Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Systems@ETH Zürich

Prof. T. Roscoe, Prof. R. Wattenhofer

# Computer Systems

**— Solution to Assignment 9 —**

## 1 Clock Sync

### 1.1 Clock Synchronization

**a)** Propagation delays are asymmetric!

**b)** Determine the range between speakers and NTP servers to factor out the signal propagation delays. The remaining time should be the processing time of the signal in the NTP server.

**c)** False. Assume a chain of nodes, each one's clock is equal to its left neighbor's clock plus $x$. The average local skew is $x$. However, the average global skew can be much larger, depending on the size of the chain.

**d)** False. Assume a chain of nodes again. Nodes at even positions have clock $c$ and nodes at odd positions $c+x$. Clearly, the average global skew is smaller than $x$. However, the average local skew is $x$.

### 1.2 Time Difference of Arrival

**a)** The time difference of arrival (3.3 µs) leads to a range difference for satellites $A$ and $B$ of $3.3\ \mu s \cdot 3 \cdot 10^8$ m/s $\approx 1km$. The relation between our location and the range difference is $\|p_B - p\| - \|p_A - p\| \approx 1$ km with $p = (x, -x + 8)$. Hence, we want to minimize the square of the residual $r = \left\|p^B - p\right\| - \left\|p^A - p\right\| - 1$ km.

**b)** The residual for point (2 km, 6 km) is much smaller than the residual for point (4 km, 4 km). Hence, point (2 km, 6 km) is more likely to be the correct location assuming that the measurements are correct.

**c)** The distance between satellite $B$ and position $(2\text{ km}, 6\text{ km})$ is 5 km. Therefore, the time is $t + 16.7$ µs when the signal arrives at $B$.

### 1.3 Clock Synchronization: Spanning Tree

The grid is composed of cells and nodes. The nodes are shown as black dots in Figure 1 and the cells are the areas between four (neighboring) nodes. Now we look at an arbitrary cell in the grid. For any tree we can draw on this grid, there has to be a way to walk out of the grid without crossing the edges of the tree as shown in Figure 1 because there are no loops in a tree. This holds for every cell in the grid, especially for the cell in the middle (or adjacent to the center node) of the grid. Let us assume we leave the grid between two nodes B and C. These two nodes are neighbors on the grid, since otherwise the nodes in between would not be connected to the spanning tree. If $m$ is even, then the distance between Node A (adjacent to the middle cell, opposite the exit side)

and Node B or C, respectively, is at least $\frac{m}{2}$. If $m$ is odd, either of the two routes can be $\frac{m-1}{2}$ but the other is at least $\frac{m+1}{2}$, since the distance from A to the edge of the grid is $\frac{m-1}{2}$, but for one of the nodes one needs to make at least one step in the perpendicular direction.

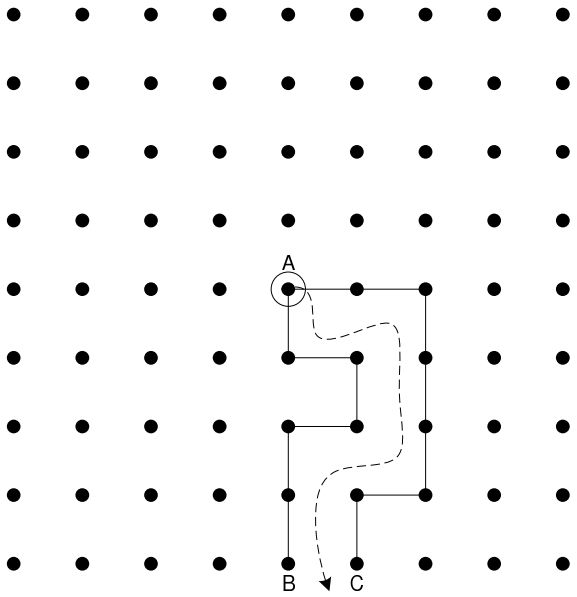


Figure 1: Node A is the center node of the spanning tree (only partially shown). The dashed line is a path through grid cells from the center Node A to the outside of the grid.

## 1.4 NTP Programming

Here is a C++ program:

```
#include <iostream>
#include <math.h>
#include <sys/timex.h>
#include <time.h>
#include <iomanip>

int main()
{
       struct timex buf;
       buf.modes = 0; // read
       adjtimex(&buf);

       double posix_time = buf.time.tv_sec;
       if (buf.status & STA_NANO)
       {
              // nanoseconds used
              posix_time += buf.time.tv_usec * 1.0e-9;
       }
       else
       {
              // microseconds used
              posix_time += buf.time.tv_usec * 1.0e-6;
       }
```

```
        // Note: This code ignores that we could be "inside" a leap second.
        time_t posix_whole_secs = (time_t)buf.time.tv_sec;
        struct tm* current_time = gmtime(&posix_whole_secs);
        double seconds = fmod(posix_time, 60.0);

        float max_error = buf.maxerror * 1.0e-6;

        std::cout << "Current␣Time:␣" << std::setfill('0')
              << current_time->tm_year + 1900 << "-"
              << std::setw(2) << current_time->tm_mon + 1 << "-"
              << std::setw(2) << current_time->tm_mday << "T"
              << std::setw(2) << current_time->tm_hour << ":"
              << std::setw(2) << current_time->tm_min << ":"
              << std::setw(6) << std::fixed << std::setprecision(3)
              << seconds << "Z␣+-␣"
              << max_error << "␣s\n";

        return EXIT_SUCCESS;
}
```

# 2 Consistency and Logical Clocks

## 2.1 Different Consistencies

**a)** This is a direct corollary of Lemma 19.10, Lemma 19.12 and Lemma 19.13: Assume for the sake of contradiction that sequential consistency implies linearizability. Since according to Lemma 19.12 linearizability implies quiescent consistency this would mean that sequential consistency implies quiescent consistency which is a contradiction to Lemma 19.13. An analogous derivation can be done to disprove that quiescent consistency implies linearizability.

**b)** We disprove the statement by a contradicting example: Assume that we have three nodes $u$, $v$ and $w$ that each execute one operation on the same object with initial value 2. $u$ and $v$ execute *inc* (increment the object by 1) and $w$ executes *double* (multiply the object by 2) with $inc^v_* < inc^u_\dagger < double^w_* < inc^v_\dagger$. Since there is no quiescent point and all nodes only have one operation, quiescent and sequential consistency are both fine with a sequential execution that doubles first and then increments: $((2 \cdot 2) + 1) + 1 = 6$. Linearizablility on the other hand requires $inc^u < double^w$ to be in this order in $S$.

## 2.2 Measure of Concurrency from Vector Clocks

Here the Python code:

```
# some random possible state of the system after one message was sent from node
    ↪ 1 to node 0
log0 = [[1, 5]] # list of logged vector clocks of node 0
log1 = [] # list of logged vector clocks of node 1
c0 = [20, 5] # current state of vector clock at node 0
c1 = [0, 10] # current state of vector clock at node 1

muS = c0[0] + c1[1] + 1
muC = (1 + c0[0]) * (1 + c1[1])

# number of consistent snapshots before receiving the message:
```

```
mu1 = (log0[0][0]) * (1 + c1[1])

# number of consistent snapshots after receiving the message:
mu2 = (1 + c0[0] - log0[0][0]) * (1 + c1[1] - log0[0][1])

# total number of consistent snapshots:
mu = mu1 + mu2

measureOfConcurrency = (mu - muS) / (muC - muS)
print(measureOfConcurrency)
```

And here the more general code:

```
# some random possible state of the system
log0 = [[1, 13], [14, 26]] # list of logged vector clocks of node 0
log1 = [[13, 14]] # list of logged vector clocks of node 1
c0 = [30, 26] # current state of vector clock at node 0
c1 = [13, 26] # current state of vector clock at node 1


muS = c0[0] + c1[1] + 1
muC = (1 + c0[0]) * (1 + c1[1])

# terminating is equivalent to receiving a message in the next operation for
    ↪ this calculation
terminalClock = [c0[0] + 1, c1[1] + 1]
log0.append(terminalClock)
log1.append(terminalClock)

# start of uninterrupted execution:
start0 = 0
start1 = 0

# end of uniterrupted execution:
nextReceive1 = log1[0][1]

mu = 0
idx1 = 0
for l0 in log0:
  mu += (l0[0] - start0) * (nextReceive1 - start1)
  # go through every message received by 1 before the next message received by 0
  for idx in range(len(log1) - 1):
    if start0 < log1[idx][0] < l0[0]:
      lastReceive1 = nextReceive1
      nextReceive1 = log1[idx + 1][1]
      mu += (l0[0] - log1[idx][0]) * (nextReceive1 - lastReceive1)
  # go to next message received by 0
  start0 = l0[0]
  start1 = l0[1]

measureOfConcurrency = (mu - muS) / (muC - muS)
print(measureOfConcurrency)
```