2007 International IEEE Symposium on Precision Clock Synchronization (ISPCS)
for Measurement, Control and Communication
Vienna, Austria, October 1-3, 2007.

Ten Microseconds Over LAN, for Free

Julien Ridoux, Darryl Veitch
ARC Special Research Center for Ultra-Broadband Information Networks (CUBIN)
An affiliated program of National ICT Australia, EEE Department
The University of Melbourne, Australia
{j.ridoux, d.veitch}@ee.unimelb.edu.au

Abstract

The status quo for timestamping in PCs is ntpd, which is accurate to 1[ms] at best. For precision applications this is inadequate, but it is a low cost solution which suits many generic applications. IEEE-1588 provides mechanisms for sub-microsecond accuracy, but to achieve this more hardware is needed. We have developed the TSCclock, which gives performance between these two, around 10 microseconds on LAN, sub millisecond beyond, but using commodity hardware. We begin detailed benchmarking of the TSCclock to show its potential as an inexpensive yet accurate software clock, which could be used with IEEE-1588 for LANs, but has wider applicability as a replacement to ntpd.

1 Introduction

Clock performance, like any system, is subject to tradeoffs of cost and ease of use. Consider the status quo for software clocks in PCs, the *ntpd* daemon [1], which is used to synchronise system software clocks via a time server hierarchy across networks. It is widely recognised that *ntpd* is inadequate for precision timekeeping, being limited to around 1[ms] at best (see figure 2). However, for many applications this is sufficient, and it does enable a time server to be inexpensively accessed over networks ranging from LANs to the Internet.

Over LANs, IEEE-1588 implementations provide for far more precise synchronisation, at the microsecond level to well below if hardware support is provided, such as when a hardware clock is embedded into a Network Interface Card (NIC). This large improvement in terms of clock performance brings with it two main constraints. First, the additional hardware comes at a financial cost, and makes legacy hardware incompatible with IEEE-1588. Second, since the IEEE-1588 protocol must be implementable in hardware, its design has to remain relatively simple. This is a fundamental constraint restricting the usage of the IEEE-1588 protocol to LAN islands.

There is a wide gap between the quality and constraints of these two timing solutions, *ntpd* and the hardware based IEEE-1588. This leaves open the prospect for a solution which lies in between that would be reliable, able to provide synchronisation within or across LANs, whilst remaining inexpensive and compatible with legacy hardware. Over the last few years ([2, 3, 4, 5, 6]) we have developed a solution which takes this middle ground: exploiting the relatively high stability of commodity hardware to provide a robust clock with accuracy well above that of the *ntpd* solution, whilst retaining the low cost and ease of use of network based synchronisation. The *TSCclock* has as its hardware basis the TSC register, available in common architectures, which counts CPU cycles. Its synchronisation algorithms, based on a client-server paradigm, are effective in filtering out delay jitter from network elements and the host system, and we use selective kernel modifications to dramatically improve timestamping performance at zero dollar cost.

To support the first release of the TSCclock over Linux and BSD systems, we are performing a set of comprehensive benchmarking studies. Using weeks of live data, we give preliminary results showing performance over Ethernet LANs to be around 10μs, a noise level essentially set by the jitter inherent in the multi-tasking operating system. Whilst not being able to compete with more hardware oriented solutions over LAN, this raising of the bar by an order of magnitude nonetheless opens up numerous possibilities for applications where cost is a factor.

2 The TSCclock

The TSCclock [7, 3, 2] leverages the fairly high stability of commodity oscillator hardware, specifically the CPU oscillator, whose cycles are conveniently counted and ready accessible via the TSC register in common PC architectures. Contrary to the *ntpd* based software clocks, the TSCclock does not actively vary its rate to track drift in a feedback loop. Instead, it is built around obtaining stable long-term clock rate estimates using a feedforward approach. This enables it to provide two clocks, one for time differences, and one for absolute time. This approach circumvents the usual tradeoff problem, for example in PID controllers, whereby gain parameters which improve short term tracking do so at the detriment of rate stability over the time-scales at which tracking operates.

