Patrick Cudahy

3/16/2018

CMPS290S

pyHLC

pyHLC is a Python 3.6 implementation of the Hybrid Logical Clocks (HLC) described in *Logical Physical Clocks*[1] by Sandeep Kukarni, Murat Demirbas, Deepak Madeppa, Bharadwaj Avva, and Marcelo Leone at the University of Buffalo, SUNY. Closely following the algorithm described in the paper, pyHLC uses ZeroMQ[2] and TCP/IP connections to communicate and track time across multiple nodes and systems. The clock system was then tested with a set of separate computers sending high-frequency messages and various update intervals for the NTP physical time server. If pyHLC continues to be a viable time tracking and messaging system, there are currently plans to implement more features and publish it as an open source Python package.

The core of pyHLC is the HLC algorithm described in *Logical Physical Clocks*. The paper describes an alternative to the TrueTime API used by Google's Spanner, avoiding the incredibly expensive and expansive hardware requirements that are infeasible for any smaller company or team and the increasing waits associating with correcting time uncertainty while also retaining serializability and timestamp consistency. To accomplish this, HLC introduces a new clock system that is a hybrid (hence the name) of physical time, as is used by the TrueTime API, and vector clocks, the version of logical clocks commonly in use today. An NTP server or pool of servers is used to update physical time across the distributed system, but since physical time

[1] https://www.cse.buffalo.edu//~demirbas/publications/hlc.pdf
[2] zeromq.org

quickly becomes uncertain without some sort of physical clock being present at every node and constantly synced, vector clocks serialize actions taken by nodes in between updates from the physical clock(s). The vector clocks act essentially as a counter, incremented and shared with the other nodes every time a node does some sort of action and reset when there is a physical time update from an NTP node. The algorithm that handles this can be seen in figure 1.

Initially $l.j := 0; c.j := 0$

Send or local event
$l'.j := l.j;$
$l.j := max(l'.j, pt.j);$
If $(l.j = l'.j)$ then $c.j := c.j + 1$
Else $c.j := 0;$
Timestamp with $l.j, c.j$

Receive event of message m
$l'.j := l.j;$
$l.j := max(l'.j, l.m, pt.j);$
If $(l.j = l'j = l.m)$
then $c.j := max(c.j, c.m) + 1$
Elseif $(l.j = l'.j)$ then $c.j := c.j + 1$
Elseif $(l.j = l.m)$ then $c.j := c.m + 1$
Else $c.j := 0$
Timestamp with $l.j, c.j$

Fig. 4. HLC algorithm for node j

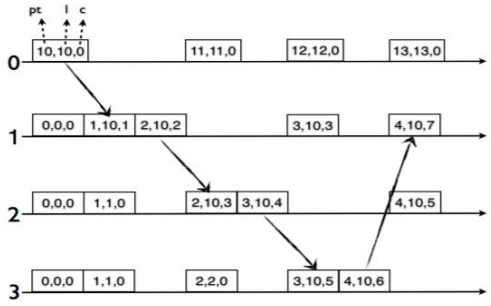

Fig. 5. Fixing the Counterexample in Figure 3 with Algorithm in Figure 4

Figure 1: HLC algorithm as shown in *Logical Physical Clocks*

The main goal of the project was implementing an HLC messaging system in Python. Messaging was handled with ZeroMQ, which provides a very low latency method for communicating between nodes. pyHLC solely uses ZeroMQ's publish/subscribe features, functioning similar to an RSS feed where publishers push messages to a queue and subscribes read them in the order in which they were published. Subscribers can watch as many feeds as they need and add queues as needed. For my implementation, ZeroMQ is used to send NTP timestamps with the last three digits replaced by the current c value. NTP are solely publishers, sending out timestamps taken from the NTP pool at their designated interval. All other nodes

have two threads, one for publishing messages with their current timestamp and c value, as well as incrementing the c value for sent messages, and one for receiving messages, thereby updating their timestamp from NTP nodes and c values from other regular nodes. A running system can be seen in figure 2, showing one NTP node (upper left) and three regular nodes sending messages to one another. The demonstration has print statements to show what messages are being sent and seen and when each thread is acquiring the flag to publish or read a message.

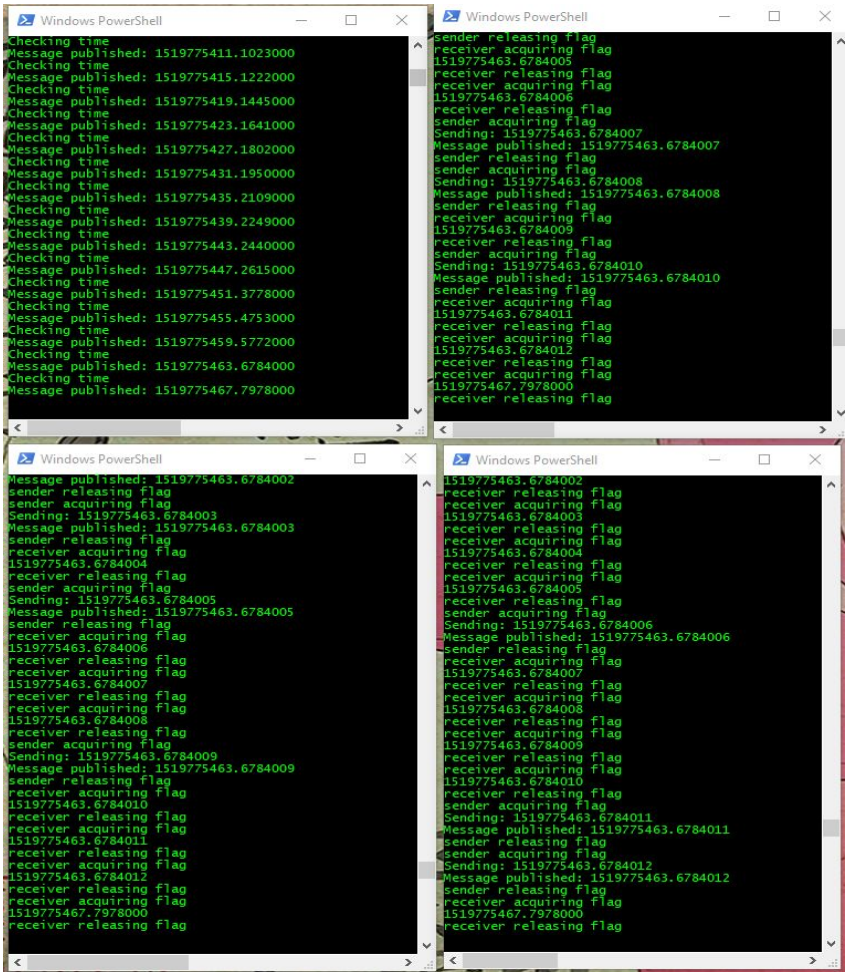

Figure 2: pyHLC Demonstration

The final step of the project was testing pyHLC. It turned out to also be the hardest part of the project since pyHLC is really more of a module to be used in larger programs rather than a full program unto itself. In the end, I chose to test it setting up five regular across three

computers to declare that they had taken some action every ten milliseconds to one second and one NTP to update physical time at different intervals. The NTP node was set to five different intervals, half second, one, two, three, and four seconds, to make sure c values did not increase exponentially as the physical time update interval grew, which would cause the system to fail if enough nodes were involved. Each interval was tested for about twenty minutes. Since every node recorded the same max value seen, I feel that it is safe to assume the nodes were not dropping messages as c values grew[3]. Sometimes a node would have a lower max c value seen by one or two, but this is to be expected as receiving a physical time update would cause c value updates with lower NTP timestamps (e.g. those sent at the same time the NTP node publishes a timestamp). As figure 3 shows, increasing the NTP time publish interval led to a rather slow increase in max and average c value seen. Most importantly, the average c value seen grows very slowly.

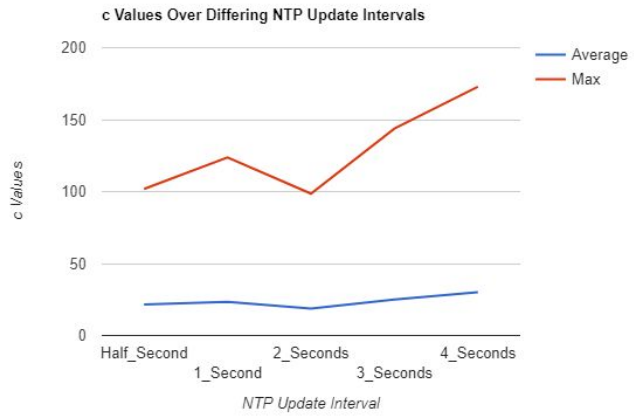

Figure 3: Max and average c values seen

Though this may be the end of the project for the purposes of CMPS-290S, I plan to continue working on pyHLC as I feel it is something that could be useful for others to use with

[3] The last node to send a message would show a max value seen as one lower than others, since it did not receive it own message to log it

their distributed system applications. The first and most obvious step is to remove print statements used to visually collect data for the project and detect problems. The next glaring issue is the requirement of manually adding TCP/IP addresses of all nodes to a file called socket_list.txt for every regular node. I think it should be trivial for the NTP node to also out new nodes' TCP/IP addresses to all subscribed nodes since all nodes must be subscribed to the NTP node and then each node would only need to be told the NTP node's TCP/IP address. The next task, which should be fairly simple given the structure of ZeroMQ, is to added more detail to the regular node's messages indicating what action is being done so as to allow for locking between nodes for PAXOS-like applications. Once this is done, I hope to make it available as an open-source and pip-installable package for Python 3.