HowTo POP3

by Randy Charles Morin

Some of the simplest, yet very rich communication protocols were born on the Internet. This is the second article in a series where I will write on these simple communication protocols. This article will focus on the POP3 (Post Office Protocol version 3) protocol. This protocol is the most often used protocol for receiving email over the Internet.

The first step I have to do is clean up our Smtp class that will created last month, in order to reuse some of the base functionality in our new Pop3 class. I'm going to divide the Smtp class into two class, a Socket and Smtp class. The Socket class will contain all the code implementing the Socket access protocol and the Smtp class will contain all the code implement the SMTP specific code.

In the previous Smtp code, I presented one exception called SmtpException and in this new code I have renamed this class SocketException. I have also removed the SmartSocket class and implemented the connection and disconnection algorithms in the constructor and destructor of the Socket class. I have also formalized the Write and Response functions as methods of the Socket class.

Lastly, I created a new Connect methods, one to connect via a service name and one to connect via a specific port. An interesting concept in the IP world is the use of well-known ports. SMTP as an example is known to operate over port 25 and POP3 is known to operator over port 110. It is usually common knowledge that HTTP operates by default over port 80. These are called well-known ports. All three protocols can operator over other ports, but they most often operate over their well-known port. This is why I've created two Connect methods. The first method takes the service name, like SMTP, POP3 or HTTP and converts them to their well-known port. The second Connect method takes actual port as a parameter. This opens up the possibility of using our Socket class over ports that are not well-known ports.

A problem with the socket API is that it does not shield the programmer from the byte order of this port value. For this reason, I used the ntohs and htons functions to properly convert the byte order of this port value.

Listing 1: Socket.h

```
///////////////////////////////////////////////////////////////////////
//
// socket.h: implementation of the Socket class.
// Copyright 2000 by Randy Charles Morin
// You have unlimited ability to distribute and modify this source,
// but this legal notice must remain intact and the Socket class must
// remain within the kbcafe namespace.
//
///////////////////////////////////////////////////////////////////////
#ifndef KBCAFE_SOCKET_H
#define KBCAFE_SOCKET_H
#include <iostream>
#include <exception>
#include "winsock.h"

namespace
{
	class WSAInit
	{
```

```
	public:
		WSAInit()
		{
			WORD w = MAKEWORD(1,1);
			WSADATA wsadata;
			::WSAStartup(w, &wsadata);
		};
		~WSAInit()
		{
			::WSACleanup();
		};
	} instance;
}

namespace kbcafe
{
	class SocketException : public std::exception
	{
		std::string m_str;
	public:
		SocketException()
			:m_str("Socket exception")
		{}

		SocketException(const std::string & str)
			:m_str("Socket exception:"+str)
		{}

		virtual const char * what() const throw()
		{
			return m_str.c_str();
		}
	};

	class Socket
	{
	public:
		SOCKET m_socket;
		Socket();
		virtual ~Socket();
		void Connect(const std::string & strAddress, const std::string &
		protocol);
		void Connect(const std::string & strAddress, int port);
		virtual std::string Response();
		virtual void Write(const std::string & str);
	};

	inline Socket::Socket()
		:m_socket(NULL)
	{}

	inline Socket::~Socket()
	{
		::closesocket(m_socket);
	};

	inline void Socket::Connect(const std::string & strAddress,
		const std::string & protocol)
	{
		struct servent * sp = ::getservbyname(protocol.c_str(),
			"tcp");
		if (sp == NULL)
		{
			throw SocketException(protocol
				+ " is an unknown TCP service");
		};
		Connect(strAddress, ::ntohs(sp->s_port));
	};

	inline void Socket::Connect(const std::string & strAddress,
		int port)
```

```
        {
                hostent * host;
                in_addr inaddr;
                inaddr.s_addr = ::inet_addr(strAddress.c_str());
                if (inaddr.s_addr == INADDR_NONE)
                {
                        host = ::gethostbyname(strAddress.c_str());
                }
                else
                {
                        host = ::gethostbyaddr((const char *)&inaddr, sizeof(inaddr),
                        AF_INET);
                }
                if (host == NULL)
                {
                        throw SocketException("invalid SMTP server");
                }
                m_socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
                if (m_socket == INVALID_SOCKET)
                {
                        throw SocketException("socket invalid");
                }
                sockaddr_in sa;
                sa.sin_family = AF_INET;
                sa.sin_port = ::htons(port);
                sa.sin_addr.s_addr = *((u_long*)host->h_addr_list[0]);
                if (::connect(m_socket, (sockaddr *)&sa, sizeof(sa)) < 0)
                {
                        throw SocketException("connection to host failed");
                };
        };

        inline std::string Socket::Response()
        {
                int roundtrips = 0;
                std::string response;
                while(true)
                {
                        char buffer[1];
                        int n = ::recv(m_socket, buffer, 1, 0);
                        if (n == -1)
                        {
                                throw SocketException("socket read failed");
                        };
                        response += std::string(buffer, n);
                        if (response.find("\n") != response.npos)
                        {
#ifdef _DEBUG
                                std::cout << response << std::endl;
#endif
                                return response;
                        }
                }
        }

        inline void Socket::Write(const std::string & str)
        {
#ifdef _DEBUG
                std::cout << str << std::endl;
#endif
                int n = str.length();
                const char * s = str.c_str();
                while (n)
                {
                        int i = ::send(m_socket, s, n, 0);
                        if (n <= 0)
                        {
                                throw SocketException("socket write failed");
                        }
                        n -= i;
                        s += i;
```

```
		}
	};
};
#endif
```

The Smtp class is now much more simple as the Socket specific code has now been moved to our Socket class. One improvement I've done with the class is to establish a TCP/IP connection in the constructor, rather than the Send method. This allows us to re-use the SMTP connection and send more than one message with each connection.

Listing 2: Smtp.h

```
//////////////////////////////////////////////////////////////////////
//
// smtp.h: implementation of the Smtp class.
// Copyright 2000 by Randy Charles Morin
// You have unlimited ability to distribute and modify this source,
// but this legal notice must remain intact and the Smtp class must
// remain within the kbcafe namespace.
//
//////////////////////////////////////////////////////////////////////
#ifndef KBCAFE_SMTP_H
#define KBCAFE_SMTP_H

#include <string>
#include <sstream>
#include "socket.h"

namespace kbcafe
{
	class Smtp : public Socket
	{
	public:
		Smtp(const std::string & strServer);
		virtual ~Smtp();
		void Send();
		std::string m_strContent;
		std::string m_strSender;
		std::string m_strRecipient;
		std::string m_strSubject;
	};

	inline Smtp::Smtp(const std::string & strServer)
	{
		Connect(strServer, "smtp");
		std::string response = Response();
		if (response.find("220") == response.npos)
		{
			throw SocketException(response);
		}
		{
			struct sockaddr_in local;
			int n = sizeof(local);
			::getsockname(m_socket, (struct sockaddr *)&local,
				&n);
			struct hostent * h = ::gethostbyaddr((char*)&local.sin_addr,
			sizeof(local.sin_addr), AF_INET);
			std::stringstream ss;
			ss << "HELO "
				<< (char *)(h ? h->h_name : ::inet_ntoa(
					local.sin_addr )) << "\r\n";
			Write(ss.str());
		}
		response = Response();
		if (response.find("250") == response.npos)
		{
			throw SocketException(response);
		}
	}
```

```
        inline Smtp::~Smtp()
        {
                {
                        std::stringstream ss;
                        ss << "QUIT\r\n";
                        Write(ss.str());
                }
                std::string response = Response();
                if (response.find("221") == response.npos)
                {
                        throw SocketException(response);
                }
        };

        inline void Smtp::Send()
        {
                {
                        std::stringstream ss;
                        ss << "MAIL FROM:<" << m_strSender << ">\r\n";
                        Write(ss.str());
                }
                std::string response = Response();
                if (response.find("250") == response.npos)
                {
                        throw SocketException(response);
                }
                {
                        std::stringstream ss;
                        ss << "RCPT TO:<" << m_strRecipient << ">\r\n";
                        Write(ss.str());
                }
                response = Response();
                if (response.find("250") == response.npos)
                {
                        throw SocketException(response);
                }
                {
                        std::stringstream ss;
                        ss << "DATA\r\n";
                        Write(ss.str());
                }
                response = Response();
                if (response.find("354") == response.npos)
                {
                        throw SocketException(response);
                }
                {
                        std::stringstream ss;
                        ss << "Subject: " << m_strSubject << "\r\n"
                                << "To: " << m_strRecipient << "\r\n"
                                << "From: " << m_strSender << "\r\n"
                                << "\r\n" << m_strContent << "\r\n.\r\n";
                        Write(ss.str());
                }
                response = Response();
                if (response.find("250") == response.npos)
                {
                        throw SocketException(response);
                }
        }
};
#endif
```

Finally, we now have a new base for creating our POP3 class. I will not go over the Socket specific code as I did in last months article, but will refer any readers back to that code if they require this knowledge. There are a dozen or so POP3 command that you can use during a POP3 session, but in this article I will limit my discussion to the six most useful verbs, that is, USER, PASS, LIST, RETR, DELE and QUIT.

The TCP/IP connection and the USER and PASS verbs are implemented in the constructor of our class. The disconnect and QUIT verbs are implemented in the destructor of our class. The LIST, RETR and DELE verbs are implemented as methods in our Pop3 class, List, Retrieve and Delete respectively.

```
class Pop3 : public Socket
{
public:
        Pop3(const std::string & strServer,
        const std::string & strUsername,
        const std::string & strPassword);
        virtual ~Pop3();
        std::vector<PopMessage> List();
        std::string Retrieve(int number);
        void Delete(int number);
};
```

The constructor and connection algorithm for our POP3 class requires three pieces of information, the remove POP3 server name, the username to specify the exact user and the password to authenticate the user. When I establish the POP3 connection, the server will respond with a status message. In the POP3 protocol, there are two status message, +OK for success and +ERR for failures. The connection response must start with the +OK status code for us to continue. Otherwise, we throw an exception.

The next step is to use the USER verb to provide the user identification to the server. One space is between the USER verb and the user's name. If the server responds with success, then we follow with the PASS verb to authenticate the user. Again one space is placed between the PASS verb and the password. Note here that the password is sent as clear text across the Internet. Not very secure, but simplicity has its price. Again the server must respond with a success status line.

```
inline Pop3::Pop3(const std::string & strServer,
        const std::string & strUsername,
        const std::string & strPassword)
{
        Connect(strServer, "pop3");
        std::string response = Response();
        if (response.find("+OK") == response.npos)
        {
                throw SocketException(response);
        }
        {
                std::stringstream ss;
                ss << "USER " << strUsername << "\r\n";
                Write(ss.str());
        }
        response = Response();
        if (response.find("+OK") == response.npos)
        {
                throw SocketException(response);
        }
        {
                std::stringstream ss;
                ss << "PASS " << strPassword << "\r\n";
                Write(ss.str());
        }
        response = Response();
        if (response.find("+OK") == response.npos)
        {
                throw SocketException(response);
```

```
	}
};
```