The synchronisation algorithm operates in client-server

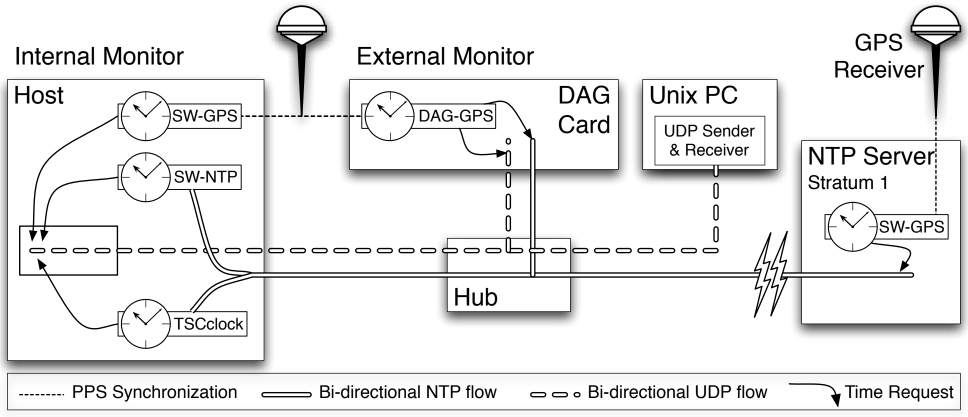

Figure 1. Testbed.

mode. Each round-trip generates four timestamps, two in `timeval` format from the server, and two raw TSC timestamps taken at the host. These timestamp 4-tuples are used to provide an estimate $\hat{p}$ of the average CPU oscillator period p. An *uncorrected* clock which does not take clock drift into account can then be defined as $C_u(t) = \hat{p} \cdot TSC + K$, where K is an estimate of the initial offset which is **not** updated. Instead, an estimate $\hat{\theta}$ of the error in $C_u(t)$ is updated at each new incoming stamp. The two clocks are then:

$$\mathbf{C}_d(t): \quad \Delta(t) = C_u(t_2) - C_u(t_1) = \Delta(TSC) \cdot \hat{p}, \quad (1)$$

$$\mathbf{C}_a(t): \quad C_a(t) = C_u(t) - \hat{\theta} = \hat{p} \cdot TSC + K - \hat{\theta}. \quad (2)$$

The difference clock $C_d(t)$ does not incorporate the correction $\hat{\theta}$, so the constant K cancels exactly. Hence, $C_d(t)$ directly benefits from the underlying rate stability of the TSC over small to medium timescales, and is not perturbed by estimates of drift, which are irrelevant over those scales. For example, a rate stability of 1 part in 10^7 over a RTT of 1 [sec] yields an error of only 0.1μs.

Measuring absolute time requires that drift be tracked. Hence, $\hat{\theta}$ is incorporated into the definition of $C_a(t)$, which results in medium term variability since estimates must be based on a limited time window, and used even if not ideal (for example due to congestion). However, by not changing K, instead applying a correction only when reading, the absolute clock avoids varying its underlying rate to track drift, which improves stability and enables meaningful sanity checking.

The key problem in synchronising clocks over a network is the variable delays due to queueing in network elements, and interrupt, queueing and processing delays in the host and/or server hardware and operating systems. Whereas in systems using the *Precision Time Protocol* (PTP, ie. IEEE-1588), these effects can be reduced to near zero by the use of boundary clocks which are typically hardware based (see however [8]). The TSCclock is designed to be robust to large and highly variable delay jitter as it receives its timestamps via packets which travel across the network and back. Thus, apart from the feed-forward design described above, the entire structure of the TSCclock synchronisation algorithms is aimed at compensating for this jitter successfully, that is both accurately and robustly. If there were no jitter, then synchronisation reduces to the calculation of propagation delays, which apart from path asymmetry issues (which are intrinsic and cannot be overcome by any algorithm), is trivial.

The TSCclock performs non-linear filtering based on round-trip times. It uses a simple but empirically very well justified model of round-trip times, namely a minimum constant plus positive random noise. This approach is extremely effective in identifying those packets which contain the best (smallest) network delays. The estimators of $\hat{p}$ and $\hat{\theta}$ above are based on these together with windowing for variance reduction. More precisely, the excess of RTT above an estimate $\hat{r}$ of the minimum RTT r is used as a basis of rejection of distorted timestamps when measuring $\hat{p}$, and as a weight when averaging estimates made over several packets in the case of $\hat{\theta}$. A more detailed account is given in [7, 3], including how to deal with changes in the minimum delay level, which can occur for example following changes to layer 2 or 3 routing. The resolution of the TSCclock is tied to that of the CPU, and is around 1[ns].

3 Testbed

Any software clock running on typical computer architecture and operating systems has to deal with system delay created by resource sharing and competition among running processes. As a general term, we refer to these delays as *system noise*. System noise affects the timestamping of any event of interest by delaying the reading of the clock. In other words, timestamps of events used to synchronise a software clock (arrival time of a reference time packet for example) or events used to assess clock performance, are all affected. Based on this simple observation, we designed a testbed to gain insights into system noise and so distinguish timestamping errors from actual

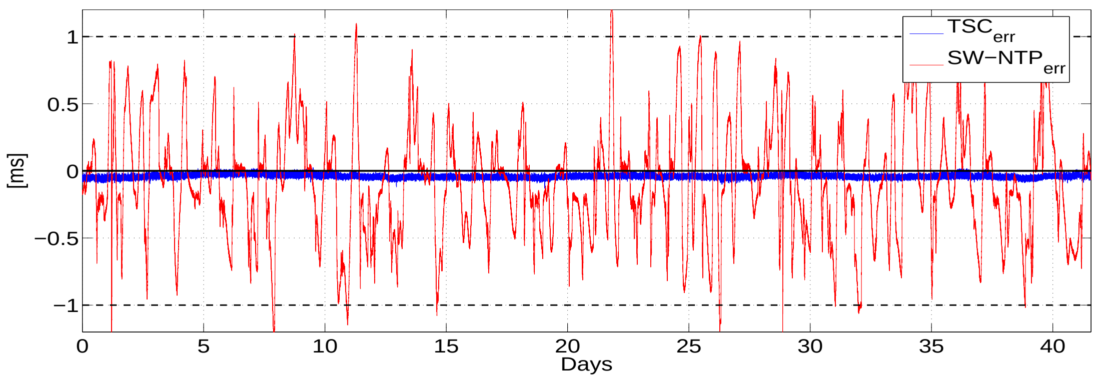

Figure 2. Performance of a *ntpd* based SW clock (resp. TSCclock) using servers inside (resp. outside) the LAN.

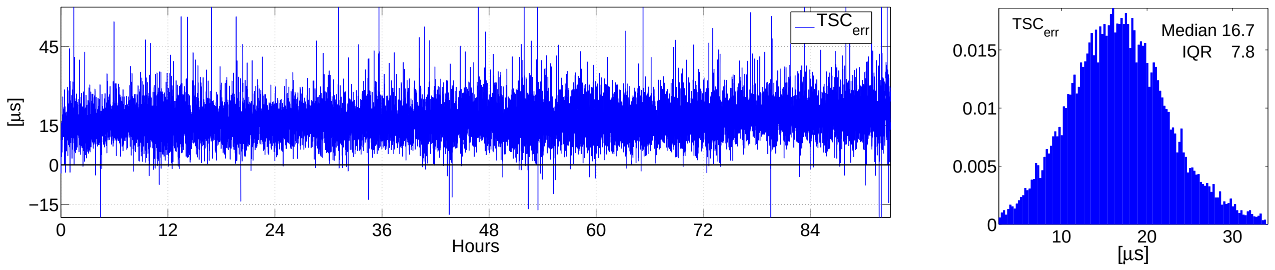

Figure 3. TSCclock synchronised to stratum-1 server on LAN, polling period 256s, Left: clock errors using external DAG comparison, Right: histogram.

clock errors.

Figure 1 shows our testbed, consisting of a PC host running three clocks: SW-GPS (*ntpd* synchronised to local GPS PPS), SW-NTP (*ntpd* synchronised to stratum-1 NTP servers), TSCclock (synchronised to stratum-1 NTP servers), and a precision external reference (GPS synchronised DAG card).

To assess the performance of each of the three clocks, UDP packets are exchanged between the Host and another Unix computer on the network. Packets are sent from the host monitor and for each packet sent, the Unix PC replies with a similar UDP packet. This packet exchange constitutes the series of events used to trigger the reading of each of the clocks and translates into several series of timestamps which can be compared.

The comparison of the timestamp time-series made available by our testbed is twofold and we refer to it as the *Internal* and *External* comparisons. The Internal comparison relies on slight modifications to the Linux and FreeBSD kernels for the timestamping of each incoming or outgoing UDP packet. The main objective of these modifications is to be able to timestamp each packet event using two clocks in a “back to back” fashion. The packet timestamping is done using the packet capture library *libpcap*. The modifications applied to the Linux and FreeBSD kernel are different and driven by the actual implementation of libpcap on these two platforms. On FreeBSD, the call to timestamping functions takes advantage of the Berkeley Packet Filter subsystem, but our modifications slightly improve the timestamping location for each of the clocks, moving them slightly closer to the network card driver code. On Linux kernels, timestamps are taken in the kernel, right after the driver code returns and are exported using RAW sockets opened by libpcap.

We carefully implemented these new and quasi-simultaneous timestamping calls so that timestamps of the same event obtained from different clocks share the same system noise. As a result, the internal comparison provides timestamps for which the clocks share the same timestamping error. When comparing corresponding timestamps from two different clocks the timestamping error cancels, revealing the relative performance of the two clocks under study. However, although free of timestamping error, this comparison does not provide an indication of absolute performance as none of the clocks used can be considered as references.

To quantify the clocks absolute performance, a trusted reference clock is needed. For the purpose of packet timestamping we use a 3.7GP DAG card ([9]). The DAG card embeds its own hardware clock synchronised us-

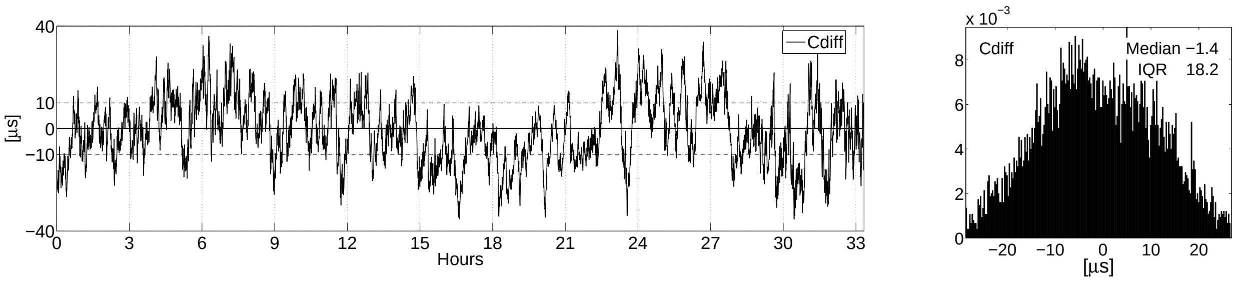

Figure 4. TSCclock synchronised to stratum-1 server two hops away, polling period 16s, internal in-host comparison against SW-GPS.

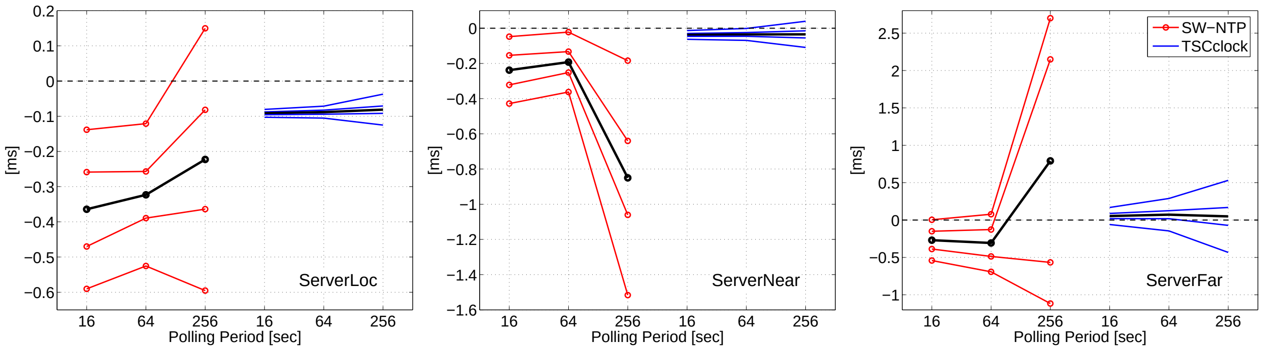

Figure 5. TSCclock versus *ntpd* for three polling periods and three time servers, left to right: LAN; same campus; across the continent.

ing a GPS Pulse-Per-Second input and timestamps packets at hardware level. Using this accurate and low system noise timestamping device, we consider the series of timestamps obtained to be our absolute time reference.

Combining both Internal and External comparison we are able to give a lower bound on system noise at the host. We are also able to provide bounds on, as well as remove, the inherent ambiguity of the results due to the network asymmetry existing in the exchange of NTP packets. More details, a precise description of the methodology associated to the testbed, and the analysis of operating system noise and path asymmetry, can be found in [6].

4 Experimental Results

Figure 2 motivates our work. It shows *ntpd* performance under ideal conditions: the SW-NTP clock running on the host being synchronised to a stratum-1 peer on the same LAN. Despite this, its absolute error (given by the external comparison using DAG packet timestamps) varies in a 1[ms] band. In contrast, the TSCclock, synchronised to a stratum-1 NTP server *outside* the LAN, is an order of magnitude smaller.

From the perspective of PTP, we are interested in seeing how well the TSCclock can perform in a LAN environment. Figure 3 gives the absolute performance of the TSCclock over 92 hours, measured externally via DAG. In this experiment, the TSCclock is synchronised to a GPS synchronised stratum-1 server located on the same LAN. Again, the TSCclock relies on NTP packets to exchange time information with the server and to synchronise to it. The inter-quartile range of this external comparison is only 7.8μs, and the median error, once path asymmetry effects (measured at 36μs in our testbed) have been removed, is only (17μs). While this performance is still far from what is achievable using dedicated hardware solution, we believe that few software solutions reach this level of accuracy. We also emphasize the fact that this performance is achieved using a server that itself suffers from large system noise in timestamping. They are not created in the kernel and as such, suffer from delays and inaccuracy. We anticipate significant performance improvements if IEEE-1588 boundary clocks were used instead of stratum-1 NTP servers.

Next (figure 4), we perform a direct comparison against a GPS synchronised software clock in the same host, using a modified kernel. We observe that the two clocks agree to less than 2μs with a spread of 18μs over the 33 hours. While this experiment is a strong challenge for the TSCclock (the quality of each clock synchronisation source is radically different) the TSCclock performance is comparable to that of the GPS synchronised server. Again, using a IEEE-1588 clock to synchronise the TSCclock instead of an NTP server should definitely improve its performance and reduce its variability. However, a precise investigation of the TSCclock absolute performance at this level is made difficult because it is partially hidden by the host system noise (usually of the order of 20μs). This is a validation issue (the testbed is not yet perfect), not a problem with the TSCclock itself.

Finally, figure 5 gives some insights into the respective absolute performance of the TSCclock and SW-NTP as a function of two parameters. First, the quantity of raw synchronisation information as controlled by the polling period to the server, and second, the distance to the server measured in hops. As the hop count increases the delay distribution moves to higher values, and the path asymmetry also (very likely) increases. For each experiment, the TSCclock and *ntpd* share the same stream of NTP packets. In each plot, the thick black lines show median errors, and the surrounding lines give $[2, 25, 75, 98]$ error percentiles over almost 2 days. *ServerLoc* is a stratum-1 server installed two hops away from the host PC running both clocks with a minimum Round Trip Time (RTT) around 0.38ms. *ServerNear* is a stratum-1 server located 5 hops away with a minimum RTT measured at 0.89ms. Finally, *ServerFar* is a stratum-1 server located 1000km away, 10 hops away (as observed over stable routes) for which we measured a minimum RTT of around 14.2ms.

For each clock we observe the expected qualitative behavior: as the polling period and the distance to the reference clock increase, the performance degrades. In this experiment the TSCclock performs clearly better than SW-NTP. The variability of the synchronisation obtained with the TSCclock is much smaller than that obtained with SW-NTP. Also in each case, the median value of the absolute performance of the TSCclock is closer to the reference given by the DAG card and much more stable as the polling period increases. This illustrates the potential of the algorithm to go well beyond the LAN setting with only a modest performance penalty.

5 Conclusion

So far we compared the TSCclock performance to both SW-NTP and SW-GPS clocks as they are the most widely deployed ones at this time. These early results help us gain understanding the performance of the TSCclock and highlight points of possible improvement. A well-known obstacle to improvement of the TSCclock performance remains the system noise present on NTP servers that are usually simply commodity hardware running *ntpd*. One of the next steps of this work is to take advantage of hardware clocks as specified by the IEEE-1588 and observe the performances of the TSCclock under conditions where noise and asymmetry are reduced. We expect it to perform very well because, although its algorithms are designed to cope with high levels of jitter, they are not 'optimised' for this. Without any need for tuning, they will immediately perform close to optimality should they be presented with a very low jitter environment.

Another obvious comparison would be to compare the performance of the TSCclock with the *PTPd* solution [8]. While having different objectives, we believe the TSCclock would be a robust alternative to *PTPd* in providing accurate software clock on LANs. In the future, we also will investigate the possibility of using the TSCclock as a boundary clock. The promising performance observed when comparing the TSCclock against SW-GPS calls for a modified version of the TSCclock capable of processing GPS input. Without being able to reach the level of accuracy offered by dedicated hardware clock, we believe this solution would offer an inexpensive alternative to network applications requiring accuracy in a $10\mu s$ range, or even below.

References

[1] D. Mills, "Network Time Protocol (Version 3) specification, implementation and analysis", IEFT, Network Working Group, RFC-1305, March 1992, 113 pages, papers in appendix.

[2] A. Pásztor and D. Veitch, "PC Based Precision Timing Without GPS", in *Proceeding of ACM Sigmetrics 2002 Conference on the Measurement and Modeling of Computer Systems*, 15-19 June 2002, pp. 1–10, Del Rey, California.

[3] D. Veitch, S. Babu, and A. Pásztor, "Robust Synchronization of Software Clocks Across the Internet", in *Proc. 2004 ACM SIGCOMM Internet Measurement Conference*, 25-27 October 2004, pp. 219–232, Taormina, Italy.

[4] D. Veitch, "Synchronising Software Clocks on the Internet", in *Winter Meeting of The North American Network Operators' Group (NANOG)*, 8-10 Feb 2004, Miami, http://www.nanog.org/mtg-0402/delay.html.

[5] E. Corell, P. Saxholm, and D. Veitch, "A User Friendly TSC Clock", in *Passive and Active Measurement Conference (PAM2006)*, March 30-31 2006, Adelaide Australia, http://www.pamconf.org/2006/program.html.

[6] J. Ridoux and D. Veitch, "A Methodology for Clock Benchmarking", in *Tridentcom*, May 21-23 2007, Orlando, Florida, USA. IEEE Computer Society.

[7] D. Veitch, J. Ridoux, and S. Babu, "Robust Synchronization of Absolute and Difference Clocks over Networks", *Submitted for publication*, 2007.

[8] K. Correll, N. Barendt, and M. Branicky, "Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol", in *ISPCS*, October 10-12 2005, Zurich, Switzerland. IEEE Computer Society.

[9] "Endace Measurement Systems", `http://www.endace.com/`.