librascal: a connection library

Justin Forest <justin.forest@gmail.com>
2006-10-29

Table Of Contents

Abstract

Almost every single application these days involves data exchange over an IPv4 network: from obvious client-server applications, such as communication programs, to regular self-updating desktop applications. However, the network interaction is usually the most boring and error prone part of the code. Writing that code from scratch takes a significant amount of time, optimizing it for performance takes even more time and porting it to a different operating system usually involves a complete rewrite, again. This paper describes a library which helps deal with the mentioned problems, making the interaction with TCP/IP networks easy and efficient.

Introduction

The name «librascal» is an acronym for «**r**ealtime **as**ynchronous **c**onnection **a**bstraction **l**ayer». Here is the explanation:

1. «Realtime» means that the library never delays the execution of your application, unless you ask it to. Most functions either finish immediately, or are scheduled for background execution; however, there are functions that wait for tasks to finish, even if it takes time; these functions are designed for infrequent use.
2. «Asynchronous» means that the library operates in the background and all events are handled outside the main thread of your program, if there is one; the main thread can serve a GUI, co-ordinate the requests or perform other functions not related to the networ.
3. «Connection abstraction layer» means that the library provides a complete set of classes for networking, and your application does not need to deal with the operating system's networking functions and structures at all.

The library simplifies such tasks as establishing and accepting TCP connections, resolving host names using DNS, discovering services[1], writing event-driven and multi-threaded applications (see below), and using the most efficient TCP/IP interface (IOCP for Windows NT, kqueue for modern Unix-like systems, etc). This all is achieved with a very simple API.

The application defines a separate class, based off an abstract base class provided by the library, for each logical type of a network client or a server, such as an HTTP server or an IMAP client. The library talks back to the application by calling virtual methods, one for each type of network activity (a data packet has been sent, a host name was resolved, etc).

The DNS subsystem is written from the ground up to allow using custom DNS servers, such as OpenNIC or whatever else. By default, the system wide settings are used, but with a few calls the library can be reconfigured to use alternative DNS.

1 *Service Discovery is a technique currently used by many technologies, from Microsoft's Active Directory to Jabber (Google Talk is a fine example). It provides service developers with transparent mirroring, load balancing and port mapping. Search the internet for «DNS-SD» and «DNS SRV RR» for more information.*

History

The first version of the library was developed during 2004 and 2005. It had a C style interface and was only available as a shared library. That version was abandoned.

This document describes the second version of the library. It has been rewritten as a set of C++ classes to achieve the following goals:

1. Simplify the API. Using C style callback handlers, passing pointers to request descriptors and context identifiers to the library isn't even remotely an efficient style of coding. Besides, it's the XXI century, and pure C should be left for low-level tasks, such as kernel and device drivers.
2. Simplify error handling. The C language is bound to the use of function return codes, which means having a complicated check after each library function call. C++ exceptions, on the other hand, make it much easier to focus on the normal workflow logic, handling unrecoverable errors separately.
3. Since librascal is now a static library, a lot more functions can be included with no cost to applications which never use them. The library is still a basic wrapper, not a framework, but handling most frequent tasks is also good.

Using the library

This section contains a number of show cases to illustrate the most typical uses of the library. For a complete class reference, see the next section.

Required files

To use the library, you must include «rascal.h» in the main header file of your application (or in each module that uses the functions provided by the library). It's a good idea to include it in the precompiled header (when using Microsoft Visual Studio, this is typically «stdafx.h»). The next step is to add «librascal.lib» to the list of external libraries.

There are no more files required to use librascal.

Base classes

The main class provided by the library is RClient, short for «RASCAL Client». You must define your own classes based off Rclient for each TCP/IP protocol you are going to work with. Your classes *must* provide a destructor. For each type of workflow you'll also need to override a few other virtual methods, which are described below.

The default constructor for the RClient class looks like this:

- RClient::RClient(bool stream = true);

  The parameter specifies whether the object will be working with TCP connections (true) or UDP datagrams (false). This and the fact that UDP loses data sometimes is the only difference between the two.

Another frequently used class is RAddr, which holds and works with numeric host addresses.

Finally, the RCore class provides a number of static methods used to interact with the library itself, for the purpose of setting up the suitable environment and so on.

Exceptions

When the library can report the problem immediately after the call was made, it does so using the C++ exceptions. Errors in scheduled operations are reported using different means, namely the dedicated callback methods.

The base class for exceptions in librascal is RException. You may find the following methods interesting:

1. virtual const char * GetMessage() const;

   Returns a pointer to the error message associated with this exception. This is typically the only method you need.

Sample use:

```
try {
   // some librascal related code
} catch (RException *e) {
   std::out << "error: ";
   std::out << e->GetMessage();
   std::out << endl;
}
```

Setting up threads

The default threading model is one working thread per CPU, leaving the main thread of the application to deal with the GUI or other tasks. Each working thread is dedicated to reading from or writing to the network, and dispatching the callback events.

