Robust Synchronization of Absolute and Difference Clocks over Networks

Darryl Veitch, *Senior Member, IEEE,* Julien Ridoux, *Member, IEEE,* and Satish Babu Korada

***Abstract*—We present a detailed re-examination of the problem of inexpensive yet accurate clock synchronization for networked devices. Based on an empirically validated, parsimonious abstraction of the CPU oscillator as a timing source, accessible via the TSC register in popular PC architectures, we build on the key observation that the measurement of time differences, and absolute time, requires separate clocks, both at a conceptual level and practically, with distinct algorithmic, robustness, and accuracy characteristics. Combined with round-trip time based filtering of network delays between the host and the remote time server, we define robust algorithms for the synchronization of the absolute and difference *TSCclocks* over a network. We demonstrate the effectiveness of the principles, and algorithms using months of real data collected using multiple servers. We give detailed performance results for a full implementation running live and unsupervised under numerous scenarios, which show very high reliability, and accuracy approaching fundamental limits due to host system noise. Our synchronization algorithms are inherently robust to many factors including packet loss, server outages, route changes, and network congestion.**

***Index Terms*—timing, synchronization, software clock, NTP, GPS, network measurement, round-trip time, TSC.**

I. Motivation

The availability of an accurate, reliable, and high resolution *clock* is fundamental to computer systems. Ongoing synchronization to a time standard is necessary to keep the *offset* of such a clock, that is its departure from the true time, small. A common way to achieve this for networked computers is to discipline the system software clock (SW) through the algorithms associated with the Network Time Protocol (NTP) [1], [2], which allows timestamp information to be exchanged between NTP time server(s) and the client. Algorithms process these timestamps, determine the offset, and deliver rate and offset adjustments to the SW clock.

For many purposes this *SW-NTP* clock solution works well. NTP is designed to provide offset accuracy bounded by the round-trip time (RTT) between the server and the client, and under ideal circumstances offset can be controlled to below 1ms. For more demanding applications however, the performance of the SW-NTP clock is insufficient. Offset errors can be well in excess of RTT's in practice, and more importantly, are susceptible to occasional reset adjustments which can in extreme cases be of the order of 10's or even 100's of milliseconds. In other words, the SW-NTP clock is not reliable enough and lacks robustness. In addition, in the SW-NTP solution the *rate* or frequency of the clock is deliberately varied as a means to adjust offset. This results in erratic rate performance. A smooth and accurate clock rate is a highly desirable feature as it determines the relative accuracy of time *differences*, which are basic to most applications.

As distributed computing and service delivery over networks increase in importance, so will a fundamental constraint to such distributed systems: packet latency, and with it clock synchronization. One application where this is critical now is inexpensive measurement of IP networks, where off the shelf PC's are used to monitor data packets as they pass by the network interface. The drawbacks of the SW-NTP clock, as for example reported in [3], [4], are widely recognised in the network measurement community. They have led many networking researchers to turn to local rather than remote clock synchronization. The Test Traffic Measurement network of RIPE NCC for example [5], consisting of over 100 customised PC's across Europe and elsewhere, uses Global Positioning System (GPS) receivers to locally discipline the standard SW clock, which improves synchronization to around 10μs. Although GPS is no longer an expensive technology as such, the need for roof access to avoid intermittent reception results in long installation delays and costs, making a multi-node efforts such as RIPE NCC's extremely ambitious, and even modest measurement efforts problematic. Radio based alternatives for synchronization rely on the presence of the appropriate network and also imply additional hardware. It is therefore desirable to provide improved network based synchronization with 'GPS-like' reliability, and increased accuracy, using inexpensive PC's with no additional hardware.

