U.S. DEPARTMENT OF ENERGY

PNNL-22591

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

The IRCConnection Utility Class

DES-0009

Revision 2

Charlie Hubbard

April 2010

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

The IRCConnection Utility Class

Charlie Hubbard

DES-0009
Revision 2
April 2010

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under
Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Contents

1 Introduction

This document describes the `IRCConnection` utility class defined and implemented in *ircLib.h* and *ircLib.cpp* respectively.

There is an old yet powerful open-specification chat facility, called Internet Relay Chat (IRC), that allows any user with appropriate client software to connect to an IRC network and "chat" with other connected users (exchange typed text messages in real time). IRC has existed on the Internet since 1989 (finally being standardized in 1993) and remains a popular real-time chat service to the present day.

A couple of features of IRC that distinguish it from other chat services (Java-driven web applications or something like Microsoft Instant Messenger for example) is that IRC clients and servers are typically free, and the specifications for an IRC client, IRC server and the IRC messaging protocol are not proprietary. They are openly available on the web, and anyone can use them to develop custom IRC clients or servers. IRC is also an extremely powerful messaging service, offering many capabilities not found in other, proprietary solutions.

Many open-source implementations of IRC servers and IRC clients exist. For some time now, the PNNL development team has been running a private IRC server (The *InspIRCd* server — see `http://www.inspircd.org`) in-house to provide a convenient way for team members, who are split across multiple buildings, to communicate.

The `IRCConnection` class was developed to provide a framework for the development of custom IRC client applications. The chief driver for this was to provide a way for the the system event logging server (see design document ***DES-0007, The System Event Logger***) to report its messages to an IRC channel in addition to writing them to the system's event log file.

The `IRCConnection` class provides methods that allow the IRC application writer to connect to an IRC server, send "chat" (and other) messages to the server, and receive and act upon messages received from the server.

In order to gain a good understanding of the `IRCConnection` class, the reader should already be familiar with the IRC communication protocol. Unfortunately, a thorough description of the protocol is beyond the scope of this document. However, like so many early Internet protocols and capabilities, the IRC specification is published as an RFC (request for comment) document, and can be easily found on the web as document *RFC-1459*. Developers should be familiar with that document before making changes or additions to the `IRCConnection` class. Also, there are many places throughout the class' implementation where it is necessary to parse strings. This is typically done using a suitable regular expression. In the code base, regular expressions are implemented by the `RE_c` utility class which is fully described by design document ***DES-0008, The RE_c Utility Class***. Developers that want to create, modify or just understand the regular expressions in the `IRCConnection` code, should read that document for details.

2 Message-Driven Architecture

Many IRC client applications are *purely reactive*. That is to say, they sit blocked on the TCP connection to the IRC server, waiting for messages to arrive. When a message does arrive, it is handed off to a specific message handler for processing. Then the client goes back to sleep until the next message arrives. The client application reacts to server messages and does nothing else.

2.1 The Need for Threads

Not all clients are *purely reactive*. Some clients cannot afford to sleep between arriving server messages because they have local tasks they need to perform in the interim. In this case, the task of processing messages arriving from the IRC server is normally handled by a separate thread, and the main-line thread does whatever local processing is necessary.

The `IRCConnection` class provides methods for this situation that will spawn a thread to handle incoming server messages and then immediately return control to the main-line thread to perform whatever additional processing needs to be performed. Thread management is completely handled by the class, so users of the class do not have to explicitly spawn threads or provide their own mutex protection of shared resources.

2.2 Message Handlers

IRC servers generate dozens of different messages to clients. Of these, typically only a small subset are of interest to a particular client application. For instance, a custom IRC client that does nothing but report the current weather conditions to a channel when asked, doesn't care when specific people join or leave the channel. Nevertheless, the IRC server will send such a client a "JOIN" or "PART" message every time someone comes or goes. For this custom client, "JOIN" and "PART" messages can be safely ignored.

IRC server messages are of two types. The common messages are known by short text names like "JOIN," "PART," "PRIVMSG," etc. Less common messages are identified by a three digit, ASCII-encoded numeric code (that is to say, the message name consists of just three digits, but it is still a text string). Internally, the `IRCConnection` class maintains an STL map that maps the name of a message to the class method that handles that specific kind of message. As new messages arrive from the server, the rule is that if they don't have an entry in the message handler map, then they are simply ignored.

2.2.1 Message Handler Fingerprint

IRC messages consist of three parts — an optional *prefix* part, a *command* part, and a *params* part (see *RFC-1459* for full details). When messages from the IRC server are delivered to an `IRCConnection`-based client, they are immediately parsed into these components and are

available to the client writer as instances of the helper classes `PrefixPart`, `CommandPart`, and `ParamsPart`. Once parsed, messages are delivered to whatever message handler has been assigned to handle the message (if any).

All message handlers have the same fingerprint (return value plus parameter list). It is defined by the `MessageHandler` type definition in *ircLib.h* and reproduced below.

```
typedef void (IRCConnection::*MessageHandler)(PrefixPart&,
                                              CommandPart&,
                                              ParamsPart&);
```

As this example shows, all message handlers return type `void` and they all take three parameters — references to the *prefix*, *command* and *params* components that all incoming messages are split into. Message handler methods are typically declared as *virtual* methods so that they can be easily overridden by derived classes and still properly called by the base class message dispatcher.

2.2.2 Built-in Handlers

The `IRCConnection` class provides built-in message handlers for "NICK" and "PING" messages, because virtually all client applications will want to handle these two. For all other message types, custom handlers need to be provided. For that reason, all real-world IRC client interfaces derive a custom class based on the `IRCConnection` class rather than instantiating `IRCConnection` directly. Any additional message handlers the application needs are then added as new methods of the derived class.

2.2.3 Registering Handlers

Authors of `IRCConnection`-based clients register new message handlers using a special macro defined in *ircLib.h* called `ADD_HANDLER()`. It takes two parameters — the name of the message to be handled, and a pointer to the method that will handle messages of that type. The following code snippet shows how the `IRCConnection` constructor registers the two built-in handlers discussed above.

```
ADD_HANDLER("NICK", &IRCConnection::NickHandler);
ADD_HANDLER("PING", &IRCConnection::PingHandler);
```

The two handlers registered in the above code snippet correspond to the two `IRCConnection` class methods `NickHandler()` and `PingHandler()`, the prototypes for which appear below.

```
void NickHandler(PrefixPart &prefix, CommandPart &command, ParamsPart &params);
void PingHandler(PrefixPart &prefix, CommandPart &command, ParamsPart &params);
```

2.3 Message Component Classes

Three message component classes (defined in *ircLib.h* and implemented in *ircLib.cpp*) are used to hold the three components into which all incoming messages are split. We look at these in more detail in this section.

2.3.1 The `PrefixPart` Class

The `PrefixPart` class is used to hold the prefix component of an incoming message. The prefix component is optional, so not all messages carry one. It is up to the authors of individual message handlers to know whether or not the specific message type they are handling will have a valid prefix component or not. The `PrefixPart` class definition appears below.

```
class PrefixPart {
public:
   PrefixPart(string s);
   string GetName();
   string GetUser();
   string GetHost();
private:
   string prefix;
   RE_c   re;
};
```

The prefix portion of a server message (when it exists at all) contains one to three fields.

```
name[!user][@host]
```

Where the "!" and "@" characters are delimiters identifying and separating the fields. Note that the "user" and "host" fields are both optional. As for the fields themselves,

- `name` — This field is the nickname of the user that generated the message (or the server name if this was a server generated message).
- `user` — This is the username of the user (that is, the user's account name on his local system).
- `host` — This is the name of the user's host system (or IP address in dotted decimal notation).