The library itself is thread safe and re-entrant; however, the applications using it must be designed with multi-threading in mind: on systems with more than one CPU callbacks can and, most probably, will be called in the context of a different thread each time. If the callback handlers interact with global, process-wide data, a locking mechanism must be used. Alternatively, if networking is not the weakest part of the library, it can be turned off, switching to only one worker thread which delivers all callbacks.

Generally, you should not want to change the threading model. But if you need to do that, use one of the following calls:

1. RCore::SetMT(RCore::mtSingle);

   Switches the library to the single threaded mode. This means that all callback methods will be fired in the context of a single thread; however, this will still be a dedicated worker thread, separate from the main thread of your application.

2. RCore::SetMT(RCore::mtOptimal);

   Switches the library to the most efficient threading model. This means one worker thread per CPU or a CPU core. This mode is enabled by default and should only be explicitly reset if the single threaded mode was previously be activated.

Changes to the threading model may be a little delayed if there is a heavy load. When a worker thread has finished processing the available data, and it is not on the first CPU, it will quit instead of waiting for more data.

Connecting to a remote host

To connect to a remote host, you must use one of the following methods:

1. void Connect(RAddr &addr);

   The input parameter must be a full description of the remote connection

endpoint.

2. void Connect(const char *addr);

   The input parameter must be a null-terminated string containing either a host name, or an IP address in the dotted octet notation. It must also be followed by a colon and the TCP port number.

3. void ConnectService(const char *domain, const char *service, const char *proto);

   This method schedules a connection attempt with a preceeding service discovery. The input parameters describe the service, to which you want to connect. The library performs a service lookup, then attempts to connect to every listed mirror, using the algorythm defined in RFC2782, until a connection is successfully established.

Each of these methods throws an exception if the request could not be scheduled, otherwise a callback will follow as soon as the connection either succeeds or fails. There are two virtual methods which handle a finished connection request:

1. bool OnConnect(const RAddr &addr);

   Handles established connections. The parameter contains a description of the remote connection endpoint.

   If the connection is still desired, the method must return true. If false is returned, the connection is closed; the object, however, remains valid (ready for another connection attempt or other operation).

   The default implementation of this method closes the connection and deletes the object for which the connection occured.

2. void OnConnectError(const char *msg);

   The parameter points to the error message which occured during the connection attempt. The default implementation of this method destroys the object; if your only reaction to a failed connection attempt is to delete the object, just leave this method unimplimented.

Example:

```
class test : public RClient
{
public:
  bool OnConnect
     (const RAddr &);
  void OnConnectError
     (const char *);
  void Init();
};

// Connection successful.
bool test::OnConnect(const RAddr &) {
  std::out << "connected ok.";
  std::out << endl;
}

// Connection failed.
void test::OnConnectError
(const char *msg) {
  std::out << "error: " << msg;
  std::out << endl;
}

// This is called from the outside
// to start connecting.
void test::Init() {
  Connect("127.0.0.1:80");
}
```

Accepting connections

Waiting (listening) for incoming connection is the main task of every server application. There are two ways of accepting incoming connections:

1. void Listen(const RAddr &);

   Starts listening on the specified address. The parameter must contain a complete address specification, with the port number.

   void Listen(const char *);

   The parameter must contain either a host name or a numeric IP address in dotted octet notation, followed by a colon and the port number.

   In both cases, the address can be “0.0.0.0” to accept all connections coming to the specified port number. If a different address is specified, only clients connecting to that particular IP address will be accepted, the rest will be ignored without triggering any callbacks.

2. void Listen(const char *domain, const char *service, const char *proto);

   Starts listening to all locally available addresses, listed as mirrors for the specified service.

When an incoming connection is found, the following

method of the listener is called:

1. RClient * OnAccept(const RAddr &from, const RAddr &to);

   The from and to parameters describe the local and remote endpoints of the connection. To accept the connection, a pointer to a new object derived from RClient must be returned; a null pointer means the connection must be dropped.

   The default implementation of this method returns a null pointer, effectively dropping all incoming connections.

Sample use:

```
class test : public RClient {
public:
   RClient * OnAccept(
     const RAddr &from,
     const RAddr &to);
   void Init();
};

// Start listening.
void test::Init() {
   Listen("0.0.0.0:80");
}

// Process incoming connections.
RClient * test::OnAccept
(const RAddr &from, const RAddr &) {
   return new RClientEx(from);
}
```

Receiving data

1. void OnRead(unsigned int);

Retreive:

1. bool GetData(char *, unsigned int &);
2. bool GetString(char *, unsigned int &);

Sending data

1. void OnSend(unsigned int);

Resolving host names