The next step is to retrieve a list of messages that reside on the server. This is done with the LIST verb. This verb returns an identifier and size for each message that resides on the server. In order to properly package this information, I return an array of a two long structure back from the List method. This structure contains the identifier and size of the individual messages. I had to move the structure outside of the kbcafe namespace as some legacy compilers have difficulty using templates and namespaces together. Visual C++ 5.0 being the compiler.

```
struct PopMessage
{
	long number;
	long bytes;
	bool operator==(const PopMessage & rhs) const
	{
		if (rhs.number == number)
		{
			return true;
		}
		return false;
	}
	bool operator<(const PopMessage & rhs) const
	{
		if (rhs.number < number)
		{
			return true;
		}
		return false;
	}
};
```

When you send the LIST verb, the response will be initial a status line, followed by one line for each message on the server and finally a termination line. The termination line is easily identifiable. It contains one character, a period.

```
inline std::vector<PopMessage> Pop3::List()
{
	std::vector<PopMessage> retval;
	{
		std::stringstream ss;
		ss << "LIST\r\n";
		Write(ss.str());
	}
	std::string response = Response();
	if (response.find("+OK") == response.npos)
	{
		throw SocketException(response);
	}
	while (true)
	{
		response = Response();
		if (response[0] == '.')
		{
			break;
		}
		PopMessage msg;
		std::stringstream ss;
		ss << response;
		ss >> msg.number;
		ss >> msg.bytes;
		retval.push_back(msg);
	}
	return retval;
};
```

I can now use the RETR verb to retrieve and one message from the server. Using the message identifiers returned from the LIST verb, I send the RETR verb followed by a space and then the message identifier. The response from the server is one status line followed by the complete message. The termination line is again a line with one character, the period.

```
inline std::string Pop3::Retrieve(int number)
{
	{
		std::stringstream ss;
		ss << "RETR " << number << "\r\n";
		Write(ss.str());
	}
	std::string response = Response();
	if (response.find("+OK") == response.npos)
	{
		throw SocketException(response);
	}
	std::string retval;
	while (true)
	{
		response = Response();
		if (response[0] == '.')
		{
			break;
		}
		retval += response;
	}
	return retval;
};
```

Retrieving a message only spills the contents of the message. The message remains on the server. To delete the message from the server you should use the DELE verb. The DELE verb is followed by a space and the message identifier. The response from the server is one status line. Hopefully, +OK.

```
inline void Pop3::Delete(int number)
{
	{
		std::stringstream ss;
		ss << "DELE " << number << "\r\n";
		Write(ss.str());
	}
	std::string response = Response();
	if (response.find("+OK") == response.npos)
	{
		throw SocketException(response);
	}
};
```

Finally, once you've completed your POP3 session you should disconnect from the POP3 server. This is done by sending the QUIT verb. The server will again respond with a status line and you can then call closesocket, called in parent class.

```
inline Pop3::~Pop3()
{
	{
		std::stringstream ss;
		ss << "QUIT\r\n";
		Write(ss.str());
	}
	std::string response = Response();
	if (response.find("+OK") == response.npos)
	{
		throw SocketException(response);
```

```
        }
};
```

The complete POP3 class is presented in the next listing.

Listing 3: Pop3.h

```
//////////////////////////////////////////////////////////////////////
//
// pop3.h: implementation of the Pop3 class.
// Copyright 2000 by Randy Charles Morin
// You have unlimited ability to distribute and modify this source,
// but this legal notice must remain intact and the Pop3 class must
// remain within the kbcafe namespace.
//
//////////////////////////////////////////////////////////////////////
#ifndef KBCAFE_POP3_H
#define KBCAFE_POP3_H

#include <string>
#include <sstream>
#include <vector>
#include "socket.h"

struct PopMessage
{
        long number;
        long bytes;
        bool operator==(const PopMessage & rhs) const
        {
                if (rhs.number == number)
                {
                        return true;
                }
                return false;
        }

        bool operator<(const PopMessage & rhs) const
        {
                if (rhs.number < number)
                {
                        return true;
                }
                return false;
        }
};

namespace kbcafe
{
        class Pop3 : public Socket
        {
        public:
                Pop3(const std::string & strServer,
                const std::string & strUsername,
                const std::string & strPassword);
                virtual ~Pop3();
                std::vector<PopMessage> List();
                std::string Retrieve(int number);
                void Delete(int number);
        };

        inline Pop3::Pop3(const std::string & strServer,
                const std::string & strUsername,
                const std::string & strPassword)
        {
                Connect(strServer, "pop3");
                std::string response = Response();
                if (response.find("+OK") == response.npos)
                {
                        throw SocketException(response);
                }
```

```
	{
		std::stringstream ss;
		ss << "USER " << strUsername << "\r\n";
		Write(ss.str());
	}
	response = Response();
	if (response.find("+OK") == response.npos)
	{
		throw SocketException(response);
	}
	{
		std::stringstream ss;
		ss << "PASS " << strPassword << "\r\n";
		Write(ss.str());
	}
	response = Response();
	if (response.find("+OK") == response.npos)
	{
		throw SocketException(response);
	}
};

inline Pop3::~Pop3()
{
	{
		std::stringstream ss;
		ss << "QUIT\r\n";
		Write(ss.str());
	}
	std::string response = Response();
	if (response.find("+OK") == response.npos)
	{
		throw SocketException(response);
	}
};

inline std::vector<PopMessage> Pop3::List()
{
	std::vector<PopMessage> retval;
	{
		std::stringstream ss;
		ss << "LIST\r\n";
		Write(ss.str());
	}
	std::string response = Response();
	if (response.find("+OK") == response.npos)
	{
		throw SocketException(response);
	}
	while (true)
	{
		response = Response();
		if (response[0] == '.')
		{
			break;
		}
		PopMessage msg;
		std::stringstream ss;
		ss << response;
		ss >> msg.number;
		ss >> msg.bytes;
		retval.push_back(msg);
	}
	return retval;
};

inline std::string Pop3::Retrieve(int number)
{
	{
		std::stringstream ss;
		ss << "RETR " << number << "\r\n";
```

```
                        Write(ss.str());
                }
                std::string response = Response();
                if (response.find("+OK") == response.npos)
                {
                        throw SocketException(response);
                }
                std::string retval;
                while (true)
                {
                        response = Response();
                        if (response[0] == '.')
                        {
                                break;
                        }
                        retval += response;
                }
                return retval;
        };

        inline void Pop3::Delete(int number)
        {
                {
                        std::stringstream ss;
                        ss << "DELE " << number << "\r\n";
                        Write(ss.str());
                }
                std::string response = Response();
                if (response.find("+OK") == response.npos)
                {
                        throw SocketException(response);
                }
        };
};
#endif
```

I tested this class with the following code.

```
void recv(const std::string & server, const std::string & user,
        const std::string & password)
{
        try
        {
                kbcafe::Pop3 pop(server, user, password);
                std::vector<PopMessage> list = pop.List();
                std::vector<PopMessage>::iterator i = list.begin();
                for (;i != list.end(); i++)
                {
                        std::cout << "Message Number = " << i->number
                                << " Byte = " << i->bytes << std::endl;
                        std::string msg = pop.Retrieve(i->number);
                        std::cout << msg << std::endl;
                        pop.Delete(i->number);
                }
        }
        catch(kbcafe::SocketException & e)
        {
                std::cerr << e.what();
        }
        catch(...)
        {
                std::cerr << "Unhandled exception";
        }
        return;
}
```

The next article in this series, I will introduce you to the NNTP protocol. This is the protocol that has been used for years to communicate with USENET newsgroups.

For a more complete understanding of POP3, I suggest you consult the RFCs (Requests For Comments). RFCs are Internet standards that have been adopted by the IETF (Internet Engineering Task Force). The RFC for POP3 is RFC 1939 and can be found at the following URL

http://www.ietf.org/rfc/rfc1939.txt?number=1939

I found it very helpful to read through the scenarios provided in the RFC. They give a better indication of the complete functionality provided for by this mail protocol.