1 Overview

Consider the setup shown in Fig. 1 with a computer system time synchronizing to a sensor of some sort; assume that both systems are able of transmitting and receiving packets, and both have a local clock that does not overflow (not necessarily counting in the same units of time).

The goal is to have a sensor timestamp (i.e., on the time scale of the sensor) of an event occurring at the sensor translated in the timescale of the computer with as small of an error as possible.

To this end the computer system is sending (not necessarily periodically) timestamped probe request packets to the sensor, the sensor adds its own timestamp to the packet, and then replies with a probe reply. Upon receiving the probe reply, the computer adds a third and final timestamp. Denote the three timestamps by $t_o$ (origin), $t_b$ (bounce), and $t_r$ (return). The origin and the return timestamps $t_o$ and $t_r$ are in the timescale of the computer, while the bounce timestamp $t_b$ is in the timescale of the sensor.

Each triplet $(t_o, t_b, t_r)$ is a data point. When receiving a new data point (i.e., after timestamping a probe reply), the computer re-evaluates the best estimate for the time correction.

The algorithm assumes a linear time relation between the computer time $t_c$ and the sensor time $t_s$:

$$t_c = at_s + b.$$  \hfill (1)

The goal of the algorithm is to compute the best estimates of $a$ and $b$ (denoted with $\hat{a}$ and $\hat{b}$ respectively). The algorithm is actually computing lower bounds on both $a$ and $b$ such that $a_{\text{min}} \leq a \leq a_{\text{max}}$ and $b_{\text{min}} \leq b \leq b_{\text{max}}$. In the absence of a better estimate, then the two estimates are the middle of their bounding interval:

$$\hat{a} = \frac{a_{\text{max}} + a_{\text{min}}}{2},$$  \hfill (2)

and

$$\hat{b} = \frac{b_{\text{max}} + b_{\text{min}}}{2}.\hfill (3)$$

For each new received data point, the first test is whether this new datapoint results in a tighter estimate for either of the bounds on $a$ and $b$. If it does not for either of them, then it is discarded. If it is, then
it is kept and any existing data point that is no longer helping in tightening the bound is being discarded instead. At most four datapoints are thus ever stored.

The final problem to solve is resetting the algorithm: it is possible that at some point, the assumption of a linear relation between the two times does not hold anymore. We detect that moment when either one of the two bounding intervals (i.e., either \((a_{\text{min}}, a_{\text{+}})\) or \((b_{\text{min}}, b_{\text{max}})\)) become invalid (i.e., when \(a_{\text{min}} \geq a_{\text{max}}\) or \(b_{\text{min}} \geq b_{\text{max}}\)). When this occurs, the relation between the two times followed a curve that cannot be approximated by the linear relation in (1). In this case, we discard the oldest datapoint and re-evaluate the bounds.

2 Pseudo-code

In what follows we present the pseudo-code for both the computer and the sensor.

2.1 At the Sensor

The sensor does a single thing for this algorithm: it listens for an incoming probe request, and it generates a probe reply.

Procedure Reply()

while(1)
    Listen for an incoming probeRequest containing to // blocks
    \(tb = \text{getCurrentTime()}\)
    send probeReply with \(tb, tr\)

2.2 At the Computer

For the computer there are several processes that are ongoing:

Procedure SendProbe()

while(1)
    to = \text{getCurrentTime()}\)
    send probeRequest containing to
    sleep(X seconds) // X can be adjusted based on traffic load
    // once a second is a reasonable rate

Procedure Main()

dataPointList = [] // empty
ahat = invalid // initially an invalid a estimate
bhat = invalid // initially an invalid b estimate
// we'll update both when we have enough datapoints

while(1)
    Listen for incoming ProbeReply // blocks
    tr = \text{getCurrentTime()}\
    extract to and \(tb\) from ProbeReply
    addNewPoint([to, \(tb, tr\]])

Procedure addNewPoint(newDataPoint)
// Adds a new point to the buffer, but will be kept only if it
// further constrains the intervals for \([amin, amax]\) and \([bmin, bmax]\)

    // add the point for now
dataPointList = [dataPointList newDataPoint]

// if only one dataPoint is available, then we can only estimate bhat,
// and we assume ahat=1 as best guess. If the two time scales are different
// and the difference in rates is known, then that should be used instead of 1
// if the difference is not known, then ahat and bhat should be left invalid
// for example, if both the sensor and the computer count microseconds,
// then ahat = 1 as best guess
// if the sensor counts on a clock of 8MHz and the the computer counts
// microseconds, then ahat = 8 as best guess
if length(dataPointList)==1 then
    (to, tb, tr) = dataPointList[0] // get the elements of the datapoint
    ahat = 1; // best guess
    bmin = tr-tb;
    bmax = to-tb;
    bhat = (bmin+bmax)/2
    return // we computed the ahat,bhat estimates from the only point we have,
    // we’ll do better next time after a second data point shows up

//now given the new, longer, list, compute the intervals for a and b
//and remove the dataPoints that do not help in constraining a and b
//goodIndexes is a list of the indexes of useful data points
([amin amax bmin bmax], goodIndexes) = ComputeConstraints(dataPointList)

// now for the kicker, if amin >= amax or bmin >= bmax
// we need to reset the process: we throw away the oldest point
// otherwise, throw away the useless ones
if (amin < amax) and (bmin < bmax) then // normal case
    throwAwayAllPointsFromDataListNotInTheIndexList(dataPointList, goodIndexes)
else // need to reset, throw away the oldest point
    dataPointList = dataPointList[1:length(dataPointsList)]
    // and recompute the constraints:
    ([amin amax bmin bmax], goodIndexes) = ComputeConstraints(dataPointList)

    ahat = (amin+amax)/2
    bhat = (bmin+bmax)/2
    return

Procedure ([amin amax bmin bmax], goodIndexes) = ComputeConstraints(dataPointList)
// Computes constraints on ahat and bhat given a list of data point
// (amin < ahat < amax) and (bmin < bhat < bmax)
// it also returns the indexes of the points used to constrain the intervals
amin = bmin = -infinity // initialize for min and max
amax = bmax = infinity

//we’ll keep track on which points are used and remove the rest
//we keep the good ones for [amin amax bmin bmax]
//likely we’ll only keep two at any time; four at the most
goodIndexes = [-1,-1,-1,-1,-1,-1,-1,-1];

//We’ll iterate through each pair of data points (i,j)
for i from 0 to length(dataPointList)-1
  for j = i+1 to length(dataPointList)
    point1 = dataPointList[i]
    point2 = dataPointList[j]
    [a_low, a_high, b_low, b_high] = ComputeTwoLines(point1, point2)
    if amin < a_low % found a better amin constraint
      amin = a_low
      goodIndexes[0] = i;
      goodIndexes[1] = j;
    if amax > a_high % found a better amax constraint
      amax = a_high
      goodIndexes[2] = i;
      goodIndexes[3] = j;
    if bmin < b_low % found a better bmin constraint
      bmin = b_low
      goodIndexes[4] = i;
      goodIndexes[5] = j;
    if bmax > b_high % found a better bmax constraint
      bmax = b_high
      goodIndexes[6] = i;
      goodIndexes[7] = j;
  // end for j
//end for i

/ * optionally, remove all duplicate indexes in goodIndexes

return ( [amin amax bmin bmax], goodIndexes )

Procedure [a_low, a_high, b_low, b_high] = ComputeTwoLines(point1, point2)
// Computes the slopes and y intercepts for the two lines passing through the
// two datapoints
(t01, b1, tr1) = datapoint1;
(t02, tb2, tr2) = datapoint2;
[a_low, b_high] = SlopeAndIntercept(tb1, tr1, tb2, to2)
[a_high, b_low] = SlopeAndIntercept(tb1, to1, tr2, tr2)
return (a_low, a_high, b_low, b_high)

Procedure [a, b ] = SlopeAndIntercept(x1, y1, x2, y2)
// Computes the slopes and y intercept for the line passing through the
// two points
if (x2 != x1)
  a = (y2 - y1)/(x2 - x1);
else
a = 0
b = y1 - x1*a
return (a,b)

Procedure computerTimeStamp = translateTime(sensorTimeStamp)
// this procedure takes a timestamp on the sensor’s time scale
// and returns the corresponding time (best estimate) on the
// computer’s time scale
if ahat or bhat are invalid then
    return invalid answer
else
    return ahat*sensorTimeStamp + bhat

Finally a few notes:

• use double as variable type for all computations, as float tends to lose significant digits;

• the algorithm assumes that the sensor is able to send a probe reply as soon it receives a probe request (or soon after). If this is not true, it’s relatively easy to use two timestamps at the sensor: $t_{bin}$ and $t_{b_{out}}$ for the times of the receipt of the probe request and sending the probe reply respectively. Each data point in this case consists of four timestamps: $(t_o, t_{bin}, t_{b_{out}}, t_r)$, and the two pair of constraints per datapoint are $(t_{bin}, t_o)$ and $(t_{b_{out}}, t_r)$ instead of $(t_b, t_o)$ and $(t_b, t_r)$ as they are in the pseudocode above.

• if you know any fixed delay $D$ (e.g., a transmission time, propagation time, etc.), you can get better precision if you subtract it from the computer timestamps, i.e., the datapoint becomes $(t_o+D, t_b, t_r-D)$

• the precision of the algorithm increases (slightly) the longer it runs (as it collects new datapoint), and it worsens again everytime it resets (i.e., when it throws away the oldest datapoint).