HowTo SMTP

by Randy Charles Morin

Some of the simplest, yet very rich communication protocols were born on the Internet. This is the first article in a series where I will write on these simple communication protocols. This article will focus on the SMTP (Simple Mail Transfer Protocol) protocol. This protocol is the most often used protocol for sending email over the Internet.

SMTP is nearly always implemented over TCP/IP (Transmission Control Protocol/Internet Protocol), but can be implemented over any type of transport protocol. However, for simplicity sake, the examples in this article will focus on using socket connections and the TCP/IP transport. That's enough about the transport and lower level details of implementing SMTP. If you want to find out more about TCP/IP and other transports, then I suggest you dive into a book on network communications.

Let's begin by defining the public interface to our class. We need five member variables to define the five parameters we need to send out one email. They are the address of the SMTP server (m_strServer) we are sending the messages through; the recipient's Internet email address (m_strRecipient), the sender's Internet email address (m_strSender), the subject line (m_strSubject) and the message body (m_strContent). Finally, we need one method (Send) to invoke the process of sending the email.

Listing 1: Declaration

```
//////////////////////////////////////////////////////////////////////
//
// smtp.h: interface for 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 <exception>
#include <string>
namespace kbcafe
{
	class SmtpException : public std::exception
	{
		std::string m_str;
	public:
		SmtpException()
			:m_str("SMTP exception")
		{}

		SmtpException(const std::string & str)
			:m_str("SMTP exception"+str)
		{}

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

	class Smtp
	{
	public:
		void Send();
```

```
                std::string m_strContent;
                std::string m_strSender;
                std::string m_strRecipient;
                std::string m_strServer;
                std::string m_strSubject;
                Smtp();
                virtual ~Smtp();
        };
};
#endif
```

I also created an exception class that I will use in the implementation of the Smtp class. As the exception is derived from the STL exception class, you can catch either the SmtpException or the base STL exception. Both use the what method to provide a more descriptive string on the symptoms of the exception.

In our example we will use the winsock library. Although this is not the most compliant version of the socket libraries, it is likely the most used. Once particular that we must get out of the way is the initialization and cleanup of the socket library. This is done by calling WSAStartup before any calls to the socket library and calling WSACleanup when you are finished. This can be encapsulated in one class.

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

Now we have our initialization and cleanup code and the only code remaining is the code behind our Send method, that is, call the logic necessary to send one email.

Let's start by asking the winsock library for the well-known-port for the smtp service. I already know the answer, SMTP is almost always over port 25. The getservbyname will always return port 25 for SMTP, but I suggest you use this function rather than relying on the magic number. I do this primarily in respect of the legacy code that I've accumulated over the years.

```
struct servent * sp = ::getservbyname("smtp", "tcp");
if (sp == NULL)
{
        throw SmtpException("SMTP is an unknown TCP service");
};
```

The next step is to translate the SMTP server address into an IP address. This is done by using the inet_addr function to determine if the address is a URL or if it is already in IP form. If the function returns INADDR_NONE, then you can assume that it is a URL and not in IP form. The functions gethostbyname and gethostbyaddr are then used to translate the URL or IP address into a hostent structure. We will require this hostent structure later when we connect to the SMTP server.

```
hostent * host;
in_addr inaddr;
```

```
inaddr.s_addr = ::inet_addr(m_strServer.c_str());
if (inaddr.s_addr == INADDR_NONE)
{
	host = ::gethostbyname(m_strServer.c_str());
}
else
{
	host = ::gethostbyaddr((const char *)&inaddr, sizeof(inaddr),
		AF_INET);
}
if (host == NULL)
{
	throw SmtpException("invalid SMTP server");
}
```

Now that we have the hostent structure, we need to open a socket where we will establish a TCP connection to the SMTP host. The socket is opened using the socket function.

```
SOCKET socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (socket == INVALID_SOCKET)
{
	throw SmtpException("socket invalid");
}
```

Part of the cleanup of sockets is to release each socket when they are no longer required. This is done by calling the closesocket function. In order to guarantee our socket is closed, I created a SmartSocket class. If you pass the socket handle to the constructor of the class, then when the class and socket handle go out of scope the socket is automatically closed.

```
class SmartSocket
{
	SOCKET m_socket;
public:
	SmartSocket(SOCKET & socket)
		:m_socket(socket)
	{}

	~SmartSocket()
	{
		::closesocket(socket);
	}
};
SmartSocket smartsocket(socket);
```

Finally, to establish the TCP connection we call the connect function with the socket and hostent structures returned in our previous calls. We now have a socket connection to the SMTP server.

```
sockaddr_in sa;
sa.sin_family = AF_INET;
sa.sin_port = sp->s_port;
sa.sin_addr.s_addr = *((u_long*)host->h_addr_list[0]);
if (::connect(socket, (sockaddr *)&sa, sizeof(sa)) < 0)
{
	throw SmtpException("connection to host failed");
};
```

Now that we have the connection we can begin transmitting bytes to and receiving bytes from the SMTP server. Writing to a socket is very simple. Using the send function, you send a blob of bytes into the socket.

```
void Write(SOCKET & socket, 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(socket, s, n, 0);
                if (n <= 0)
                {
                        throw kbcafe::SmtpException("socket write failed");
                }
                n -= i;
                s += i;
        }
};
```

Reading from a socket is a bit more complex. You have to account for to protocol behaviors. The first behavior is from TCP. When receiving data from a TCP socket connection, you might not get all the data in your first call to the recv function. You may have to recursively call the function until you know all the bytes have been received. The second behavior is from the SMTP protocol. The protocol outlines that all responses returned from the SMTP server are delimited by a linefeed. This means we can setup a while loop that look forever until the at least one linefeed is received from the socket.

```
std::string Response(SOCKET & socket)
{
        int roundtrips = 0;
        std::string response;
        while(true)
        {
                char buffer[1024];
                int n = ::recv(socket, buffer, sizeof(buffer), 0);
                if (n == -1)
                {
                        throw kbcafe::SmtpException("socket read failed");
                };
                response += std::string(buffer, n);
                if (response.find("\n") != response.npos)
                {
#ifdef _DEBUG
                        std::cout << response << std::endl;
#endif
                        return response;
                }
                roundtrips++;
                if (roundtrips > 1000)
                {
                        throw kbcafe::SmtpException("socket read timeout");
                }
        }
}
```

As soon as the socket connection is established with an SMTP server, the SMTP server will send you a response indicating that it is alive and ready. This response will begin with the status code 220. Any other status code is generally out of protocol and can be considered an error.

```
std::string response = Response(socket);
if (response.find("220") == response.npos)
{
        throw SmtpException(response);
}
```

The next step is to identify yourself to the SMTP server. This is done by issuing the HELO SMTP command following by your identification.

```
struct sockaddr_in local;
int n = sizeof(local);
::getsockname(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(socket, ss.str());
```

The SMTP server will respond with a status of 250.

```
response = Response(socket);
if (response.find("250") == response.npos)
{
       throw SmtpException(response);
}
```

Now you can begin sending email. We'll begin by identifying the sender of the email using the MAIL FROM SMTP command.

```
std::stringstream ss;
ss << "MAIL FROM:<" << m_strSender << ">\r\n";
Write(socket, ss.str());
```

The SMTP server will respond with a status of 250.

```
response = Response(socket);
if (response.find("250") == response.npos)
{
       throw SmtpException(response);
}
```

Next we identify the recipient of the email using the RCPT TO SMTP command.

```
std::stringstream ss;
ss << "RCPT TO:<" << m_strRecipient << ">\r\n";
Write(socket, ss.str());
```

The SMTP server will respond with a status of 250.

```
response = Response(socket);
if (response.find("250") == response.npos)
{
       throw SmtpException(response);
}
```

Finally, we issue the DATA SMTP command. This command indicates to the server that we are prepared to send the content of the email.

```
std::stringstream ss;
ss << "DATA\r\n";
Write(socket, ss.str());
```

The SMTP server will respond with a status of 354.

```
response = Response(socket);
if (response.find("354") == response.npos)
{
       throw SmtpException(response);
}
```

The format of the SMTP message is several headers, followed by a blank line, followed by the message content and finally a line that contains one character, a period.

```
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(socket, ss.str());
```

If the SMTP server received everything without problems, it will respond with a status of 250.

```
response = Response(socket);
if (response.find("250") == response.npos)
{
	throw SmtpException(response);
}
```

The last thing we should do is inform the SMTP server that we are finished. We can do this by issuing the QUIT SMTP command.

```
std::stringstream ss;
ss << "QUIT\r\n";
Write(socket, ss.str());
```

The SMTP server will respond with a status of 221.

```
response = Response(socket);
if (response.find("221") == response.npos)
{
	throw SmtpException(response);
}
```

Now our SmartSocket goes out of scope and the socket is closed along with the connection to the SMTP server.

Listing 2: Implementation

```
//////////////////////////////////////////////////////////////////////
//
// smtp.cpp: 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.
//
//////////////////////////////////////////////////////////////////////
#include "smtp.h"
#include "winsock.h"
#include <sstream>
#include <iostream>
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
namespace
{
	std::string Response(SOCKET & socket)
	{
		int roundtrips = 0;
		std::string response;
		while(true)
		{
			char buffer[1024];
			int n = ::recv(socket, buffer, sizeof(buffer), 0);
			if (n == -1)
			{
				throw kbcafe::SmtpException("socket read failed");
			};
			response += std::string(buffer, n);
			if (response.find("\n") != response.npos)
			{
#ifdef _DEBUG
				std::cout << response << std::endl;
#endif
				return response;
			}
```

```
				roundtrips++;
				if (roundtrips > 1000)
				{
					throw kbcafe::SmtpException("socket read timeout");
				}
			}
		}

		void Write(SOCKET & socket, 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(socket, s, n, 0);
				if (n <= 0)
				{
					throw kbcafe::SmtpException("socket write failed");
				}
				n -= i;
				s += i;
			}
		};

		class WSAInit
		{
		public:
			WSAInit()
			{
				WORD w = MAKEWORD(1,1);
				WSADATA wsadata;
				::WSAStartup(w, &wsadata);
			};

			~WSAInit()
			{
				::WSACleanup();
			};
		} instance;

		class SmartSocket
		{
			SOCKET m_socket;
			public:
			SmartSocket(SOCKET & socket)
			:m_socket(socket){}
			~SmartSocket()
			{
				::closesocket(socket);
			}
		};
}

namespace kbcafe
{
		Smtp::Smtp()
		{
		}

		Smtp::~Smtp()
		{
		}

		void Smtp::Send()
		{
			struct servent * sp = ::getservbyname("smtp", "tcp");
			if (sp == NULL)
			{
        throw SmtpException("SMTP is an unknown TCP service");
};
hostent * host;
in_addr inaddr;
inaddr.s_addr = ::inet_addr(m_strServer.c_str());
if (inaddr.s_addr == INADDR_NONE)
{
        host = ::gethostbyname(m_strServer.c_str());
}
else
{
        host = ::gethostbyaddr((const char *)&inaddr, sizeof(inaddr),
                AF_INET);
}
if (host == NULL)
{
        throw SmtpException("invalid SMTP server");
}
SOCKET socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (socket == INVALID_SOCKET)
{
        throw SmtpException("socket invalid");
}
SmartSocket smartsocket(socket);
sockaddr_in sa;
sa.sin_family = AF_INET;
sa.sin_port = sp->s_port;
sa.sin_addr.s_addr = *((u_long*)host->h_addr_list[0]);
if (::connect(socket, (sockaddr *)&sa, sizeof(sa)) < 0)
{
        throw SmtpException("connection to host failed");
};
std::string response = Response(socket);
if (response.find("220") == response.npos)
{
        throw SmtpException(response);
}
{
        struct sockaddr_in local;
        int n = sizeof(local);
        ::getsockname(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(socket, ss.str());
}
response = Response(socket);
if (response.find("250") == response.npos)
{
        throw SmtpException(response);
}
{
        std::stringstream ss;
        ss << "MAIL FROM:<" << m_strSender << ">\r\n";
        Write(socket, ss.str());
}
response = Response(socket);
if (response.find("250") == response.npos)
{
        throw SmtpException(response);
}
{
        std::stringstream ss;
        ss << "RCPT TO:<" << m_strRecipient << ">\r\n";
        Write(socket, ss.str());
}
response = Response(socket);
if (response.find("250") == response.npos)
```

```
		{
			throw SmtpException(response);
		}
		{
			std::stringstream ss;
			ss << "DATA\r\n";
			Write(socket, ss.str());
		}
		response = Response(socket);
		if (response.find("354") == response.npos)
		{
			throw SmtpException(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(socket, ss.str());
		}
		response = Response(socket);
		if (response.find("250") == response.npos)
		{
			throw SmtpException(response);
		}
		{
			std::stringstream ss;
			ss << "QUIT\r\n";
			Write(socket, ss.str());
		}
		response = Response(socket);
		if (response.find("221") == response.npos)
		{
			throw SmtpException(response);
		}
	}
};
```

Now that we have a functional class, we should write a small test program to show how this class would be used. The sample I'm providing is the code I use to send my monthly newsletter. I fudged the SMTP server address, in order to avoid problems with my ISP.

Listing 3: Sample

```
// newsletter.cpp : How I send my newsletter every month
//
#include "smtp.h"

std::string a[] = { "rmorin@kbcafe.com",
	"newsletter@kbcafe.com",
	"" };

int main(int argc, char* argv[])
{
	kbcafe::Smtp smtp;
	smtp.m_strServer = "smtp.kbcafe.com";
	smtp.m_strSender = "newsletter@kbcafe.com";
	smtp.m_strSubject = "KBCafe.COM August Newsletter";
	smtp.m_strContent = "KBCafe.COM <http://www.kbcafe.com> "
		"August 2000 Newsletter\r\n"
		" by Randy Charles Morin\r\n"
		"\r\n"
		"=================================================\r\n"
		"ARTICLE OF THE MONTH = HowTo SMTP\r\n"
		"\r\n"
		"Some of the simplest, yet very rich communication protocols were born "
		"on the\r\n"
		"Internet. This is the first article in a series where I will write on "
		"these\r\n"
```

```
                "simple communication protocols. This article will focus on the SMTP "
                "(Simple\r\n"
                "Mail Transfer Protocol) protocol. This protocol is the most often "
                "used\r\n"
                "protocol for sending email over the Internet.\r\n"
                "MORE OF THE ARCTICLE @\r\n"
                "http://www.kbcafe.com/articles/smtp.html\r\n"
                "\r\n"
                "==================================================\r\n"
                "ARTICLE OF NEXT MONTH = HowTo POP3\r\n"
                "\r\n"
                "A simple to use POP3 class for all occasions. This is the second "
                "article in\r\n"
                "Internet protocol series.\r\n"
                "\r\n"
                "==================================================\r\n"
                "\r\n"
                ;
        for (int i=0;;i++)
        {
                smtp.m_strRecipient = a[i];
                if (a[i].empty())
                {
                        break;
                }
                try
                {
                        smtp.Send();
                }
                catch(...)
                {
                }
        }
        return 0;
}
```

For a more complete understanding of SMTP, 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 SMTP is RFC 821 and can be found at the following URL

http://www.ietf.org/rfc/rfc0821.txt?number=821

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.