Jordan Hughes
CS 271

Homework Assignment 1

For this clock synchronization assignment, I opted to use Java as the programming language on my MacBook Pro. In a situation where only one time server is available to query, one could query the time server and receive the time, t_{serv}. Using the calculated round trip time, rtt, t the local system could then set the new local time to $\frac{t_{serv}+rtt}{2}$. This is Cristian's algorithm for clock synchronization, and it relies on the assumption that the send and receive times are equal for a t_{serv} request.

In this assignment however, there are multiple servers and it was necessary to incorporate all times into the calculation for a new time. One way to do this, and the way I chose, was to use Marzullo's algorithm. This algorithm relies on multiple ranges of possible times. To get a single range, I used $[t_{serv}, t_{serv} + rtt]$. Cristian's algorithm uses the midpoint of that range. In Marzullo's algorithm, we take all begin and endpoints of the range, sort them chronologically, and figure out where there is overlap. The algorithm will find a new "best interval" using the ranges that have the greatest occurrence of overlap. We can then use this range of $[best_{begin}, best_{end}]$ to find a midpoint and use that midpoint for the new local clock time.

There are some special cases to consider for Marzullo's algorithm. In this assignment, five time servers were used. If there happened to be a case where we had 2 servers that overlapped and then another 2 servers that overlapped, we would have a tie for the greatest number of overlapping intervals. Marzullo's algorithm does not specify what to do in the case of the tie. I decided to leave the algorithm as is meaning that it will simply take the first of those intervals chronologically and use that to find a new local clock time. However, one could solve this in other ways such as choosing the smaller of the two intervals or perhaps even choosing the range with the midpoint that is closer to current local time.

Another special case to consider is a situation where there is no overlap between any two intervals. Note that this is actually just another variation of a tie for the greatest number of overlapping intervals, but all intervals are tied at one. In this case, which did come up in the assignment, I chose to handle it by calculating the midpoint of each interval range (i.e. Cristian's algorithm), and then simply average those midpoints. There are certainly other ways to handle this scenario such as incorporating the local clock into the mean calculation or simply choosing the midpoint closest to the local clock. None of these are particularly optimal solutions, but there are probably larger issues at hand if there are no overlapping intervals.

One thing to consider is how often a resynchronization such as this would have to be done. According to my ntp.drift file on my Mac, my clock drifts at a rate of 27.872 PPM or 27.872 microseconds per second. This gives a drift of 1.67232 milliseconds per minute. To keep clock drift under 1 millisecond, we would have to resync about every 35.87 seconds. This would be a good enough approach to synchronize a local clock. If we were to attempt to keep clock drift under one microsecond, we would have to resync approximately every 36 milliseconds, which would be impractical.

In my actual implementation, I used multiple threads for each UPD time request so the requests were done concurrently. The time servers returned their server times and I calculated a round trip time to create the time ranges. Note that you can hit enter to do another request and time calculation when running the program. Entering 'q' will end the program. I chose not to update my physical clock but merely output results. This was written in Eclipse with Java 1.7.0_51-b13 on a MacBook Pro running OS X 10.10.