As a convenience, the `PrefixPart` constructor automatically splits out whichever of the above fields actually exist and make them available through the three accessor methods, `GetName()`, `GetUser()`, and `GetHost()`. These methods return empty strings if their corresponding field daes not appear in the prefix.

2.3.2 The `CommandPart` Class

The `CommandPart` class is used to hold the command component of a message. This is the component that carries the name of the message ("JOIN," "PART," "376," etc.). The `CommandPart` class definition appears below.

```
class CommandPart {
public:
   CommandPart(string s);
   string GetCommand();
private:
   string command;
};
```

The command portion of a server message contains exactly one field; the name of the message. This field is passed in to the class constructor and is available to message handlers though the `GetCommand()` accessor method.

2.3.3 The `ParamsPart` Class

The `ParamsPart` class is used to hold the *params* component of a message. Its definition appears below.

```
class ParamsPart {
public:
   ParamsPart(string s);
   string GetTarget();
   string GetMessage();
   string GetRaw();
private:
   string params;
   RE_c   re;
};
```

The params portion of a message has no fixed fields, and how it is interpreted depends on the specific message in question. Nevertheless, there is one message, "PRIVMSG", that is so common, it is worthwhile providing accessor methods for its fields. The params component of a PRIVMSG message is formatted as follows.

```
target :message
```

The space and colon after "target" are string literals. As for the fields themselves,

- `target` — This can either be the name of a channel, or the name of a user. Most PRIVMSG messages are from users that are posting normal chat messages in the channel, so most of the time, this field contains the channel name. It is also possible to

generate a truly private message (one that only the target user can see). In this case, this field contains the nickname of the target user.

- `message` — This is the text of the message the sender is sending either to the target channel or target user.

For other message types, it is up to the developer of the corresponding message handler to know how to properly parse out and interpret the fields in the message's params component. In these situations, the `GetRaw()` accessor method can be used to get the complete params string.

3 Generating Client Messages

In addition to being able to respond to server messages, clients also need to be able to send new messages of their own back to the server. The IRC protocol defines several client messages (see *RFC-1459*), and the `IRCConnection()` class has convenience methods for generating most of them. The method definitions for those methods that generate client messages to the server are shown below.

```
string Away(string reason = "");
string Invite(string nick, string channel);
string Join(string channel, string key = "");
string Kick(string channel, string nick, string reason = "");
string List(string channels = "");
string Message(string target, string message);
string Names(string channel);
string Nick(string newNick);
string Pass(string password);
string Part(string channel, string reason = "leaving");
string Pong(string daemon);
string Quit(string reason = "Bye!");
string Topic(string channel, string topic = "");
string User(string uname, string rname);
string Whois(string nick);
```

Of these, the `Message()` method is probably the one most often used by clients, because it is the one that actually sends text either to be displayed in an IRC channel, or as a private message intended for a specific user. The other methods each correspond to one of the client messages defined in *RFC-1459*. Certainly not every possible client message is represented in the above list. IRC client application authors can use the `RawMessage()` method to send custom message strings to the server as necessary.

As the above example demonstrates, all client message generation methods return an STL string. In all cases, this string contains the text "ok" if the call was successful, or a text description of the failure if unsuccessful.

Not much more will be said about these methods here. They are typically short, and they are well commented in the source code. The reader should refer to the method implementations

in *ircLib.cpp* and the descriptions of the various messages in the IRC protocol document (*RFC-1459*) for more information.

4 Low-Level Communication Details

Clients based on the `IRCConnection` class communicate with the IRC server over a TCP connection. A few methods and class variables are used to manage and use the TCP connection, and they are described below.

4.1 The `Connect()` Method

```
string Connect(string server, unsigned short port, string password = "");
```

The `Connect()` method establishes a connection to the IRC server and then runs in a continuous loop, receiving server messages and dispatching them to the appropriate message handler. `Connect()` does not return to the caller except on an error condition. For that reason, it is really only suitable for client applications that are purely reactive. Applications that need to perform other tasks in addition to handling server messages *should not* call `Connect()` directly. Instead, they should call the `StartSession()` method described in section 4.2.

