COSC 465: Computer Networking
Spring 2014
Project 2: Reliable message board application with stop-and-wait
Due: 9 February 2014, 11:59:59pm
Note: You are welcome and encouraged to work in pairs or threes if you wish

Overview

The goal for this lab is to improve the reliability of your message board application client by **adding a stop-and-wait protocol to the application**. The application layer data is completely unchanged from the description of Project 1. The only changes in the messages to and from the server are in the application layer header. Thus in your code, almost all the changes should be to the `MessageBoardNetwork` class.

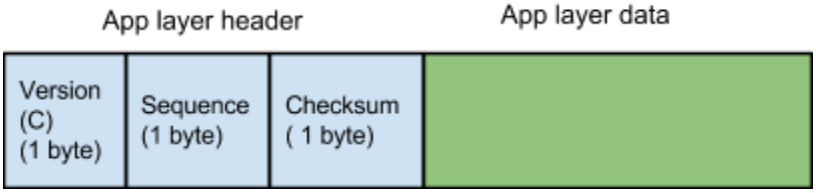
Stop-and-wait

Stop-and-wait is a simple protocol that is used to verify receipt of messages over an unreliable network. The basic idea is for request messages to contain a sequence number (0 or 1), and for reply messages to include the same sequence number as the request, indicating that the request was successfully received. If a reply is not received in a certain amount of time, the request is resent. A limited number of retries are used; if no reply is received after the maximum number of attempts, the request is aborted.

We will use a form of the stop-and-wait protocol that closely mirrors the one described in the textbook, as discussed below.

Application layer header format

In this version of the application, the application layer header is composed of three bytes: a version identifier ('C'), a sequence identifier, and a checksum:



- The version byte is the single character 'C'.
- The checksum should be computed as described in lab 1
- The sequence will be one of two characters: '0' (character zero) or '1' (character 1)

Stop-and-wait operation

As noted above, there are only two sequence numbers (encoded as characters '0' and '1'). The reply to either a GET or POST request should be treated as an ACKnowledgment from the server of having received the corresponding GET or POST request. For example, if a GET message is sent with sequence '0', the reply should also have sequence '0'.

Some other details regarding how sequence numbers, timeouts, and retransmissions should be handled:

- The sequence should be initialized at '0'. That is, when your message board client application starts up, it should start using sequence '0'. If a reply message is correctly received for a given sequence number, the sequence number should alternate. For example, if you receive a reply for sequence '0', the next message you send should have sequence '1'.
- You should have two configurable parameters: a timeout value (in seconds), and a maximum number of retransmissions.
- When a request is sent to the server, the client should wait for the timeout number of seconds (for example, 0.1 seconds). If no reply is received, the client should resend the request. The client should keep doing this (waiting for a timeout, then retransmitting the request) until it has sent the maximum number of retransmissions. At that point, the client should give up and display an error message in the view.
- Note that if the client attempted a GET request with sequence '0', and the request ultimately fails (i.e., it sent the max number of retransmissions but didn't get a response) the sequence number of the next GET request should still be '0'. That is, the sequence number shouldn't be updated except when you get a "positive" reply/ACK.
- There are no Negative Acknowledgments (NAKs) in this protocol, so if there is anything wrong with the header (version, sequence, or checksum) you will not get a reply message. You will only get a reply if all header items are correct.
- It is quite possible that when you try a POST request that the same message gets added to the server more than once due to a dropped ACK and a subsequent retransmission. This is expected behavior.
- One nice way to add the two new parameters to your program (the timeout and max retries) is to add the capability to include two new command line parameters, then pass those parameters to the controller class (and then on to the network/model class). You can add the following two lines toward the bottom of your code (just before the call to `args = parser.parse_args()`):

```
parser.add_argument("--retries", dest='retries', type=int, default=3,
                    help='Set the number of retransmissions in case of a timeout')
parser.add_argument("--timeout", dest='timeout', type=float, default=0.1,
                    help='Set the RTO value')
```

  You can then add two parameters to the controller class constructor (args.retries and args.timeout), then pass these values on to the constructor of MessageBoardNetwork (which is really where they're needed).

  To start your program and give new values for these parameters, you could type at the terminal something like:

```
python messenger465.py --timeout 0.05 --retries 5
```

Testing

You can use the MBserver465.pyc file from project 1 for testing on your own computer, but you'll need to start it up differently. You should start the server as follows:

```
$ python MBserver465.pyc --version C
```

If you'd like to simulate dropping different messages and mangling bits in messages, you can type `python MBserver465.pyc --help` to see options for randomly dropping messages and/or mangling bits.

Additionally, there are two version C servers running on the host 149.43.80.25: one that behaves nicely, and one that does not (i.e., it drops and mangles messages).

The nice one is located on port 3111 and the unnice one is on port 3112.