Yuval Shavitt

Programming Assignment 2: Distributed Bellman-Ford

Due: April 28, 2011.

1 Introduction

In this assignment you will implement a simple version of the Distributed Bellman-Ford algorithm. The algorithm will operate as a set of user-level processes. Each process p gets, as input, the set of its neighboring nodes, and for each neighbor p', the cost of the link (p, p'). The task is, for each process, to compute its root, its next link in its shortest path to the root, and the length of that shortest path.

The processes will communicate by exchanging UDP messages. Specifically, each process will listen to a UDP socket for incoming messages (the number of that port is known to its neighbors). Similarly to BPDU in the Spanning-Tree Protocol of IEEE 802.1, update messages sent by a node contain the following information:

- myRoot - Smallest ID known to the process.
- myCost - cost of best known path to myRoot.
- myID - Identification number of the sending process.
- myRootTime - How long does myRoot have until expiration, in milliseconds.

These messages are sent to all neighbors. A message is sent if and only if one of the following holds:

- The process view (namely, the value of myRoot, myCost or myRootTime) has changed.
- Every HELLO-TIMEOUT (a predefined constant) period of time.

Your program should work even if processes crash and recover. A crash is detected if no message is received from a neighbor for MAX-TIME. The HELLO timer, counting from HELLO-TIMEOUT down to zero, is reset whenever an update message is sent from current process to its neighbor nodes(for debugging purposes, 2 seconds may be a good value for HELLO-TIMEOUT).

To allow recovery in case the root crashes, processes must also keep track of the expiration time as implied by myRootTime. Whenever a message is sent, it is sent with the myRootTime value adjusted to account for the time since it was received (for example, if a message with myRootTime=100 is received at time 3, then a message sent at time 12 will have myRootTime=100-(12-3)=91). The expiration time may increase only when a fresher message (i.e., with a larger myRootTime value) arrives for with the same ID of myRoot. Whenever myRootTime expires (reaches 0), the process sets myRoot to it own ID, and myCost to 0. A process whose myRoot is its own ID has expiration time infinity, and in the messages it sends out, it sets myRootTime=MAX-TIME, which is another predefined constant called (usually 3 times HELLO-TIMEOUT).

2 User Interface

You will write code for a process that receives command-line arguments as follows:

> `bfproc` *`procid localport lifetime hellotimeout maxtime ipaddress1 port1 cost1 [ipaddress2 port2 cost2...]`*

Where:

- `bfproc` is the name of the local process (your executable).
- *procid* is the ID of the current process.
- *localport* is the number of the local port that the process should listen to. Use numbers higher than 5000.
- *lifetime* is the time, in seconds, the process should be up before it gracefully terminates.
- *hellotimeout* is the maximal time, in seconds, the process should wait between HELLO messages.
- *maxtime* is the maximal time, in seconds, before a node is considered crashed.
- The rest of the line consists of triples (at least one) indicating incident links. In each triple we have:
  - `ipaddress` is the IP address of the node at the other endpoint, given in *dotted decimal notation.*
  - `port` is the port number to which the remote process is listening.
  - `cost` is a natural (i.e. a positive integer) number indicating the cost of the link.

We will consider only bidirectional links, namely if node A appears in the neighbor list of node B, then node B appears in the neighbor list of node A.

When the process runs, it prints out (to the standard output) the time, and every event that happens, including timeout-triggered events. For example, a typical output may look like

```
>bfproc 100 6734 10 3 9 132.66.32.10 6734 3  132.66.50.50 6734 1
time=120.0 Root:100 parent: NULL        distance = 0 Update Root
time=120.1 Sent update to neighbors
time=123.1 Sent update to neighbors
time=125.3 Root: 95 parent:132.66.32.10 distance = 3 Update Root
time=125.4 Sent update to neighbors
time=125.6 Root: 95 parent:132.66.50.50 distance = 2 Update Parent
time=128.3 Received update from 132.66.32.10, No change
time=128.4 Sent update to neighbors
time=128.6 Received update from 132.66.50.50, No change, updated timestamp
time=128.7 Sent update to neighbors
time=130.0 Lifetime expired. Shutting down.
```

Obviously, for debugging purposes more printouts may be helpful.

Notes:

1. Time is obtained using *GetTickCount*. Take a look at the manual for explanation and examples. This can be used to calculate timer's status.
2. The initial state is printed out. Also note that special values are printed ("NULL").
3. In the third line, an update is sent due to a HELLO timer expiring. However, in rows 2 and 5, updates are sent to all neighbors due to topology change.

Tips.

- It is convenient to work with a thread for each neighbor is possible (but you you need to take care when accessing the shared data structures and following timeout events). If you choose a single-threaded process, you need to design a timer mechanism that allows for multiple timeouts: in that case you will be using only one timer (implicit in the *select* system call) when you block, which means that you have to manage a queue of timeout values that should occur in the future (e.g., the HELLO, MAX-TIME and lifetime timeouts). When a call to *select* returns, you should read the current time using *GetTickCount*, and when you call it again, deduct the elapsed time from the timeout value. Note that some events should also be canceled from the queue.
- As the application uses UDP messages, it is recommended to use *sendto* and *recvfrom*. Note that myID is not the same as the IP address of the process, and therefore myID must be sent in every message.
- Test your program with non-trivial topologies. Try to run the program with different initial distance values and with loops.
- Note that you can run multiple processes on the same machine: just use different port numbers.
- A process should be terminated when its lifetime expires. Make sure to properly close all sockets and release other used resources.
- An example of *GetTickCount* usage:

```
DWORD t_start, t_end, t_elapsed;
t_start = GetTickCount();
DoSomething();
t_end = GetTickCount();
t_elapsed = t_end - t_start;
```

For examples of using *select* (including timeouts), see Beej [2]).

- There is a known bug in windows XP that causes recvfrom to return WSAECONNRESET error message for UDP packets. The problem has a simple fix described in http://support.microsoft.com/?kbid=263823.

The exercise will be graded first and foremost by implementing functionality, which will be tested under simple and extreme scenarios. Efficiency, both of communication and coding, is also part of the grade. Good coding practices are required as well. Points will be taken off for lack of documentation.

3 What to submit

Submission will be done by email to `networks@eng.tau.ac.il`. You need to submit an attachment containing a zip archive that contains the following files, *and them only*:

- README: contains the full documentation, including names and ID numbers of the team members. We will read this file.
- Visual studio project: used to build your application under the name bfproc.
- All project .h and .c (or .cpp) source code files.

The documentation should described whatever is unusual about your implementation (such as additional features). Include brief explanation of your algorithm and data structures in the code. Describe the sent packets structure and the output you print on screen.

4 References

1. MSDN documentation on Windows sockets (`http://msdn.microsoft.com/library/en-us/winsock/winsock/winsock_reference.asp`).
2. Beej's Guide to Network Programming - Explaining select() and an example of usage of select() with a timeout. (`http://beej.us/guide/bgnet/output/html/multipage/advanced.html#select`).
3. Spencer's Socket Site is a nice collection of socket resources, including tutorials and sample code for Visual C. (`http://www.lowtek.com/sockets/`)
4. *Unix Network Programming vol. 1*, R.W. by Stevens. It's written for BSD Unix, but it's still one of the best sources on socket programming.