The `Connect()` method takes three parameters.

- `server` — This is the host name of the IRC server to connect to (or the server's IP address specified in dotted decimal notation)
- `port` — This is the TCP port on the server with which `Connect()` will try to establish a connection
- `passwd` — This is the password needed to successfully log in to the IRC server. Most servers do not require a password, in which case this parameter should just be set to an empty (zero byte) string. As a convenience, the empty string is the default value for this parameter, so, when connecting to a server that requires no password, the parameter can be omitted entirely.

4.1.1 Message Processing

Once `Connect()` has established a connection to the server, it *does not* return unless an error occurs. Instead, it enters a continuous loop where it endlessly waits for new messages to arrive from the server. As messages arrive, they are parsed into *prefix*, *command* and *params*

components by the `reMsg` regular expression[1] and then passed on to appropriate message handlers to be dealt with (see section 2.2).

4.1.2 Error Handling

`Connect()` will return with an error if it cannot establish a connection to the IRC server or if the connection to the server is closed or otherwise lost. Applications that use `Connect()`, often wrap the call in a loop that will, after some suitable time delay (60 seconds is typical) call `Connect()` again. In this way, clients that lose their connection to the server will automatically attempt to reestablish the connection until they succeed or until the application exits.

4.2 The `StartSession()` Method

```
string StartSession(string server, unsigned short port, string password = "");
```

The `StartSession()` method is similar to the `Connect()` method in that it establishes a connection to the IRC server and then runs in a continuous loop, receiving server messages and dispatching them to the appropriate message handler. The difference is `StartSession()` first spawns a separate thread and then performs the message receipt-and-dispatch task in that thread. `StartSession()` is expected to be used in place of `Connect()` by those client applications that need to perform other tasks in addition to processing messages from the IRC server.

The `StartSession()` method takes the same three parameters that `Connect()` does.

- `server` — This is the host name of the IRC server to connect to (or the server's IP address specified in dotted decimal notation)
- `port` — This is the TCP port on the server with which `Connect()` will try to establish a connection
- `passwd` — This is the password needed to successfully log in to the IRC server. Most servers do not require a password, in which case this parameter should just be set to an empty (zero byte) string. As a convenience, the empty string is the default value for this parameter, so, when connecting to a server that requires no password, the parameter can be omitted entirely.

Connection management is handled automatically when using the `StartSession()` method. If no connection can be made or the connection is lost, `StartSession()` automatically tries to reconnect to the server at 60 second intervals. This continues until the connection is

[1] The `reMsg` member variable is an instance of the regular expression utility class `RE_c`. See ***DES-0008, The RE_c Utility Class*** for a full description.

reestablished or until the thread is shut down with the `EndSession()` method (described below).

4.3 The `EndSession()` Method

```
void EndSession();
```

When using the `StartSession()` method, a separate, detached thread is created to handle IRC server message processing. The thread processes server messages according to which message handlers the application author has registered, but there is no direct interaction between the thread and the client's main-line execution thread — with one exception; the `EndSession()` method provides the main-line code a way to close down the message handling thread.

The `EndSession()` method takes no parameters and returns no value. However, there are two things application writers will need to know. First, if no message handling thread exists, `EndSession()` immediately returns without doing anything. Secondly, if a message handling thread does exist, `EndSession()` *will not* return, until the message handling thread has exited.

4.4 Low-Level Reads and Writes

At the lowest level, two private methods are used to read and write messages from/to the TCP connection.

4.4.1 The `ReadMessageBuffer()` Method

```
string ReadMessageBuffer(string &msg);
```

`ReadMessageBuffer()` is a private helper method to the `Connect()` method. It is only called from there. On successful return from a call to `ReadMessageBuffer()`, the `msg` parameter will contain a complete server message received from the server and will return the string "ok." If the call is not successful, then a description of the problem is returned and the value of `msg` is undefined.

According to *RFC-1459*, all messages arriving from the server are terminated with a carriage-return/line-feed combination. `ReadMessageBuffer()` reads characters off the TCP connection, one at a time, looking for that pattern. As new characters arrive, they are placed into a local character array called `inBuff`. When the CR/LF pair is seen, the contents of `inBuff` (not including the CR/LF pair) are copied to `msg` and the method returns.

4.4.2 The `WriteMessageBuffer()` Method

```
string WriteMessageBuffer(const string &msg);
```

`WriteMessageBuffer()` takes as its sole parameter, a reference to a string containing the message to be sent to the server. On success, the method returns the string "ok." If the call fails, then a text description of the problem is returned. Callers should not include the CR/LF message termination characters in messages sent using `WriteMessageBuffer()`. The method automatically appends these to the end of the message before sending it to the server. `WriteMessageBuffer()` is called directly by all of the client message generation convenience methods (see section 3).

5 An Example Client

In this section we'll describe a simple, `IRCConnection`-based client that connects to an IRC server, joins a chat channel, and then reacts in certain ways to other users in the channel. The code will be presented in segments for easy discussion, but nothing is left out. All the following code segments, when appended together, should form a complete program ready to be compiled.[2]

5.1 Defining the `MyIRC` Class

```
#include "ircLib.h"
class MyIRC: public IRCConnection {
public:
   MyIRC(string nick, string uname, string rname);

protected:
   void EndMotDHandler(PrefixPart &prefix, CommandPart &command, ParamsPart &params);
   void PrivmsgHandler(PrefixPart &prefix, CommandPart &command, ParamsPart &params);
};
```

The above code snippet gets us started. Here we have included *ircLib.h* to get access to the `IRCConnection` class definition, and then we have derived a new class from it called `MyIRC`. `MyIRC` defines two message handlers that will be described more completely below.

5.2 The `MyIRC` Constructor

```
MyIRC::MyIRC(string nick, string uname, string rname) :
   IRCConnection(nick, uname, rname)
```

[2]Note that in order to successfully compile, the code will need to link in *ircLib.o* and *re.o* object files.

```
{
   ADD_HANDLER("376", &MyIRC::EndMotDHandler);
   ADD_HANDLER("PRIVMSG", &MyIRC::PrivmsgHandler);
}
```

The above snippet shows the implementation of the class constructor. The constructor takes three parameters.

- `nick` — This is the nickname the client wants to be known by when connected to the server.
- `uname` — According to *RFC-1459*, this is *supposed* to be the user's login name on the system from which he is connecting. But not all systems support the "login name" concept, and even fewer users want to actually give out that information, so quite often this parameter is just filled with some phony name.
- `rname` — According to *RFC-1459*, this is *supposed* to contain the real name of connecting user, but few people want to give that information out, so often any phony name is used instead.

The constructor does nothing with the input parameters but pass them down to the base constructor. The base class stores them to be used later during the connect.

The constructor registers two message handlers — one for message "376" and one for the "PRIVMSG" message. More will be said about these in a minute. Also keep in mind that the `IRCConnection` class itself registers handlers for "NICK" and "PING" messages, so, after our constructor completes, our client will have a total of four registered message handlers. Server messages other than these four types will be silently discarded.

5.3 The End of Message-of-the-Day Handler ("376")

```
void MyIRC::EndMotDHandler(PrefixPart &prefix, CommandPart &command, ParamsPart &params)
{
   Join("#test");
}
```

The above code snippet shows our handler for the "376" message. Message "376" has to do with the server's "message of the day" feature. This amounts to several lines of text (defined by the operators of the server) that the server displays to any new client that connects to it. Three messages are involved in handling the "message of the day" text — "375," "372" and "376." Receipt of a "375" message is the client's queue that the server is about to start sending its "message of the day" text. Each line in the "message of the day" is delivered as the payload of a "372" message. Once the full text has been delivered, the server sends a "376" message to indicate to the client that all the text has been delivered.

