA()
    {}

    void serialize(SF::Archive &ar)
    {
        ar & mpA;
    }

    typedef boost::shared_ptr<A> APtr;
    APtr mpA;
};
```

Pointer tracking

If you serialize a pointer to the same object twice, SF will by default serialize the entire object twice. This means that when the pointers are deserialized, they will point to two distinct objects. In most applications this is usually not an issue. However, some applications may want the deserialization code to instead create two pointers to the same object.

SF supports this through a pointer tracking concept, where an object is serialized in its entirety, only once, regardless of how many pointers to it are serialized. Upon deserialization, only a single object is created, and multiple pointers can then be deserialized, pointing to the same object.

To demonstrate pointer tracking, here is an an `I_Echo` interface with an `Echo()` function that takes a pair of `boost::shared_ptr<>`'s:

```
typedef
    std::pair< boost::shared_ptr<int>, boost::shared_ptr<int> >
    TwoPointers;

RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(TwoPointers, Echo, TwoPointers)
RCF_END(I_Echo2)
```

Here is the client-side code:

```
boost::shared_ptr<int> spn1( new int(1));
boost::shared_ptr<int> spn2( spn1 );

TwoPointers ret = client.Echo( std::make_pair(spn1,spn2));
```

If we make a call to `Echo()` with a pair of `shared_ptr<>`'s pointing to the same `int`, we'll find that the returned pair points to two distinct `int`'s. To get them to point to the same `int`, we enable pointer tracking on the client-side:

```
RcfClient<I_Echo> client(( RCF::TcpEndpoint(port)));

client.getClientStub().setEnableSfPointerTracking(true);
```

, and also on the server-side, since we are returning the pointers back to the client:

```
class EchoImpl
{
public:
    template<typename T>
    T Echo(T t)
    {
        RCF::getCurrentRcfSession().setEnableSfPointerTracking(true);
        return t;
    }
};
```

The two returned `shared_ptr<>`'s will now point to the same instance.

It is worth keeping in mind that pointer tracking is relatively expensive. It requires the serialization framework to track all pointers and values that are being serialized, with significant performance overhead.

Interchangeable types

SF guarantees that certain types are interchangeable, as far as serialization is concerned. For example, it is possible to serialize a pointer, and subsequently deserialize it into a value:

```
// Serializing a pointer.
int * pn = new int(5);
std::ostringstream ostr;
SF::OBinaryStream(ostr) << pn;
delete pn;

// Deserializing a value.
int m = 0;
std::istringstream istr(ostr.str());
SF::IBinaryStream(istr) >> m;
// m is now 5
```

Essentially, `T *` and `T` have compatible serialization formats, for any `T`. SF also guarantees that smart pointers are interchangeable with native pointers, so it is fine to serialize a `boost::shared_ptr<>` and then deserialize it into a value. Here is another example:

```
// Client-side implementation of X, using int.
class X
{
public:
    int n;

    void serialize(SF::Archive & ar, unsigned int)
    {
        ar & n;
    }
};
```

```
// Server-side implementation of X, using shared_ptr<int>.
class X
{
public:
    boost::shared_ptr<int> spn;

    void serialize(SF::Archive & ar)
    {
        ar & spn;
    }
};
```

```
// Even with different X implementations, client and server can still
// interact through this interface.

RCF_BEGIN(I_EchoX,"I_EchoX")
RCF_METHOD_R1(X, Echo, X)
RCF_END(I_EchoX)
```

The following table lists the sets of types that SF guarantees to be interchangeable:

Sets of interchangeable types
`T`, `T *`, `std::auto_ptr<T>`, `boost::shared_ptr<T>`, `boost::scoped_ptr<T>`
Integral types of equivalent size, e.g. `unsigned int` and `int`
32 bit integral types, enums
STL containers of `T` (including `std::vector<>` and `std::basic_string<>`), where `T` is non-primitive
STL containers of `T` (except `std::vector<>` and `std::basic_string<>`), where `T` is primitive
`std::vector<T>` and `std::basic_string<T>`, where `T` is primitive
`std::string`, `std::vector<char>`, `RCF::ByteBuffer`

The reason for the exceptions concerning `std::vector<>` and `std::basic_string<>`, is that SF implements fast `memcpy()`-based serialization for these containers, if their elements are of primitive type.

Unicode strings

When a `std::wstring` is serialized through SF, it is serialized as a UTF-8 encoded string, in order to ensure portability. On platforms with 16 bit `wchar_t`, SF assumes that any `std::wstring` passed to it, is encoded in UTF-16, and converts between UTF-16 and UTF-8. On platforms with 32 bit `wchar_t`, SF assumes that any `std::wstring` passed to it is encoded in UTF-32, and converts between UTF-32 and UTF-8.

If you have `std::wstring` objects, encoded in something other than UTF-16 or UTF-32, you would need to either convert them to an 8-bit represention (`std::string`) yourself, before serializing, or write a wrapper class around `std::wstring`, with a customized serialization function.

Third Party Serialization

RCF provides support for several third pary serialization frameworks.

Boost.Serialization

Boost.Serialization is a library conceptually similar to RCF's internal serialization framework, providing generic serialization for C++ classes. For reference material and FAQ's on Boost.Serialization itself, please refer to Boost documentation and the Boost mailing lists.

To utilize Boost.Serialization within RCF, you will need to define `RCF_USE_BOOST_SERIALIZATION` when building RCF, and then provide Boost.Serialization `serialize()` functions for all your datatypes.

RCF supports two archive types from Boost.Serialization - binary archives and text archives. To specify which one you want to use, call `ClientStub::setSerializationProtocol()`:

```
RcfClient<I_X> client( RCF::TcpEndpoint(50001) );

// To use Boost.Serialization with binary archives.
client.getClientStub().setSerializationProtocol(RCF::Sp_BsBinary);

// To use Boost.Serialization with text archives.
client.getClientStub().setSerializationProtocol(RCF::Sp_BsText);
```

If RCF has also been built with `RCF_USE_SF_SERIALIZATION` defined, you can also specify SF serialization:

```
// To use SF serialization.
client.getClientStub().setSerializationProtocol(RCF::Sp_SfBinary);
```

Given the choice between using SF and using Boost.Serialization, which should you use? The answer of course depends largely on the needs of your own application. However, within the context of RCF, we feel SF is a better choice for a number of reasons:

- SF is RCF's native serialization implementation. It is updated along with RCF, and tested against RCF. It's behavior and configuration are not dependent on any particular version of Boost.
- SF has a robust internal versioning scheme. The SF runtime can adjusts its archive formats to be compatible with any older version of SF. This capability is crucial for implementing backwards compatibility in distributed applications, and is lacking in Boost.Serialization.
- SF is optimized for IPC usage, where tens of thousands of archives may be created and destroyed every second.
- SF intentionally makes it easy to modify the types in your application, so that newer versions of an application can evolve their object models, without affecting wire compatibility with older versions of the same application (see Interchangeable types). In contrast, Boost.Serialization requires exact matches on source and destination types.
- SF has a robust type registration system. Type registration with Boost.Serialization involves macros, global static objects, and implicit code generation, with application wide side-effects. In some versions of Boost, type registration even requires including headers in a particular order. In contrast, type registration with SF is as simple as calling `SF::registerType()` and `SF::registerBaseAndDerived()`.
- SF archives are binary and guaranteed to be portable across platforms. Boost.Serialization binary archives are not portable, and hence can't be used for cross-platform IPC. Boost.Serialization text archives are portable, but at the cost of being much slower than binary archives.
- SF serialization functions are not templated, which means their definitions can be moved to source files. Boost.Serialization serialization functions are generally templated on the archive type, thus requiring their implementations to be placed inline in header files, causing excessive coupling and longer compile times.

- Boost.Serialization archive formats can change implicitly, depending on global static code instantiations in unrelated parts of the same executable, to the point where the archives produced cannot be loaded again (see this bug report, which unfortunately was closed without a fix).

Protocol Buffers

Protocol Buffers is an open source project released by Google, which uses a specialized language to define serialization schemas, from which actual serialization code is generated. The official Protocol Buffers compiler can generate code in three languages (C++, Java and Python), and various third party projects are available to provide support for other languages.

Protocol Buffer classes

RCF provides native marshaling support for classes generated by the Protocol Buffers compiler. To enable this support, define `RCF_USE_PROTOBUF` when building RCF. With `RCF_USE_PROTOBUF` defined, classes produced by the Protocol Buffers compiler, can be used directly in RCF interface declarations, and will be serialized and deserialized using the Protocol Buffer runtime.

For more information on building RCF with Protocol Buffer support, see Appendix - Building.

As an example, consider this .proto file.

```
// Person.proto

message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

Running the Protocol Buffer compiler (in C++ mode) over `Person.proto` results in the two files `Person.pb.h` and `Person.pb.cc`, containing the implementation of the C++ class `Person`.

```
// Person.pb.h

class Person : public ::google::protobuf::Message {
// ...
}
```

To use the class `Person` in an RCF interface, simply include `Person.pb.h` . RCF will detect that the `Person` class is a Protocol Buffer class, and will use Protocol Buffer functions to serialize and deserialize `Person` instances.

```
#include <RCF/../../test/protobuf/messages/cpp/Person.pb.h>


RCF_BEGIN(I_X, "I_X")
    RCF_METHOD_R1(Person, echo, const Person &)
RCF_END(I_X)
```

Protocol Buffer classes can be intermingled with native C++ classes, in RCF interfaces:

```
RCF_BEGIN(I_X, "I_X")
    RCF_METHOD_R1(Person, echo, const Person &)
    RCF_METHOD_V4(void , func, int, const std::vector<std::string> &, Person &, RCF::ByteBuffer)
RCF_END(I_X)
```

Protocol Buffer caching

The serialization and deserialization code generated by Protocol Buffers is highly optimized. However, to make the most of this performance, you may want to cache and reuse Protocol Buffer objects from call to call, rather than creating new ones. Protocol Buffer objects will hold on to any memory they allocate, and will reuse the memory in subsequent serialization and deserialization operations.

RCF provides server-side object caching, for this purpose. For example, to enable server-side caching of the `Person` class:

```
RCF::ObjectPool & cache = RCF::getObjectPool();

// Enable server-side caching for Person.
// * Don't cache more than 10 Person objects.
// * Call Person::Clear() before putting a Person object into the cache.
cache.enableCaching<Person>(10, boost::bind(&Person::Clear, _1));
```

JSON-RPC

In addition to serving RCF clients, `RCF::RcfServer` let's you expose C++ server functionality to JSON-RPC clients, by functioning as a JSON-RPC server.

To enable JSON-RPC support, you will need to download the JSON Spirit library, set the relevant compiler include paths for it, and define `RCF_USE_JSON` when building RCF (see Building).

To serve JSON-RPC clients, you will need a `RcfServer` with an HTTP or HTTPS endpoint, configured to receive JSON-RPC requests instead of regular RCF requests:

```
RCF::RcfServer server;
server.addEndpoint( RCF::HttpEndpoint(80) )
    .setRpcProtocol(RCF::Rp_JsonRpc);
```

For each JSON-RPC service you want to implement, you need to define a function that takes a `RCF::JsonRpcRequest` and returns a `RCF::JsonRpcResponse`:

```
void JsonPrint(
    const RCF::JsonRpcRequest & request,
    RCF::JsonRpcResponse & response)
{
    // ...
}
```

Use `RcfServer::bindJsonRpc()` to expose this function as a JSON-RPC service, with the given method name:

```
server.bindJsonRpc(
    boost::bind(&JsonPrint, _1, _2),
    "JsonPrint");
```

In the servant implemenation of the JSON-RPC request (in this case the `JsonPrint()` function), you can use `JsonRpcRequest` to access the JSON-RPC attributes and JSON-RPC parameters of the request. To send a response back to the client, fill out the `JsonRpcResponse` object with the relevant details:

```
void JsonPrint(
    const RCF::JsonRpcRequest & request,
    RCF::JsonRpcResponse & response)
{
    // Print out all the strings passed in, and return the number of
    // characters printed.

    int charsPrinted = 0;

    const json_spirit::Array & params = request.getJsonParams();
    for (std::size_t i=0; i<params.size(); ++i)
    {
        const std::string & s = params[i].get_str();
        std::cout << "I_HelloWorld service: " << s << std::endl;
        charsPrinted += s.size();
    }

    // Return number of characters printed.
    json_spirit::mObject & responseObj = response.getJsonResponse();
    responseObj["result"] = charsPrinted;
}
```

RCF supports JSON-RPC over both HTTP and HTTPS. When using JSON-RPC over HTTPS, you will need to configure a certificate for the server. This is done in the same way as when configuring the SSL transport protocol (see Transport protocols). For example:

```
RCF::RcfServer server;

server.addEndpoint( RCF::HttpsEndpoint(443) )
    .setRpcProtocol(RCF::Rp_JsonRpc);

// Assume we are using Schannel on Windows.
RCF::CertificatePtr serverCertPtr( new RCF::PfxCertificate(
    "C:\\ServerCert.p12",
    "password",
    "CertificateName") );

server.setCertificate(serverCertPtr);
```

Finally, as `RcfServer` instances support multiple transports (see Transports), you can serve regular RCF clients as well as JSON-RPC clients, from the same `RcfServer`:

```
RCF::RcfServer server;

// Serve JSON-RPC clients over HTTP on port 80.
server.addEndpoint( RCF::HttpEndpoint(80) )
    .setRpcProtocol(RCF::Rp_JsonRpc);

server.bindJsonRpc(
    boost::bind(&JsonPrint, _1, _2),
    "JsonPrint");

// Serve RCF clients over TCP on port 50001.
server.addEndpoint( RCF::TcpEndpoint(50001) );

// Serve RCF clients over HTTP on port 50002.
server.addEndpoint( RCF::HttpEndpoint(50002) );

HelloWorldImpl helloWorldImpl;
server.bind<I_HelloWorld>(helloWorldImpl);

server.start();
```

Appendix - Building RCF

RCF is provided as source code, and is intended to be compiled along with the rest of the source code that comprises your application. To compile RCF, you only compile a single file - `RCF.cpp`, located in the distribution at `src/RCF/RCF.cpp`.

You can compile `RCF.cpp` directly into the same module as your own sources, or you can compile it into a static or dynamic library and subsequently link it into your application.

The Boost header files are RCF's only mandatory dependency. Other dependencies (zlib, OpenSSL, JSON-Spirit, Boost.FileSystem, Boost.Serialization, Protocol Buffers) are optional, and are enabled through the configuration macros listed further down.

In summary, to build RCF, you need to:

1. Make sure a version of the Boost library is available.

2. Make sure any relevant third party libraries are available.

3. Define any relevant configuration macros.

4. Compile RCF.cpp .

5. Link to any relevant third party libraries.

The actual details of compiling and linking depend on the compiler toolset you are using. If you are using Visual Studio, you will need to set include paths, compiler defines, linker paths, linker input in the project properties of your Visual C++ project. If you are using command line tools (in particular gcc, with some flavor of makefiles), you will need to set include paths, compiler defines, linker paths, and linker input on the relevant command lines.

For a simple example of building with Visual Studio 2010 and gcc, please see the Tutorial.

Configuration

The following table lists all RCF build configuration macros:

Macro	Meaning	Requires linking to
`RCF_USE_OPENSSL`	Build with OpenSSL support.	OpenSSL
`RCF_OPENSSL_STATIC`	Enable static linking to OpenSSL.	OpenSSL
`RCF_USE_ZLIB`	Build with zlib support.	Zlib
`RCF_ZLIB_STATIC`	Enable static linking to zlib.	Zlib
`RCF_USE_SF_SERIALIZATION`	Build with SF serialization support (defined by default).	
`RCF_USE_BOOST_SERIALIZATION`	Build with Boost.Serialization support.	Boost.Serialization
`RCF_USE_BOOST_FILESYSTEM`	Build with Boost.Filesystem support (required for file transfer support).	Boost.Filesystem
`RCF_USE_PROTOBUF`	Build with Protocol Buffer support.	Protocol Buffers
`RCF_USE_JSON`	Build with JSON Spirit (required for JSON-RPC support).	
`RCF_USE_IPV6`	Build with IPv6 support.	
`RCF_USE_CUSTOM_ALLOCATOR`	Build with custom allocator support.	
`RCF_BUILD_DLL`	Build a DLL exporting RCF.	
`RCF_AUTO_INIT_DEINIT`	Enable automatic RCF initialization and deinitialization.	

Building RCF without any of the configuration macros defined, is equivalent to building RCF with the `RCF_USE_SF_SERIALIZATION` define. If you build RCF with `RCF_USE_BOOST_SERIALIZATION` defined, `RCF_USE_SF_SERIALIZATION` is automatically **not** defined, and needs to be manually defined if you intend to use both Boost and SF serialization in the same build.

Make sure that all modules that will be linked together are compiled with the same set of RCF configuration macros, as some of the configuration macros are not ABI compatible with each other.

`RCF_USE_CUSTOM_ALLOCATOR` is used to customize memory allocation for network send and receive buffers. If you define `RCF_USE_CUSTOM_ALLOCATOR`, you will need to implement the following two functions:

```
namespace RCF {
    void * RCF_new(std::size_t bytes);
    void RCF_delete(void * ptr);
} // namespace RCF
```

`RCF_new()` and `RCF_delete()` will be called by RCF whenever allocating or deallocating memory for buffers of data that are being sent or received.

`RCF_BUILD_DLL` needs to be defined when building dynamic libraries, and will mark RCF classes and functions as DLL exports.

The `RCF_AUTO_INIT_DEINIT` define is provided, for users who want to preserve the automatic initialization and deinitialization behavior of earlier versions of RCF. With `RCF_AUTO_INIT_DEINIT` defined, a static object in RCF will automatically call `RCF::init()` when RCF is loaded, and `RCF::deinit()` when RCF is unloaded.

In RCF 2.0 and later, you are expected to initialize RCF explicitly, by calling `RCF::init()`.

RCF initialization is reference counted, which means RCF won't be deinitialized until you have called `RCF::deinit()` as many times as you have called `RCF::init()`. The `RCF::RcfInitDeinit` class will call `RCF::init()` from its constructor, and `RCF::deinit()` from its destructor, and can be used to make sure all calls to `RCF::init()` are matched with a subsequent call to `RCF::deinit()`.

Third party library versions

For reference, these are versions of third party libaries RCF has been tested against. Newer versions of these libraries should also be fine to use.

Library	**Version**
Boost	1.35.0 - 1.50.0
OpenSSL	0.9.8d
Zlib	1.2.1
Protocol Buffers	2.1.0
JSON Spirit	4.05

Compilers

RCF has been tested against the following compilers:

- Visual C++ - any version newer than and including Visual C++ 7.1 (Visual Studio 2003).
- gcc - any version newer than and including gcc 3.4.

In addition, RCF will generally compile, possibly with minor modifications, on standard compliant compilers on most platforms. However, RCF releases are only tested against the compilers listed above.

Platforms

RCF has been tested on the following platforms:

- Windows
- Linux
- Solaris
- FreeBSD
- OS X

Appendix - Logging

RCF has a configurable logging subsystem which can be controlled through the `RCF::enableLogging()` and `RCF::disableLogging()` functions. To use these functions, you will need to include the `<RCF/util/Log.hpp>` header.

By default, logging is disabled. To enable logging, call `RCF::enableLogging()`. `RCF::enableLogging()` takes two optional parameters: log level and log target:

```
// Using default values for log target and log level.
RCF::enableLogging();

// Using custom values for log target and log level.
int logLevel = 2;
RCF::enableLogging(RCF::LogToDebugWindow(), logLevel);
```

The log level can range from 0 (no logging at all) to 4 (maximum logging). The default log level is 2.

The log target parameter can be one of the following:

Log target	**Log output location**
`LogToDebugWindow()`	Windows only. Log output appears in Visual Studio debug output window.
`LogToStdout()`	Log output appears on standard output.
`LogToFile(const std::string & logFilePath)`	Log output appears in the nominated file.
`LogToFunc(boost::function<void(const RCF::ByteBuffer &)>)`	Log output is passed to a user-defined function.

On Windows platforms, the default log target is `LogToDebugWindow`.

On non-Windows platforms, the default log target is `LogToStdout`.

Finally, to disable logging, call `RCF::disableLogging()`:

```
// Disable logging.
RCF::disableLogging();
```

`RCF::enableLogging()` and `RCF::disableLogging()` are internally threadsafe and can be called by multiple threads concurrently.

Appendix - Release Notes

RCF 2.0.1.101 - 2014-09-06

- Fix default settings of `RCF_FEATURE_SF` and `RCF_FEATURE_BOOST_SERIALIZATION` config macros, to respect legacy `RCF_USE_SF_SERIALIZATION` and `RCF_USE_BOOST_SERIALIZATION` macros.
- Fix compiler error when deriving custom class from `RcfClient<>`.
- Fix compiler warnings appearing in Visual Studio 2013 Update 3, about deprecated functions in Winsock.
- Downloads:
  - RCF-2.0.1.101.zip
  - RCF-2.0.1.101.tar.gz

RCF 2.0.1.100 - 2014-04-24

- Fix potential busy loop for multithreaded UDP servers.
- Expose `RcfServer::getFilterService()` for code using legacy custom filters.
- Fix for `RCF_USE_PROTOBUF` define not working.
- Fix potential handle leak in Windows thread implementation (`win_thread.hpp`).
- Fix for thread joining in Windows thread implementation (`win_thread::join()`) - contributed by acDev on support forums.
- Fix bug in shutdown order for `RcfServer`.
- Implement `RCF_OPENSSL_STATIC` define, to allow static linking to OpenSSL.
- Explicitly unload zlib and OpenSSL DLL's, when deinitializing RCF.
- Implement serialization for various QT classes - see `SF/Q*.hpp`. Contributed by acDev on support forums.
- Fix serialization for `std::vector<bool>`.
- Implement serialization for `std::bitset<>`.
- Implement `SF_SERIALIZE_ENUM_CLASS` macro, to simplify serialization of C++11 enum classes.
- Fix compiler error when serializing `shared_ptr<const T>`.
- Publish/subscribe - published messages are now sent asynchronously and concurrently on all subscriber connections.
- Implement `Publisher::getSubscriberCount()`, to return number of subscriptions for a publisher.
- Downloads:
  - RCF-2.0.1.100.zip
  - RCF-2.0.1.100.tar.gz

RCF 2.0.0.2685 - 2013-07-17

- More informative error messages upon certificate validation failure, when using OpenSSL.

- Fix for thread local cache leaks reported by Visual Leak Detector.
- Fix order of destruction issue in `RcfServer` destructor, when using multiple server transports.
- Downloads:
  - RCF-2.0.0.2685.zip
  - RCF-2.0.0.2685.tar.gz

RCF 2.0.0.2683 - 2013-06-18

- Allow compile-time linking to zlib, by defining `RCF_ZLIB_STATIC` in build.
- Downloads:
  - RCF-2.0.0.2683.zip
  - RCF-2.0.0.2683.tar.gz

RCF 2.0.0.2682 - 2013-06-01

- Fix potential order of destruction problem in `SessionTimeoutService`.
- Fix bool conversion compiler error in `RcfSession::hasDefaultServerStub()` (reported on gcc 4.6.3) .
- Accumulate multiple `ByteBuffer`'s in SSPI filter, to prevent potential fragmentation of network send operations.
- `ThreadPool` refactored, with simpler constructors and get / set methods.
- Classes renamed: `I_Endpoint` to `Endpoint`, `I_ServerTransport` to `ServerTransport`, `I_ClientTransport` to `ClientTransport`, `I_SessionState` to `SessionState`, `I_RemoteAddress` to `RemoteAddress`.
- Removed `NamedPipeEndpoint` class. Code using `NamedPipeEndpoint` should use `Win32NamedPipeEndpoint` or `UnixLocalEndpoint` instead.
- Refactoring and simplifications to SSL functionality:
  - Added `RCF::setSslImplementation()`, `RcfServer::setSslImplementation()` and `ClientStub::setSslImplementation()` , to configure whether RCF uses Schannel or OpenSSL.
  - Renamed `setSslCertificate()` to `setCertificate()`, and `setSslCaCertificate()` to `setCaCertificate()`.
  - Merged certificate validation callback signature, from `boost::function<bool(OpenSslEncryptionFilter &)>` and `boost::function<bool(SspiFilter &)>` , to `boost::function<bool(I_Certificate *)>`.
  - Merged `setSchannelCertificateValidationCb()` and `setOpenSslCertificateValidationCb()` , to `setCertificateValidationCallback()`.
  - Renamed `setSchannelDefaultCertificateValidation()` to `setEnableSchannelCertificateValidation()`.
  - Added `Win32Certificate::getCertificateName()`, `Win32Certificate::getIssuerName()`, `X509Certificate::getCertificateName()`, and `X509Certificate::getIssuerName()` functions.
  - Referencing arbitrary Windows stores from `PfxCertificate::addToStore()` and `StoreCertificate::StoreCertificate()` , using the `Win32CertificateLocation` and `Win32CertificateStore` enums.
  - Added `StoreCertificateIterator` class, for iterating through certificates in a Windows store.
- Downloads:

  - RCF-2.0.0.2682.zip
  - RCF-2.0.0.2682.tar.gz

RCF 2.0.0.2679 - 2013-03-27

- Add dummy parameters to `RCF::init()` and `RCF::RcfInitDeinit::RcfInitDeinit()`, to trap compiler define mismatches when building RCF as a DLL.
- Server throws an exception immediately, if a client specifies a ping-back interval that is less than the minimum ping-back interval of the server.
- Downloads:
  - RCF-2.0.0.2679.zip
  - RCF-2.0.0.2679.tar.gz

RCF 2.0.0.2678 - 2013-03-07

- Fix compatibility with Boost 1.53.0, by removing usages of `boost::shared_static_cast<>`.
- Increase the maximum number of RCF methods in a RCF interface, from 100 to 200. If you need more than 100 methods, define `RCF_MAX_METHOD_COUNT=200` in your build.
- zlib and OpenSSL libraries are now loaded dynamically at runtime, rather than linked. This means that if you are building RCF with `RCF_USE_ZLIB` or `RCF_USE_OPENSSL`, you no longer need to link to zlib and OpenSSL. RCF will load the zlib and OpenSSL libraries at runtime, on the first execution of any compression code, or OpenSSL-based encryption code.
- Changed Visual Studio demo solution, to build and use a RCF DLL.
- Changed cmake demo projects, to link against `pthread` and `dl` on non-Windows platforms.
- Downloads:
  - RCF-2.0.0.2678.zip
  - RCF-2.0.0.2678.tar.gz

RCF 2.0.0.2675 - 2013-01-03

- Trigger compiler error if user defines `UNICODE` or `_UNICODE` when building RCF on non-Windows platforms.
- Fix compiler error when using `RCF_USE_BOOST_FILESYSTEM` together with `RCF_USE_BOOST_SERIALIZATION`.
- Downloads:
  - RCF-2.0.0.2675.zip
  - RCF-2.0.0.2675.tar.gz

RCF 2.0.0.2673 - 2012-10-28

- Fix bug with `ClientStub::setEnableCompression()` not applying compression in some circumstances.
- Downloads:
  - RCF-2.0.0.2673.zip

- RCF-2.0.0.2673.tar.gz

RCF 2.0.0.2672 - 2012-10-23

- Add demo solution for Visual Studio 2012.
- Add workaround for IOCP bug in Windows 8 and Windows Server 2012.
- Add SF serialization for various C++0x types: `std::unique_ptr<>`, `std::tuple<>`, `std::shared_ptr<>`, `std::array<>`. See Serialization.
- Add PDF version of RCF User Guide.
- Downloads:
  - RCF-2.0.0.2672.zip
  - RCF-2.0.0.2672.tar.gz

RCF 2.0.0.2670 - 2012-10-16

- Fix compiler error occuring on gcc 4.7+.
- Fix compiler error occuring on clang.
- Downloads:
  - RCF-2.0.0.2670.zip
  - RCF-2.0.0.2670.tar.gz

RCF 2.0.0.2668 - 2012-10-15

- Fix compiler error when `RCF_USE_BOOST_SERIALIZATION` is defined but not `RCF_USE_SF_SERIALIZATION`.
- Improve JSON-RPC response message from server, when server fails to parse JSON-RPC request.
- Downloads:
  - RCF-2.0.0.2668.zip
  - RCF-2.0.0.2668.tar.gz

RCF 2.0.0.2665 - 2012-09-29

- Configuration of RCF logging (see Appendix - Logging).
- Disable `SIGPIPE` signals on non-Windows platforms.
- Downloads:
  - RCF-2.0.0.2665.zip
  - RCF-2.0.0.2665.tar.gz

RCF 2.0.0.2664 - 2012-09-24

- Reinstate support for UNIX domain socket transports.

- Preserve last file modification timestamps, when transferring files.
- Add `I_ServerTransport::getInitialNumberOfConnections()` / `I_ServerTransport::setInitialNumberOfConnections()` to get and set the initial number of listening connections that are created when a server transport starts. Default value is 1.
- Fix Winsock compiler errors in Visual Studio 2003 demo projects, by defining `WIN32_LEAN_AND_MEAN`.
- Downloads:
  - RCF-2.0.0.2664.zip
  - RCF-2.0.0.2664.tar.gz

RCF 2.0.0.2661 - 2012-09-02

- Introduce `RCF_USE_BOOST_ASIO` configuration macro, to build RCF against the version of Asio that is in Boost. If `RCF_USE_BOOST_ASIO` is not defined, RCF will be built against the version of Asio that is included in RCF.
- Fix compiler errors when building on Cygwin. For Cygwin builds, `RCF_USE_BOOST_ASIO` is defined automatically.
- Fix bug with asynchronous call dispatching, when used with oneway calls.
- Fix possible busy loop in `RCF_VERIFY()` macro.
- Downloads:
  - RCF-2.0.0.2661.zip
  - RCF-2.0.0.2661.tar.gz

RCF 2.0.0.2648 - 2012-08-19

- RCF 2.0 is a major upgrade of RCF, with signficant under-the-hood changes and a number of new features. This is a summary of the major changes.
- New Feature: Asynchronous Remote Calls
- New Feature: File Transfers
- New Feature: JSON-RPC Support
- New Feature: HTTP/HTTPS Tunneling
- Simplified configuration of transport protocols.
- Simplified configuration of publish/subscribe.
- Simplified configuration of server-to-client callbacks.
- Access controls for servant bindings.
- Improved support for creating server-side session objects.
- Deprecated server-side dynamic binding and session binding.
- `RCF_SINGLE_THREADED` no longer supported.
- Visual C++ 6.0 no longer supported.
- Temporarily not supporting UNIX named pipe transports. This will be reinstated as soon as possible.

- Downloads:
  - RCF-2.0.0.2648.zip
  - RCF-2.0.0.2648.tar.gz

RCF 1.3.1

- This is a bug-fix release for RCF 1.3.
- Fix for compiler error when using `signed char` in RCF method signatures.
- Fix performance problem when using `RCF::SspiFilter`. Multiple small message chunks are now merged into a single larger chunk to improve network performance.
- Improve SF serialization performance. Only call `typeid()` when necessary.
- Reduced SF archive sizes. Encode small integers using a single byte rather than 4 bytes.
- Fix excessive CPU usage when using multithreaded thread pools with Boost.Asio based transports.
- Fix for `boost::any` serialization. Empty `boost::any` instances were causing an exception to be thrown.
- Fix bug in client-side timeout logic when polling network connection using Windows `MsgWaitForMultipleObjects()` function.
- Services can no longer be added or removed while an `RcfServer` is running.
- Fix potential null pointer crash in marshaling logic.
- Rename variables named signals and slots in order not to interfere with QT preprocessor.
- Fix preprocessor redefinition which was causing a compiler warning on OSX.
- Downloads:
  - RCF-1.3.1.zip
  - RCF-1.3.1.tar.gz

RCF 1.3

- Support for IPv6.
- IP-based access rules, to grant or deny access to IP-based servers.
- IP-based client transports able to bind to a specific local interface.
- Client able to query for size of its latest request and response messages.
- User data fields in request and response headers.
- Running multiple server transports on the same thread pool.
- Server-side caching of application objects.
- Default max max message length changed from 10Kb to 1Mb.
- Maximum number of methods in an RCF interface increased from 35 to 100.
- Extended auto-versioning to negotiate archive version as well as runtime version.

- SF serialization
  - Support for `tr1` containers and `tr1` smart pointers.
  - Support for `boost::intrusive_ptr<>`.
  - Serialization of `std::wstring` changed, to use UTF-8.
- Boost.Serialization serialization of `RCF::Exception` now includes error arguments as well as the error message.
- Fix byte ordering of fast vector serialization, for big-endian platforms.
- Improved efficiency of Boost.Asio based server transports - no memory allocations when reading or writing network data.
- For Boost.Asio based server transports, added `AsioServerTransport::getIoService()` to expose the internal `boost::asio::io_service` object.
- Added complete example of server-to-client callbacks, to User Guide.
- Added FAQ on server-side detection of client disconnection.
- Compatibility
  - Support for Visual Studio 2010 compiler.
  - Dropped support for mingw gcc 3.2 & 3.3.
  - Dropped support for Borland C++ Builder 2006.
  - Tested against Boost versions up to 1.45.0.
- Downloads:
  - RCF-1.3.zip
  - RCF-1.3.tar.gz

RCF 1.2

- Support use of Protocol Buffers-generated classes, in RCF interfaces.
- Support Protocol Buffers-based marshaling protocol.
- Support for batched oneway calls.
- Improved error messages, with context-specific arguments.
- Improvements to versioning - SF archive version now passed in message header.
- Signature of SF serialization functions changed, to remove redundant version parameter.
- Calling `serializeParent()` no longer requires type registration.
- `serializeParent()` moved into `SF` namespace.
- Added configuration macro to control behavior on assert (`RCF_ALWAYS_ABORT_ON_ASSERT`).
- Fix serialization of `RemoteException`, when using Boost.Serialization.
- Fix several regressions introduced in 1.1.
  - Fix for thread-safety issue with pingback functionality.

  - Fix for subsecond timeouts on Unix platforms.
  - Fix compilation errors on Windows, if `_WIN32_WINNT <= 0x0500`.
- Tested against Boost versions up to 1.42.0.
- Dropped support for gcc 2.95.
- Updates to the RCF User Guide:
  - Added documentation for publish/subscribe topics.
  - Added documentation for Protocol Buffer support.
  - Rewrote section on Versioning.
  - Documented workaround for internal compiler error with Borland C++Builder 2007.
  - Added release notes.
- Downloads:
  - RCF-1.2.zip
  - RCF-1.2.tar.gz

RCF 1.1

- Ping function added to `RCF::ClientStub`.
- Server-to-client pingbacks, for maintaining connectivity during long-running calls (`RCF::PingBackService`).
- Server-to-client callbacks.
- Dynamic thread pool grows and shrinks, to accommodate client load. User level code no longer needs to call `ThreadManager::notifyBusy()`.
- Progress callbacks on all transports.
- Schannel-based transport filter, for SSL encryption on Windows platforms (`RCF::SchannelFilter`).
- Support for `__attribute__(visibility())` when exporting RCF from shared libraries with GCC 4.x.
- Memory usage optimizations.
- Tested against Boost versions up to 1.39.0.
- Downloads:
  - RCF-1.1.zip
  - RCF-1.1.tar.gz

RCF 1.0

- Support for more compilers (Intel C++ 9 and 10.1 for Windows, GCC 4.3)
- Support for more platforms (FreeBSD and OS X).
- Supports use of UNIX domain sockets as transport (`RCF::UnixLocalEndpoint`).

- Tested against Boost versions up to 1.37.0.
- Downloads:
  - RCF-1.0.zip
  - RCF-1.0.tar.gz

RCF 0.9d

- Win32 named pipe transport implementations (`RCF::Win32NamedPipeEndpoint`).
- Boost.Thread no longer required.
- UDP multicasting and broadcasting.
- SF serialization for `boost::tuple`, `boost::variant`, and `boost::any`.
- Support for exporting RCF from a DLL.
- Tested against Boost versions up to 1.35.0 and Boost.Asio versions from 0.3.8.
- Downloads:
  - RCF-0.9d.zip
  - RCF-0.9d.tar.gz

RCF 0.9c

- Zero-copy, zero-heap allocation core for fast and scalable performance.
- Kerberos and NTLM authentication and encryption on Windows platforms.
- OpenSSL filter, for SSL authentication and encryption.
- Server-side multithreading.
- Server-side session objects.
- Built-in runtime versioning, for backward and forward runtime compatibility.
- Robust publish/subscribe functionality.
- Support for legacy compilers, in particular, Visual C++ 6, Borland C++ Builder 6, and GCC 2.95.
- Support for 64-bit compilers.

Appendix - FAQ

Building

Why is my program trying to link to a Boost library?

If you have defined `RCF_USE_BOOST_SERIALIZATION`, RCF will need to link to the Boost.Serialization libraries.

If you have defined `RCF_USE_BOOST_FILESYSTEM`, RCF will need to link to the Boost.Filesystem and Boost.System libraries.

RCF does not link to any other Boost libraries.

Boost libraries have an auto linking feature that may cause your linker to look for the wrong file to link to. You can define `BOOST_ALL_NO_LIB`, and then explicitly tell the linker which files to link to.

Can I compile RCF into a DLL?

Yes. To export RCF functions from a DLL you will need to define `RCF_BUILD_DLL`.

Why do I sometimes get compiler errors when I include <windows.h> before RCF headers?

Because of header ordering issues with the Windows platform headers `<windows.h>` and `<winsock2.h>`. Including `<windows.h>` will by default include an older version of Winsock, and makes it impossible to subsequently include `<winsock2.h>` in the same translation unit.

The easiest workaround for this issue is to define `WIN32_LEAN_AND_MEAN`, before including `<windows.h>`.

Does RCF compile warning free on level 4 on Visual C++?

Yes, if the following warnings are disabled:

```
C4510 'class' : default constructor could not be generated
C4511 'class' : copy constructor could not be generated
C4512 'class' : assignment operator could not be generated
C4127 conditional expression is constant
```

Can I run RCF on platform XYZ?

Probably, but you may need to make minor modifications to RCF yourself, to accomodate platform specific issues, such as which platform headers to include.

Why do I get linker errors?

On Windows, if you are using TCP transports, you'll need to link to ws2_32.lib. On *nix platforms, you'll need to link to libraries like `libnsl` or `libsocket`.

Does RCF support Visual C++ 6 (Visual Studio 98) ?

No. RCF 1.3.1 is the last RCF version that supports Visual C++ 6.

Does RCF support 64 bit compilers?

Yes.

Does RCF support Unicode builds on Windows?

Yes.

Why do I get duplicate symbol linker errors for my external serialization functions?

If you define external serialization functions in a header file, without the `inline` modifier, and include them in two or more source files, you will get linker errors about duplicate symbols. The solution is to either add an `inline` modifier:

```
// X.hpp
inline void serialize(SF::Archive & ar, X & x)
{
    ...
}
```

, or to declare the serialization function in a header and define it in a source file :

```
// X.hpp
void serialize(SF::Archive & ar, X & x);
```

```
// X.cpp
void serialize(SF::Archive & ar, X & x)
{
    ...
}
```

How do I reduce build times for my application?

If you include RCF headers into commonly used application headers of your own, you may notice an increase in build time, as the compiler will parse the RCF headers once for every source file that happens to include them.

You should only include RCF headers when you need to - in other words, only include them into source files that use RCF functionality. In your application header files, you should be able to use forward declarations, rather than including the corresponding header files.

For example, if you are defining a class `X` with a `RcfClient<>` member, you can forward declare the `RcfClient<>`, and then use a pointer for the member:

```
// X.h
template<typename T>
class RcfClient;

class SomeInterface;
typedef RcfClient<SomeInterface> MyRcfClient;
typedef boost::shared_ptr<MyRcfClient> MyRcfClientPtr;

class RcfServer;
typedef boost::shared_ptr<RcfServer> RcfServerPtr;

// Application specific class that holds a RcfClient and a RcfServer.
class X
{
    X();
    MyRcfClientPtr mClientPtr;
    RcfServerPtr mServerPtr;
};
```

```
// X.cpp
#include "X.h"
#include <RCF/RcfClient.hpp>
#include <RCF/RcfServer.hpp>

X::X() :
    mClientPtr( new RcfClient<SomeInterface>(...) ),
    mRcfServerPtr( new RcfServer(...) )
{}
```

You can then include `X.h` anywhere in your application, without including any RCF headers.

Platforms

Why do I run out of socket handles on Windows XP?

Whenever an outgoing TCP connection is made, a local port number has to be assigned to the connection. On Windows XP, those local ports (sometimes referred to as ephemeral ports) are by default assigned from a range of about 4000 port numbers.

If you create many `RcfClient<>` objects, with TCP endpoints, you will eventually exhaust the available ports, as Windows holds on to them for a short while after the connection is closed.

You should use as few TCP connections as possible (use the same `RcfClient<>` object instead of creating new ones). There are also registry settings on Windows XP that alleviate the issue. Locate the following key:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

, and set

TcpNumConnections = 0x800, MaxUserPort = 65534

After restarting, the system will allow an expanded range of ephemeral ports.

Why does my leak detector think there is a leak in RCF?

Probably because `RCF::deinit()` has not yet been executed.

Is RCF's TCP server implemenation based on I/O completion ports?

On Windows, yes. See Performance.

Does RCF support a shared memory transport?

For local RPC, RCF supports named pipe transports (`Win32NamedPipeEndpoint`), which are backed by shared memory.

Does RCF support IPv6?

Yes - see Transport configuration.

Programming

How do I keep my user interface from freezing when there's a remote call in progress?

Either run remote calls on a non-UI thread, or use progress callbacks to repaint the UI at short intervals. See Progress callbacks.

How do I cancel a long-running client call?

Use a progress callback (see Progress callbacks). You can configure the callback to be called at any given frequency, and when you want to cancel the call, throw an exception.

How do I stop a server from within a remote call?

You can't call `RcfServer::stop()` from within a remote call, because the `stop()` call will wait for all worker threads to exit, including the thread calling `stop()`, hence causing a deadlock. If you really need to stop the server from within a remote call, you can start a new thread to do so:

```
RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(std::string, echo, const std::string &)
RCF_END(I_Echo)

RCF::RcfServer *gpServer = NULL;

class Echo
{
public:
    std::string echo(const std::string &s)
    {
        if (s == "stop")
        {
            // Spawn a temporary thread to stop the server.
            RCF::Thread(boost::bind(&RCF::RcfServer::stop, gpServer));
        }
        return s;
    }
};


int main()
{
    Echo echo;
    RCF::RcfServer server( RCF::TcpEndpoint(0));
    server.bind<I_Echo>(echo);
    server.start();

    gpServer = &server;

    int port = server.getIpServerTransport().getPort();
    // This call will stop the server.
    RcfClient<I_Echo>(RCF::TcpEndpoint(port)).echo("stop");

    return 0;
}
```

How can I make remotely accessible functions private?

Make `RcfClient<>` a friend of your implementation class:

```
RCF_BEGIN(I_Echo, "I_Echo")
    RCF_METHOD_R1(std::string, echo, const std::string &)
RCF_END(I_Echo)

class Echo
{
private:

    friend RcfClient<I_Echo>;

    std::string echo(const std::string &s)
    {
        return s;
    }
};
```

How do I use a single TCP connection with several `RcfClient<>` instancesd?

You can move network connections from one `RcfClient<>` to another. See Access to underlying transport.

Can I publish/subscribe through a firewall?

Yes. As long as a subscriber is able to initiate a connection to the publisher, it will receive published messages.

Why can't I use pointers in RCF interfaces?

Pointers can't be used as return types in RCF interfaces, because there is no safe way of marshaling them. Pointers can however be used as in parameters to a remote call, and behave essentially like a reference.

How do I start a TCP server on the first available port on my machine?

Specify a port number of zero in the `RCF::TcpEndpoint` object passed to the `RcfServer` constructor. After the server is started, retrieve the port number by calling `RcfServer::getIpServerTransport().getPort()`.

How do I forcibly disconnect a client from the server?

Call `RCF::getCurrentRcfSession().disconnect()` in your server implementation.

How do I know that a client has received a published message?

The publisher won't know, because messages are published using oneway semantics.

How do I expose several servant objects having the same interface?

Use servant binding names. See Binding servant objects.

Why is my server only accessible from the local machine, and not across the network?

In the `RCF::TcpEndpoint` passed to your `RcfServer`, you need to specify `0.0.0.0` as the IP address, to allow clients to access it through any network interface. By default `127.0.0.1` is used, which restricts clients to running on the same machine as the server.

At what point does a `RcfClient<>` object connect to the server?

When `getClientStub().connect()` is called, or a remote call is made.

Why does my program crash or assert when exiting?

Probably because your program has a global static object whose destructor is trying to destroy a `RcfClient<>` or `RcfServer` object (or some other RCF object), after RCF has been deinitialized.

Make sure you destroy all RCF objects before deinitializing RCF.

How do I determine what IP address a client is connecting from?

Call `RCF::getCurrentRcfSession().getClientAddress()` in your server implementation.

How do I serialize enums?

SF will serialize and deserialize enums automatically, as integer representations.

Can I use SF to serialize objects to and from files?

Yes, see Serialization.

How do I send a file using RCF?

See File Transfers.

How does a subscriber know if a publisher has stopped publishing?

You can use disconnect notifications, or poll the subscriber for connectedness. See Publish/subscribe.

How can I access the internal `asio::io_service` used by the RCF server?

You can call `AsioServerTransport::getIoService()`:

```
#include <RCF/AsioServerTransport.hpp>

RCF::RcfServer server( ... );
RCF::I_ServerTransport & transport = server.getServerTransport();
RCF::AsioServerTransport & asioTransport = dynamic_cast<RCF::AsioServerTransport &>(transport);
boost::asio::io_service & ioService = asioTransport.getIoService();
```

The `io_service` will be destroyed when the `RcfServer` is stopped.

How do I detect client disconnections, from server-side code?

When a client disconnects, the associated `RcfSession` on the server is destroyed. You can use `RcfSession::setOnDestroyCallback()`, to notify your application code when this happens.

```
void onClientDisconnect(RCF::RcfSession & session)
{
    // ...
}

class ServerImpl
{
    void SomeFunc()
    {
        // From within a remote call, configure a callback on the current RcfSession.

        RCF::getCurrentRcfSession().setOnDestroyCallback( boost::bind(
            onClientDisconnect,
            _1));
    }
}
```

`boost::bind()` can be used to pass extra arguments to the callback function.

How do I pass a security token from the client to the server, without changing the RCF interface?

You can use user data slots (`ClientStub::setRequestUserData()`, `RcfSession::getRequestUserData()`), to pass application specific data from the client to the server. See Per-request user data.

Can I send `std::wstring` objects between Linux and Windows?

In RCF 1.2 and earlier, Unicode strings stored as `std::wstring`, were serialized as a sequence of `wchar_t` characters. This would cause serialization errors if the client and server were running on platforms with different `std::wstring` encodings, such as Linux and Windows.

In RCF 1.3 and later, `std::wstring` is serialized in UTF-8 encoding, with conversions to and from UTF-16 or UTF-32, depending on the platform. See Advanced Serialization - Unicode strings.

Can I send UTF-8 encoded `std::string` objects between Linux and Windows?

Yes - SF serializes `std::string` a sequence of 8-bit characters, so it doesn't matter whether the encoding is ASCII, ISO-8859-1, UTF-8, or anything else. Your application just needs to be aware of the encoding of its own strings,

Miscellaneous

Why are there double parentheses in so many of the examples?

The following code snippet will cause compiler errors:

```
int port = 0;
RcfClient<I_Echo> client( RCF::TcpEndpoint(port));
RCF::RcfServer server( RCF::TcpEndpoint(port));

// Will get compiler errors on both of these lines...
client.getClientStub();
server.start();
```

Because of a C++ language idiosyncracy, the declarations of `client` and `server` are actually interpreted as function declarations, taking a `RCF::TcpEndpoint` parameter named `port`. C++ compilers interpret the declarations this way to maintain backwards compatibility with C.

To clear up this ambiguity, we need to put in extra parentheses around the `RCF::TcpEndpoint`:

```
int port = 0;
RcfClient<I_Echo> client(( RCF::TcpEndpoint(port)));
RCF::RcfServer server(( RCF::TcpEndpoint(port)));

// Now its OK.
client.getClientStub();
server.start();
```

This quirk of the C++ language is sometimes referred to as "C++'s most vexing parse".