# COSC 465: Computer Networking
# Lab 2: Blather: the new twitter
# Due: 15 February 2012, before next lab

## 1 Overview

In this lab, you'll write a client program to work with the amazing new nano-blogging site, *Blather*. Your client program will need to be able to (1) make a request to post a "blat" to the Blather server, and (2) make a request to the Blather server to download the most recent blats (up to 10). If you think Twitter's 140 characters is awesome, wait until you try Blather's 50 characters!

Blather uses a simple application-layer protocol running on a UDP-based transport, with no security mechanisms to get in the way of your free expression. Best of all, the only Blather server is hosted in the computer science department at Colgate University. You have exclusive access to this amazing new service to make your voice heard.

## 2 Blather protocol details

The Blather server is located at IP address `149.43.80.25`, UDP port 2468. Blather's application protocol uses strings formatted in particular ways to specify requests and responses. Since Blather uses UDP, messages are *not* reliably delivered. In this lab, we'll ignore the issue of reliability.

There are exactly two request types: `GETBLAT` and `PUTBLAT`.

The `GETBLAT` message consists of the 7-character string (in uppercase) `GETBLAT`.

There are two possible response messages to a `GETBLAT` request: `OK` and `NO`. The `NO` response consists of the two characters `NO` followed by 100 characters of explanation text. The `OK` response consists of the 2 characters `OK`, followed by two ASCII digits, which indicate the number of blats to follow. The digits can be `'00'` through `'10'`, so there can be at minimum zero blats to follow, and at maximum 10 blats to follow. Each blat consists of 60 characters: the username (10 characters) and the blat (50 characters). Each field (the username, the blat, and the explanation in case of a `NO` response) is not NULL-terminated; unused characters are filled with spaces.

The `PUTBLAT` message is 67 bytes long and consists of the 7 character string `PUTBLAT`, followed by a 10 character user name, followed by a 50 character blat. You can use the username field to identify yourself (e.g., `@itsme!`), and your message must fit within the last 50 characters of the `PUTBLAT` message. Again, the strings should not be NULL-terminated like a standard C string; any unused characters should be filled with spaces.

There are two possible responses to the `PUTBLAT` request: `OK` and `NO`. The `OK` response consists of the two characters `OK`. The `NO` response consists of the two characters `NO` followed by 100 characters of explanation text.

For any unrecognized request message, the Blather server will respond with `NO` followed by 100 characters of explanation text.

For *full* details of the protocol, see the text file `blather_protocol.txt`, posted on Moodle.

## 3 Blather client requirements

Your Blather client must be able to (1) send a `GETBLAT` request and print the returned blats (or the error message, if one is returned), and (2) send a `PUTBLAT` message with a username and a message to send. You can choose any reasonable way to allow a user to specify which request they'd like to perform. For the `PUTBLAT` request, you should allow the user to easily modify the username and message to be sent.

Your client program should not *block* when it attempts to receive a response from the server. If no response is received to a request within 1 second, your program should print an error message and exit. (Note that this is the behavior of the example `client.cc` code posted on Moodle; it uses the `poll` system call to achieve this behavior.)

One way you might implement your client program is to have it behave differently, depending on the number of command-line arguments: if there is only one argument (i.e., the program name itself), you can just perform a `GETBLAT`. If there are exactly 3 arguments (i.e., the program name, a user name, and a blat), you can perform a `PUTBLAT`. For any other number of arguments, you can print an error message and exit.

Given the above idea, here are some examples of how your program might run. Note that in the example below, the server responds with a spurious error — the server is programmed to generate fake errors randomly. You'll know if that happens because the server will include the text ("fake error") at the end of the error response.

```
# a GETBLAT request
$ ./blather_client
Response len: 64.  Response status: OK
 Got 1 blats
    1 @SYSTEM   : A clear conscience is usually a sign of bad memory


# a PUTBLAT request
$ ./blather_client '@itsme!' 'my first blat - oh boy! #underwhelming'
Response len: 2.  Response status: OK


# another GETBLAT request
$ ./blather_client
Response len: 124.  Response status: OK
 Got 2 blats
    1 @SYSTEM   : A clear conscience is usually a sign of bad memory
    2 @itsme!   : my first blat - oh boy! #underwhelming


# a GETBLAT request that results in a spurious server error
$ ./blather_client
Response len: 102.  Response status: NO
Reason: It's too hard, and alpha particles have decimated my memory. (fake error)
```

# 4 Hints

Feel free to use the UDP client program (`client.cc`) posted on Moodle. It includes code to send a UDP packet to a server, and to receive a response. The main things you'll need to do will be to implement the Blather protocol, described above, to handle the various requests and responses that can occur. There is no need to create classes for this lab (though if you'd like to design your code that way, you're certainly welcome to do so).

Since all Blather protocol messages are strings, you might find the built-in C++ `string` class helpful, as well as the `ostringstream` and class (and possibly the `istringsream` class) (found in the <sstream> header file).

The `ostringstream` class provides a string buffer that behaves like a file-like object. You can use it much like the `std::cout` standard output stream, and send different variables and literal values to it for output. For example, you could use an `ostringstream` to convert an integer to a string, as follows:

```
std::ostringstream ostr;
ostr << 42;
std::string mydigits = ostr.str();
// mydigits now contains the string "42"
```

The full reference for the `ostringstream` class is here: `http://en.cppreference.com/w/cpp/io/basic_ostringstream`.

Some other useful parts of the C++ iostreams library can be used specify the field width for an item to be output on a stream, as well as the specific "fill" characters, if an item's width is less than that specified. For example, say you want to convert the integer 13 to a string, but to have the string be exactly 3 characters long, and the unused leading character (since "13" is only 2 characters long) be a '0'. The following code would do the trick:

```
std::ostringstream ostr;
ostr << std::setw(3) << std::setfill('0') << 13;
std::string mydigits = ostr.str();
// mydigits now contains the string "013"
```

Note that the `setw` and `setfill` manipulators only apply to the *next* item output to the stream. These manipulators (and others) are found in the <iomanip> header, with some documentation available at: `http://en.cppreference.com/w/cpp/io/manip`.