For our example client, we are not really interested in the content of the "message of the day" at all, which is why we have not registered handlers for "375" or "372" messages. However,

receipt of the "376" message is a handy way to verify that client has successfully connected to, and been accepted by the server. When our client receives the "376" message, it calls our handler. We respond by sending a "JOIN" client message to the server with the intention of joining a chat channel called "#test." When the server processes this message, our client will be added to the list of users that are currently in the "#test" channel. If there is no existing "#test" channel, one will be created.

5.4 The "PRIVMSG" Message Handler

```
void MyIRC::PrivmsgHandler(PrefixPart &prefix, CommandPart &command, ParamsPart &params)
{
   if (params.GetTarget() == "TestBot") {
      Message(prefix.GetName(), "Not now, I have a headache");
   }
   else {
      if (params.GetMessage() == "hi") {
         Message("#test", "Hey, how's it going?");
      }
   }
}
```

The code snippet above is our client's message handler for the "PRIVMSG" server message. "PRIVMSG" messages are used both to deliver chat messages to individual users that are connected to the server (private messages) and to deliver public messages that everyone in the current channel will see. The difference is in the "target" portion of the message (part of the message's *params* component — see section 2.2). If the target is our client's nickname ("TestBot" in this case), then this is a private message sent just to us. We will see the message, but others in the channel will not. If the target is the name of the channel ("#test" in this case), then the message is not private, and everyone in the channel will see it.

Our message handler reacts differently to these two different cases. If the message is a private one directed only to us from some other user, we always send a private message back to the sending user that says "Not now, I have a headache". If, on the other hand, the message was a public message, we look to see if the text of the message was "hi" (that is to say, if someone typed "hi" into the channel). If so, we send a public message of our own back to the channel with the text "Hey, how's it going?" If the public message was anything other than "hi," we do nothing at all.

5.5 The `main()` Function

And that's it for the class methods. All that remains is to provide a little `main()` function to make this a stand-alone program. Because our client is strictly reactive. It does nothing but look for and respond to messages coming from the server. We don't need to worry about running the connection management and message handling tasks in a separate thread. If we

did, we would call `StartSession()`. As it is, we can call `Connect()` directly. The code for our `main()` appears below.

```
int main()
{
   string errorMessage;
   MyIRC myIRC("TestBot", "testbot", "Test");

   errorMessage = myIRC.Connect("localhost", 6667, "");
   printf("Connect() returned with error '%s'\n", errorMessage.c_str());
}
```

Our `main()` consists of only a few lines. On the second line, we create an instance of our IRC client class called `myIRC`, passing to it the nickname we want to be known by on the server. Phony names are used for our client's local user name (doesn't really have one) and real name (again, doesn't really have one).

On the next line, is our call to `Connect()`. Recall from section 4.1 that this does two things. First it attempts to establish a TCP connection to the specified IRC server (in the code we are just connecting to *localhost*, implying that we are running an IRC server on the same machine the client code is running on), at the specified TCP port. If the connection is successful, then `Connect()` enters a continuous loop, endlessly waiting for messages to arrive from the server and processing them when they do. If the call to `Connect()` returns, it either means the server rejected our connection request, or disconnected us at some later time for some reason. If that happens, our test client just prints the problem description and exits. However, for real clients, quite often the call to `Connect()` is placed inside a continuous loop together with a call to `sleep(60)`, so that if the connection is ever lost, the client will wait 60 seconds and then automatically attempt to reconnect. It'll continue trying to reconnect at 60 second intervals until it is successful or the client process is killed.

This has been a simple example of a purely reactive IRC client application. For an example of a client that is not purely reactive, and therefore must call `StartSession()` instead of `Connect()`, see design document ***DES-0007, The System Event Logger***.

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

U.S. DEPARTMENT OF
ENERGY

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov