Readme for DSP EXE 1

Tyrväinen, Jenny
013483708

January 27, 2016

1 Code and design

My implementation of Lamport Clocks consists of following Python (v. 2.7) files:

- lamport.py: The main class, containing main program logic. It checks, that the parameters are sufficient, reads the host file and parses it, then sleeps to wait, that all the nodes are "online" before executing. The main event loop iterates over 100 events (local, sending, receiving), before finishing.
- lnode.py: The class LNode implements a node's functionality. The node has functions for local events and sending. It creates an instance of a LListener class, that handles receiving messages.
- llistener.py: The LListener class has it's own thread for handling incoming messages.
- lclock.py: The LListener and LNode share a Lamport Clock, implemented in class LClock. It has a lock to prevent both threads trying to access the same resource (clock value) at the same time.
- eventcounter.py: The EventCounter keeps track of events. Because receiving messages happens on it's own thread, the EventCounter is shared as well as the LClock. Therefore it also implements threading.Lock.

The host list is saved to a dictionary to handle possibly missing numbers and unordered nodes. The indexing is assumed to begin from 0 (and documented here). After reading hosts and separating the running host (this host) from the dictionary, the program waits for 10 seconds.

The Lamport Clock algorithm I have implemented as follows:

- local event: increment local clock by a random value from 1 to 5
- send event: increment local clock by one, send msg (containing sender ID and clock value)

- receive event: increment local clock by one, compare this and received clock value, take maximum of these and add one

[1]

Using the threading.Lock probably wasn't the best choice to handle concurrency, but as it wasn't necessary to even have a separate thread for listening, I consider my solution quite adequate.

2 Running the code

I ran my code in the Ukko cluster as follows:

I started three terminals, navigated to the directory humaloja/ in all. I used as test host file.

```
3   ukko035.hpc.cs.helsinki.fi 1234
17  ukko036.hpc.cs.helsinki.fi 1235
7   ukko037.hpc.cs.helsinki.fi 1236
4   ukko040.hpc.cs.helsinki.fi 1237
100 ukko041.hpc.cs.helsinki.fi 1238
29  ukko042.hpc.cs.helsinki.fi 1239
1   ukko043.hpc.cs.helsinki.fi 1334
2   ukko044.hpc.cs.helsinki.fi 1434
5   ukko045.hpc.cs.helsinki.fi 1534
28  ukko046.hpc.cs.helsinki.fi 1634
11  ukko050.hpc.cs.helsinki.fi 1734
10  ukko051.hpc.cs.helsinki.fi 1834
15  ukko052.hpc.cs.helsinki.fi 1934
8   ukko053.hpc.cs.helsinki.fi 1134
```

- 1st terminal: humaloja@ukko042:/cs/work/scratch/humaloja/humaloja/ python lamport.py hosts.txt 5
- 2nd terminal: humaloja@ukko040:/cs/work/scratch/humaloja/humaloja/ python lamport.py hosts.txt 3
- 3rd terminal: humaloja@ukko041:/cs/work/scratch/humaloja/humaloja/ python lamport.py hosts.txt 4

Unfortunately, I didn't have time to do a starting script, although I understand it wasn't compulsory.

The program waits for socket timeouts at the end for 10 seconds, so it isnt't "freezed".

References

[1] Paul Krzyzanowski. Cs 417 documents, 2016.