Synchronising Software Clocks on the Internet

Darryl Veitch*

Collaboration with

Satish Babu† and Attila Pásztor†

Supported by Ericsson Australia

* CUBIN, Dept. of Electrical & Electronic Engineering, University of Melbourne.
† IIT, New Delhi.
† Ericsson Hungary R&D, Budapest.

Motivation

Everyone needs a *Software Clock*:

- Physical support (e.g. timer chip)
- Synchronisation (absolute, and rate)
- Timestamping

And wants performance:

- Inexpensive (off the shelf hardware)
- Inexpensive, convenient synchronisation (off a network)
- Accurate and Robust
Motivation

Everyone needs a *Software Clock*:  
- Physical support (e.g. timer chip)
- Synchronisation (absolute, and rate)
- Timestamping

And wants performance:  
- Inexpensive (off the shelf hardware)
- Inexpensive, convenient synchronisation (off a network)
- Accurate and Robust

*Widely used SW-NTP solution in PC’s not good enough*  
- *Varies rate* to drive offset to 0 – but rate essential for $\Delta(t)$ measurement!
- *Not robust!* jumps can occur, ms to seconds or worse
So Why Can We Do Better?

Advances in hardware have changed the status quo,

From

*Local clock is very unreliable, must ask (and trust) the expert*

To

*Local clock is excellent, must simply calibrate it (and mistrust the expert)*
So Why Can We Do Better?

Advances in hardware have changed the status quo,

From

   _Local clock is very unreliable, must ask (and trust) the expert_

To

   _Local clock is excellent, must simply calibrate it (and mistrust the expert)_

Still need reference source

- use (nearby stratum–1) NTP servers, and NTP protocol
So Why Can We Do Better?

Advances in hardware have changed the status quo,

From

*(Local clock is very unreliable, must ask (and trust) the expert)*

To

*(Local clock is excellent, must simply calibrate it (and mistrust the expert)*

Still need reference source

- use (nearby stratum–1) NTP servers, and NTP protocol

But new perspective: *local* centric & *rate* centric

- inspires new filtering philosophy → greater accuracy and robustness
- defines separate *difference* and *absolute* clocks
- we deliver both *rate* and *offset* estimates
The CPU Oscillator as a Clock

TSC (or CCC) register counts CPU cycles, 1 cycle per $p$ [sec].

Two simple ideas, based on *Simple Skew Model* (pure frequency):

- **Difference Clock**: $\Delta(t) = \Delta(\text{TSC register value}) \times p$
- **Absolute Clock**: $t = \theta_0 + (\text{TSC register value}) \times p$
The CPU Oscillator as a Clock

TSC (or CCC) register counts CPU cycles, 1 cycle per $p$ [sec].

Two simple ideas, based on *Simple Skew Model* (pure frequency):

- **Difference Clock:** $\Delta(t) = \Delta(\text{TSC register value}) \times p$
- **Absolute Clock:** $t = \theta_0 + (\text{TSC register value}) \times p$

Features/Advantages

- CPU oscillator and TSC Register stable features of PC architecture
- Hardware updating
- Ultra fast raw timestamping (read register): ($< 50 \text{ ns}$)
- Nanosecond resolution
The CPU Oscillator as a Clock

TSC (or CCC) register counts CPU cycles, 1 cycle per $p$ [sec].

Two simple ideas, based on *Simple Skew Model* (pure frequency):

- **Difference Clock**: $\Delta(t) = \Delta(\text{TSC register value}) \times p$
- **Absolute Clock**: $t = \theta_0 + (\text{TSC register value}) \times p$

Features/Advantages

- CPU oscillator and TSC Register stable features of PC architecture
- Hardware updating
- Ultra fast raw timestamping (read register): (< 50 ns)
- Nanosecond resolution

The Catch:

- must estimate the cycle period $p$ and offset $\theta_0$ without special hardware.
Study of Oscillator Stability (variability) over different timescales shows:

- **Simple Skew Model**: holds strictly for $\tau^* = 1000$ [sec]
- **Average rate error** never more than $10^{-7}$ no matter the scale

**Very smooth constant rate: must take advantage!**

**Measured using reference**: GPS synchronised DAG3.2E card (100ns)
timestamps \{T_{a,i}, T_{b,i}, T_{e,i}, T_{f,i}\} are the raw data from the \(i\)-th NTP packet.

- \{T_{a,i}, T_{f,i}\}: host timestamps in TSC counter units
- \{T_{b,i}, T_{e,i}\}: server timestamps in seconds
Filtering Network and Server Delay

Choose RTT based filtering, not one-way (using same clock good!)

Round-Trip Times $r_i$ of packet $i$

Model for RTT: $r_i = r + \text{positive random noise}$

Filter using point error: excess over minimum RTT
Naive Rate Synchronisation

Wish to exploit the relation $\Delta(t) = \Delta(TSC) \times p$

Simple naive estimate based on widely separated packets:

Network delay and timestamping noise $\sim 1/\Delta(TSC)$, but errors not bounded.
Use selected naive estimates based on point errors

Key Features

- Error quickly $< 10^{-7}$, In 10mins, better than GPS for most active probing!
- Error reduction (in timestamping, latency, queueing) guaranteed by increasing $\Delta(t)$
- Inherently robust to packet loss, congestion, loss of server..
- Simple algorithm, no need for local estimates
Naive Offset Synchronisation

Wish to exploit SKM over small scales to measure $\theta(t)$

**Naive estimate** again ignores network congestion:

![Graph showing offset estimates over time](image)
Offset Synchronisation Algorithm

Must track, so use all naive estimates, but be very fussy

Algorithm for $\hat{\theta}(t)$

- For packets in SKM window $\tau'$ wide before $t$, define strict weight
- Form weighted estimate over window (If quality very bad, repeat previous packet)
- Sanity check: if behaviour impossible, repeat previous estimate.
Offset Synchronisation Algorithm

Must track, so use all naive estimates, but be very fussy

Algorithm for $\hat{\theta}(t)$

- For packets in SKM window $\tau'$ wide before $t$, define strict weight
- Form weighted estimate over window (If quality very bad, repeat previous)
- Sanity check: if behaviour impossible, ignore.
The Path Asymmetry $\Delta$

Fundamental ambiguity:

Asymmetry $\Delta \equiv d^{\rightarrow} - d^{\leftarrow}$ and error in offset indistinguishable up to a constant.

$\Delta$ unknown: forced to assume $\Delta = 0$

But, error bounded by RTT: $\Delta \in (-r, r)$
## Server Characteristics: RTT and $\Delta$

<table>
<thead>
<tr>
<th>Server</th>
<th>Reference</th>
<th>Distance</th>
<th>RTT</th>
<th>Hops</th>
<th>$\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServerLoc</td>
<td>GPS</td>
<td>3 m</td>
<td>0.38 ms</td>
<td>2</td>
<td>$\approx 0.05$ ms</td>
</tr>
<tr>
<td>ServerInt</td>
<td>GPS</td>
<td>300 m</td>
<td>0.89 ms</td>
<td>5</td>
<td>$\approx 0.05$ ms</td>
</tr>
<tr>
<td>ServerExt</td>
<td>Atomic</td>
<td>1000 km</td>
<td>14.2 ms</td>
<td>$\approx 10$</td>
<td>$\approx 0.5$ ms</td>
</tr>
</tbody>
</table>

For **ServerInt**:  $\Delta \approx 50 \mu s$

Theoretical best offset error is:  $\Delta/2 \approx 25 \mu s$

Measured median offset error only:  $28 \mu s$

Choice of server is crucial:

- Close server has small RTT
- Close server more likely to have symmetric path:  $\Delta \ll r$
- **Aim**: find a stratum–1 NTP server, with small $r \sim 1$ ms, and known, symmetric path.
Build in Robustness to Diverse Factors

- Changes in environment/oscillator
- Changes in temperature
- Packet Loss (loss of connectivity..)
- Network congestion
- Timestamping errors (scheduling)
- Server errors
- Route/Server changes – Level shifts in RTT
  Possible to handle level shifts reliably with almost no dedicated code
Histograms show 99% of all values, \( \tau' = 2\tau^* \)
Performance as Function of Window Size $\tau'$

ServerInt, Machine-Room, $E = 4\delta$
Performance as Function of Polling Period

\[ \text{Offset Error} \ [\mu s] \]

\[ \text{Polling Period} \ [\text{sec}] \]

\text{ServerInt, Machine-Room, } \tau' = \tau^*, \ E = 4\delta
Conclusions

- Defined **CPU oscillator based** software clock for devices with TSC registers
- Absolute Clock:
  - more accurate than *SW–NTP*
  - far more robust
- Difference Clock:
  - not available under *SW–NTP*
  - more accurate than *SW–GPS* for many applications
- Low computational requirements
- Suited to network measurement, or as generic replacement for *SW–NTP*
Web Resources

- Publications (including preprint):

- Software and documentation (Linux) for TSC clock and early $p$ calibration:

- Software for new algorithms: BSD implementation coming

- TSC based sender/receiver (Linux, LinuxTSC, RT-Linux) for active probing: