

How to build an e-mail sniffer



*Mailbox plugged with Aunt Maude's attachments?
David Jewell helps you write the net's equivalent to 'Caller ID'*

Aunt Maude has just got back from her skiing trip, and thoughtfully sent you 20Mb of high res bitmaps courtesy of her new digital camera. They're plugging up your e-mail – you don't want to spend the time downloading them, but you can't get at the rest of your mail until you have.

In last month's *C++ Workshop*, we introduced you to the delights of WinShoes, a free library of Internet components which works with C++ Builder 4, the version of Borland's development system that we gave away on the *SuperDisc* a couple of months back. Just in case you missed last month's Workshop, you can find WinShoes on the *SuperDisc*.

This time round, we're going to build an e-mail 'sniffer' program which uses WinShoes to interrogate your POP3 server – the machine belonging to your ISP which is responsible for receiving and storing your incoming e-mail.

Whenever it detects that new e-mail is waiting, it displays the available messages and beeps for your attention. At the same time, the program indicates whom the e-mail is from, the subject of each message, and when it was sent.

Perhaps most importantly, our 'sniffer' also shows the total size of each message. Next time Aunt Maude goes snap-happy, you can discreetly delete her pics at the server without downloading them first. Handy, eh?

Even if you're not related to Aunt Maude and her digital camera, there are times when an application like this can get you out of a big hole. We've had experiences where Outlook Express crashes when downloading a particular message, and since the offending e-mail remains on the server after each crash, you're effectively locked out until the message has been manually deleted. This month's program allows you to do just that.

Using the program

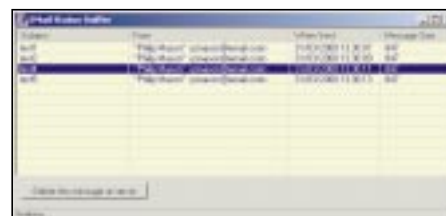
At any point in time, our program will display the list of messages waiting for download on your POP3 server. Each message has an indication of the date/time it was sent, together with the byte size of that message. Obviously, in a real-life situation, you'd expect a bit more variety here, but these are just a set of test messages for developing the software.

Once every minute, the program asks the POP3 server how many messages are waiting. If the reply is greater than expected (in other words, if new mail has arrived), then the program sounds a beep and refreshes the list of displayed message headers. At any time, you can click on a message and then press the 'Delete this message at server' pushbutton at the bottom of the window. This will delete the message from the server without having to go to the bother of downloading it first.



SEE OUR **SUPERDISC**

PATH: PATH: \handson\files\cppworkshp



Here's a screenshot of our e-mail sniffer program working. By combining it with techniques that we've presented in previous *C++ Workshops*, you could convert it into a program which sits in the taskbar tray icon and pops up when new e-mail is available.

Before a program like this can work, it needs to know some details about your e-mail account. The POP3 component we've used needs three things: the address of your POP3 server, the username that you use on that server and the corresponding password for that account. You should have received all this information when you signed up with your current ISP.

Rather than having to tediously type this stuff in every time you run the program, we've arranged things so that the application displays a small 'Account Details' dialog 

CODE CORNER

How to make your Timer tick

The trick to sniffing is to do it often

```
void __fastcall TfrmMain::Timer1Timer(TObject
*Sender)
{
    // Disallow nested calls to this routine
    if (Lock) return; Lock++;

    ++NoStatusChange;
    POP3->KeepAlive(); // keep the server interested...
    --NoStatusChange;

    if (TickCount++ % 6 != 0) return; // only do real
    check once per minute...

    // Time to recheck email status
    try
```

```
{
    // Get number of messages waiting
    int MessageCount = POP3->CheckMessages();
    if (MessageCount > LastMessageCount) {
        MessageBeep (-1);
        StatusBar->SimpleText = IntToStr (MessageCount)
+ "messages were found.";
        RetrieveMessageHeaders (MessageCount);
        LastMessageCount = MessageCount;
    }
}
finally
{
    Lock--;
    StatusBar->SimpleText = "Waiting...";
}
}
```

THE INSIDE STORY

The POP3 Protocol

Get the low down on Post Office Protocol

This month's e-mail 'sniffer' program is based around the TWInShoePop3 component which encapsulates the standard POP3 protocol used to retrieve e-mail over the Internet. If you want to see how this works at the nuts-and-bolts level, you can examine the source code for the POP3 component which is included as part of the WinShoes package.

In order to really appreciate what's going on here, you should look at this source code in conjunction with the documentation that actually relates to POP, otherwise known as the Post Office Protocol. As you probably suspect, POP3 represents the third version of this protocol. On the net, protocol specifications are usually provided as RFC (Request For Comment) documents. Basically, an RFC is a design document which is made widely available in order that as many people as possible will read it and provide (hopefully constructive!) feedback.

You can find the specification for POP3 at www.faqs.org/rfcs/rfc1725.html.

At the same Web site, you'll also find specifications for other common Internet protocols such as SMTP (used to send mail), FTP (for transferring files), HTTP, and others. While reading the spec, you need to bear in mind that TWInShoePop3 doesn't implement the entire POP3 specification because there's no support for the optional UIDL or the APOP commands.

The absence of these commands isn't really relevant to our 'sniffer' application. For example, the APOP command is designed to increase security in cases where a client would otherwise have to send a username and password to the server at frequent intervals. Because we retain a connection to the server for as long as the sniffer program is running, this isn't an issue.

The Account Details form is used to obtain information that's needed for accessing your e-mail server. Note that the user account shown here is entirely imaginary!

box – see the picture above. When the 'sniffer' is first started, it displays this dialog for you to enter your e-mail details.

This information is then written to a small .INI file, account.ini, which lives in the same directory as the program itself. From then on, every time that you start the program, it will display your account details and you only need to click OK to proceed to the main window. If you're in a security-conscious environment, you'll need to adopt a more secure strategy.

Talking POP

Let's roll up our sleeves and take a look at how this code works.

When the application starts running, the FormCreate handler is called in the usual way. It creates an instance of the 'Account Details' form and uses this to retrieve the e-mail account information from the user. We won't look at how the Account form reads/writes the .INI file, because this is all standard stuff – this month we're concentrating on how to use WinShoes.

Once the account information has been retrieved, the server address, username and password information get plugged into the corresponding properties of the POP3 component – and at that point, we're ready to roll!

The program quickly checks to see if the user hit Cancel to leave the Account Details form and – if not – it calls the Connect method to establish a connection with the POP3 server. This doesn't connect you to the Internet – the software assumes that you've already got a dial up connection going.

If there isn't a dial up connection at the time Connect is called, then an exception is raised and the software terminates with an appropriate error message. If everything went okay, the Timer1Timer procedure is called, and the associated timer component is enabled.

Timer component? What timer component? In order to periodically 'sniff' the state of play on the POP3 server, we need the services of a

timer. This calls the Timer1Timer routine once every ten seconds. However, it doesn't interrogate the server every ten seconds – that would be excessive and lead to poor performance.

Instead, it simply calls the POP3 component's KeepAlive method once every ten seconds. This sends a special 'do-nothing' command to the server – its purpose is just to let the server know that we're still alive and wish to keep the connection open. You can see how this works in this month's Code Corner – *TimerTick snippet*.

On every sixth call to the Timer1Timer routine (that is, once every minute) the code calls the CheckMessages method which tells us how many messages are currently waiting at the server. If this number is greater than it was last time, then we know that new messages have arrived. This causes the program to beep and call the RetrieveMessageHeaders method.

As the name suggests, this RetrieveMessageHeaders interrogates the server, retrieving the current state of play and redisplaying it in the associated list-view control. Of course, some folks will feel that interrogating the server once every minute is still excessive –

If you're interested in Internet protocols, you can find documentation galore at the www.faqs.org Web site which has an extensive list of RFC (Request For Comment) documents covering all the important protocols.



you can easily change things by altering the Interval property of the timer control.

There are some interesting subtleties here such as the use of the private Lock variable. By incrementing this variable on entry to the Timer1Timer routine and decrementing it when we exit, it prevents the routine from being invoked by a timer tick while a previous tick is being processed. If that were to happen, the program would get its knickers in a twist!

Similarly, the btnRemoveClick code (called when you hit the Delete button) is written so as to do nothing if a timer tick is in progress.

One interesting wrinkle here is the need to disconnect from the POP3 server once a message has been deleted. This is required by the POP3 protocol; see *POP3 Protocol – The Inside Story*. If you delete a message on an e-mail server, the message doesn't get physically deleted until the connection with the server is terminated. You can think of this as being the e-mail equivalent of a 'commit' command when working with database transactions.

Once the offending e-mail has been deleted, the code sets up the LastMessageCount and TickCount variables so as to ensure that the list-view control is immediately updated. See if you can figure out how this works.

If you want to investigate e-mail access further, you might take a look at the POP3Client demo program which is provided as part of the WinShoes package. It's in the Demos\CBuilder\POP3Client directory and demonstrates not only how to delete messages, but also how to retrieve them – complete with any attachments.

Exciting stuff! **PCP**

Dave Jewell
dave.jewell@futurenet.co.uk

NEXT MONTH

Dave Jewell shows you how to use WinShoes to build your own news reader.