In [4] a new clock was proposed which made significant progress towards this aim. It was based on the *TimeStamp Counter* (TSC) register, found in Pentium class PCs and other architectures, which counts CPU cycles. The essence of this *TSCclock* was very simple. The TSC register is used to keep track of time at high resolution, for example 1 nanosecond for a 1 gigahertz processor. Updating this register is a hardware operation, and reading it and storing its value is also fast. Provided that we have an accurate estimate of the true period, p, of a clock cycle, time differences measured in TSC units can readily be converted to time intervals in seconds: $\Delta(t) = \Delta(\text{TSC}) \cdot p$. This simple idea is feasible because CPU oscillators have high *stability*, so the cycle period is, to high precision, constant over quite long time periods, or in other words, accumulated drift is small over these time scales (below 1 part in 10^7). Two methods of remote calibration over a network were given in [4] for measuring p, however

Darryl Veitch and Julien Ridoux are with the ARC Special Centre for Ultra-Broadband Information Networks (CUBIN), an affiliated program of National ICT Australia (NICTA), Dept. of E&E Engineering, University of Melbourne, Australia (Email: {dveitch,jridoux}@unimelb.edu.au).

Satish Babu is a graduate student at École Polytechnique Fédérale de Lausanne, Switzerland (Email: satish.korada@epfl.ch).

neither were robust enough for unsupervised use under real-world conditions. The first aim of this paper is to provide an accurate and highly robust algorithm for p measurement. The result is the *difference TSCclock*, a highly accurate and robust clock for the measurement of time differences below a critical scale (typically around 1000[sec] for PCs). As an example, for typical round-trip times (RTTs), its error is below 0.1μs, even after days of connectivity loss.

The second, and main aim of this paper is to address in detail robust *absolute* synchronization in the context of the TSCclock. Absolute synchronization is a very different, and more difficult problem than that of *rate* synchronization, that is the measurement of p. Here we describe principles, a methodology, and filtering procedures which make reliable absolute synchronization possible, within a client-server paradigm of timestamp exchange with a reference clock across a noisy network. The result is the *absolute TSCclock*, an accurate robust clock for the measurement of absolute time. As an example, if the host has a symmetric path to a nearby server, it can synchronize down to around 30μs or even below, close to the level of host system 'noise'. Under loss of connectivity, the clock will slowly drift but no abrupt errors will be introduced.

Pointing out the need for two separate clocks, which are not just different conceptually but also in terms of synchronization algorithms, robustness, and accuracy, is in itself of considerable importance. The SW-NTP clock is an absolute clock only, and is therefore fundamentally unsuitable for many applications such as the measurement of RTT, delay variation, and code execution time. It deliberately couples the measurement of offset and rate, whereas the TSCclocks succeed in (almost) entirely decoupling them. The TSCclocks considerably raise the bar for the accuracy, and more importantly, the reliability, of synchronization achievable inexpensively across a network.

The TSC is already routinely employed in software clock solutions (not as the primary local source, but to interpolate between the timer interrupts generated by another hardware oscillator). By 'TSCclock' we refer to the overall solution described here, its principles and algorithms, and *not simply to the fact that the TSC is employed*. Indeed the clock definition and algorithms (and implementation) are generic, and could be used with some other hardware counter.

This paper is an enhanced and extended version of [6]. The additions are mainly in section VI-B, which uses an improved validation methodology and several new long traces from new servers to provide a substantially expanded set of performance results, on both old and new hardware, using a new implementation. We include some experiments using stratum-2 NTP servers to complement our core stratum-1 results, and demonstrate the impact of very high host load. For interest, we have also added a simple comparison against SW-NTP. A detailed comparison is beyond scope and will be the subject of another paper. Notations from [6] have been revised and simplified throughout, and the timestamping and path asymmetry discussions have been revisited. For space reasons some details, including on Allan deviation, machine room temperature, and local rates, have been omitted.

II. Preliminaries

In this section we provide background on the infrastructure underlying our clock, its synchronization and characterization.

A. Terminology

A perfect clock, denoted simply by t, runs at a rate of 1 second per second, and has an origin $t = 0$ at some arbitrary instant. A given real clock is imperfect. It reads $C(t)$ at the true instant t and suffers from an error or *offset* $\theta(t)$ given by

$$\theta(t) = C(t) - t \tag{1}$$

at true time t. The *skew* γ corresponds to the difference between the clock's rate and the reference rate of 1. The model which captures this idea in its simplest form we call the *Simple Skew Model* (SKM). It assumes that

$$\textbf{SKM:} \quad \theta(t) = \theta_0 + \gamma\, t. \tag{2}$$

To refine the concept of skew we write

$$\theta(t) = \theta_0 + \gamma\, t + \omega(t), \tag{3}$$

where the 'simple skew' γ is just the coefficient of the deterministic linear part, $\omega(t)$ being a detrended remainder obeying $\omega(0) = 0$, which encapsulates non-linear deviations.

The *oscillator stability* [7] partially characterizes $\omega(t)$ via the family, indexed by timescale τ, of relative offset errors:

$$y_\tau(t) = \frac{\theta(t+\tau) - \theta(t)}{\tau} = \gamma + \frac{\omega(t+\tau) - \omega(t)}{\tau}. \tag{4}$$

In other words, $y_\tau(t)$ is the average skew at time t when measured over time scale τ, and consists of the mean skew γ plus non-linear variations which impact at time scale τ.

Table I translates skew values (expressed as *Parts Per Million* (PPM) since skew is dimensionless) into absolute error over key time intervals: $\Delta(\text{offset}) = \Delta(t) \cdot (\text{average skew})$. The typical skew of CPU oscillators from nominal rate is around 50PPM [7].

B. The TSCclock

We propose a clock based on the TSC register which counts CPU cycles. Denote the register contents at time t by $\text{TSC}(t)$, and set $\text{TSC}_0 = \text{TSC}(0)$. The construction of an absolute clock from the counter is based on the intuition of the simple skew model, where the oscillator period p is constant, implying that $t = (\text{TSC}(t) - \text{TSC}_0)p$. In practice we must obtain estimates,

Significance of Time Interval	Interval Duration	Skew [PPM] 0.02	Skew [PPM] $\gamma^* = 0.1$
Target RTT to NTP server	1ms	0.02ns	**0.1ns**
Typical Internet RTT	100ms	2ns	10ns
Standard unit	1s	20ns	0.1μs
Local SKM validity	τ^*=1000s	20μs	**0.1ms**
1 Daily cycle	86400s	1.7ms	8.6ms
1 Weekly cycle	604800s	12.1ms	60.5ms

TABLE I
Absolute errors at key error rates and time intervals.

$\hat{p}$ of p, and $\widehat{\text{TSC}}_0$ of TSC_0. The definition of a simple TSC based (absolute) clock $C_u(t)$ is therefore

$$\textbf{SKM:} \quad C_u(t) = (\text{TSC}(t) - \widehat{\text{TSC}}_0)\hat{p} = \text{TSC}(t)\hat{p} + K, \quad (5)$$

where the constant $K = -\widehat{\text{TSC}}_0\hat{p}$ tries to align the origins of $C_u(t)$ and t, but with some error. It is easy to show that the errors $p_\epsilon = \hat{p} - p$ and $\text{TSC}_\epsilon = \widehat{\text{TSC}}_0 - \text{TSC}_0$ lead to $\theta(t) = p_\epsilon/p \cdot t - \hat{p}\,\text{TSC}_\epsilon$, which, comparing to (2), identifies $\gamma = p_\epsilon/p = \hat{p}/p - 1$ and $\theta_0 = C_u(0) = \text{TSC}_0\,\hat{p} + K$.

The SKM model does not hold over all timescales, so the above estimates must be taken as time varying. A key consequence is that the variation of offset over time is no longer a simple, known function of γ, and so must be measured independently, that is the clock drift must be tracked. In practice therefore we must correct the *uncorrected* clock $C_u(t)$. In fact two variants, depending on whether time differences (valid up to SKM timescales), or absolute time, are needed:

difference: $C_d(t) = \text{TSC}(t)\hat{p}(t)$

absolute: $C_a(t) = \text{TSC}(t)\hat{p}(t) + K - \hat{\theta}(t) = C_u(t) - \hat{\theta}(t)$,

where $\hat{p}(t)$ is the current period estimate, and $\hat{\theta}(t)$ is the current estimate of the offset of the uncorrected clock $C_u(t)$, which we correct for to obtain $C_a(t)$. Only by defining two clocks in this way can we provide an absolute clock without negating the smooth rate of the underlying hardware, which is the basis of the extremely high accuracy of the difference clock. The absolute clock should only be used for applications which truly require it, because the estimation of $\hat{\theta}$ is inherently challenging. On the other hand the difference clock does not involve $\hat{\theta}$, and so can be used to measure time differences very accurately [4] provided drift can be ignored, which is the case for time intervals which are small compared to the critical 'SKM scale' τ^*, defined below. As $\tau^* \approx 1000$ [sec], this includes most cases of importance to traffic measurement. Above this scale clock drift is significant, and the time difference will be more accurately measured using $C_a(t)$.

C. Timestamping

Even a perfect clock is of little use if one is unable to read at the right time. Dealing with this *timestamping* issue is application dependent. Here, for the purpose of remote synchronization of the TSCclock, the application is the timestamping of arriving and departing NTP packets.

In [4] (see also [8]), timestamping was achieved in the Linux 2.4 kernel by exploiting the existing API. Here we modified the existing Berkely Packet Filter tapping mechanism under BSD 2.2.8, 5.3 and 6.1 kernels, which results in driver based timestamping for incoming packets, and a kernel based one for outgoing ones. The timestamping locations were carefully chosen to achieve two aims: closeness to the hardware in order to reduce system noise in both outgoing and incoming directions, and 'causality-compliance', namely that timestamps on the sending side are taken before the packets are sent. Compliant timestamping ensures that system noise can only **increase** the round-trip time relative to the timestamping events compared to the true round-trip time. It therefore appears as a positive host delay on top of network delay, which can be dealt with by the filtering mechanisms described below. The use of user-level timestamping breaks compliance, leading to serious problems, which can however be circumvented [9].

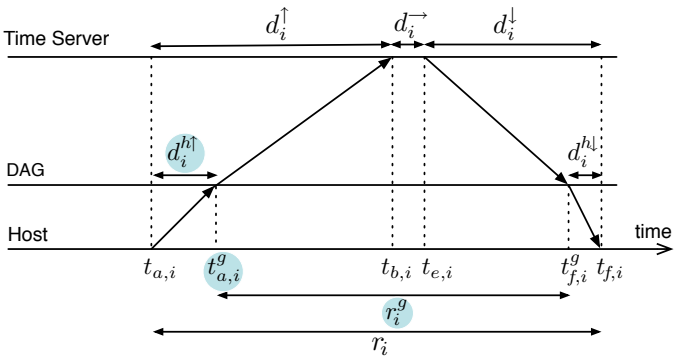

Fig. 1. Timeline of the ith host-server exchange including forward path delay ($d_i^{\uparrow}$), server delay ($d_i^{\rightarrow}$), backward path delay ($d_i^{\downarrow}$) and RTT ($r_i = d_i^{\uparrow} + d_i^{\rightarrow} + d_i^{\downarrow}$). The shaded quantities were not available for the 2003 traces.

D. NTP Time Servers

Network Time Protocol (NTP) servers are networked computers running the *ntpd* daemon. We will be principally concerned with *stratum-1* servers, whose clocks are synchronized by a local reference time source. *ServerLoc* is in our laboratory just outside the local network where the hosts reside. *ServerInt* is also located at the University of Melbourne, but is on a distinct network in a different building and uses a different GPS receiver. Finally, *ServerExt* is located in a distant city. The distances between the host and the servers are given in table II in terms of physical distance, the minimum RTT of NTP packets over at least a week, and in the number of IP hops as reported by the *traceroute* utility. We also give the path asymmetry A, which as discussed in detail in section IV-B is the difference of the minimum one-way delays to and from the server. The servers of table II were used in 2003 to collect the traces used in Sections III, IV, V and VI-A. The new traces from 2006-7 use different servers described in section VI-B. Hosts wishing to synchronize their system clock run an application which communicates with a NTP server via NTP packets. The client-server exchange works as follows. The ith NTP packet is generated in the host. Before being sent, the timestamp $T_{a,i}$ is generated by the SW clock and is placed in the packet payload. Upon arrival at the server, the timestamp $T_{b,i}$ is made by the server clock and inserted into the payload. The server then immediately sends the packet back to the host, adding a new departure timestamp $T_{e,i}$, and the host timestamps its return as $T_{f,i}$. The four timestamps $\{T_{a,i}, T_{b,i}, T_{e,i}, T_{f,i}\}$ are the raw data from the ith exchange from which the host clock must be synchronized. None of these are perfect however due to clock and timestamping

Server	Reference	Distance	r (min RTT)	Hops	A
ServerLoc	GPS	3 m	0.38 ms	2	$\approx 50\mu$s
ServerInt	GPS	300 m	0.89 ms	5	$\approx 50\mu$s
ServerExt	Atomic	1000 km	14.2 ms	≈ 10	$\approx 500\mu$s

TABLE II
CHARACTERISTICS OF THE STRATUM-1 NTP SERVERS USED IN 2003.

limitations. The *actual* times of the corresponding events we denote by $\{t_{a,i}, t_{b,i}, t_{e,i}, t_{f,i}\}$, as shown in Figure 1.

The TSCclock does **not** use the usual timestamps made by the SW-NTP clock at the host, but instead takes separate raw TSC timestamps. We denote these by $T_{a,i}$, $T_{f,i}$ as above even though they are in 'TSC units', rather than seconds. Being a stratum-1 NTP server, the server's clock should be synchronized, and so we could expect that $T_{b,i} = t_{b,i}$ and $T_{e,i} = t_{e,i}$. However, as servers are often just PC's running *ntpd* to synchronize the SW to GPS, this may not be the case.

In the implementation described here, we rely on the normal flow of NTP packets between host and server, to minimize the disruption to normal system operations. We use a polling rate of 16 seconds, which is higher than the usual default, but which provides a detailed set of data base from which to examine both the underlying TSC oscillator and the TSCclock performance. Other values are considered later.

We assume that an effort has been made to locate a nearby stratum-1 server, such as *ServerInt*, which has a RTT of the order of only 1ms, but is not on the local network. It also has the advantage (see section IV-B) of having, in terms of network elements, a verifiably symmetric route. We believe that such a server can be readily found in institutions with significant networking infrastructure. We stress however that the presence of such an 'optimal' server is not required for very good results in most cases (see section VI-B).

E. Reference Timing

Validation of timing methods would not be possible without a reliable timing reference. We used DAG3.2e (2003 data sets) and DAG3.7GP (2006-7) measurement cards, designed for high performance passive monitoring of wired Ethernet with timestamping accuracy around 200ns ([10], [11]). The cards were synchronized to a Trimble Acutime 2000 GPS receiver mounted on the roof of the laboratory.

Ideally the DAG and TSCclock would timestamp the same events, however as the DAG monitors packets via a passive tap on the Ethernet cable **outside** the hosts, as shown in Figure 1 the DAG event times obey $t^g_{a,i} > t_{a,i}$ and $t^g_{f,i} < t_{f,i}$ (as DAG timestamps the first bit rather than the last, a correction of $90 * 8/100 = 7.2\mu$s is included in $t^g_{a,i}, t^g_{f,i}$ as 100Mbps interface cards were used in all hosts). The resulting host-DAG and DAG-host delays $d_i^{h\uparrow}$, $d_i^{h\downarrow} > 0$ are primarily due to the processing time of the network card and the host scheduling and interrupt latency. We call these collectively *system noise*.

System noise has a constant and a variable component and depends on the host and its operating system. By removing the network side RTT $r^g_i = t^g_{f,i} - t^g_{a,i}$, as seen by the DAG, from the total RTT r_i, we isolate the total system noise or $r^h_i = d_i^{h\uparrow} + d_i^{h\downarrow} = r_i - r^g_i$. This 'host RTT' can be reliably measured. Typical characterising values are a minimum value of $r^h = \min_i(r^h_i) = 150\mu$s and an inter-quartile range of 10μs. The latter figure places a lower limit to our validation methodology for clock offset (but not rate) errors, whereas the impact of the former depends on other factors linked to path asymmetry. For the experiments conducted in 2003 the DAG timestamps $t^g_{a,i}$ for outgoing packets were not available, so that r^g_i and r^h_i could not be measured exactly.

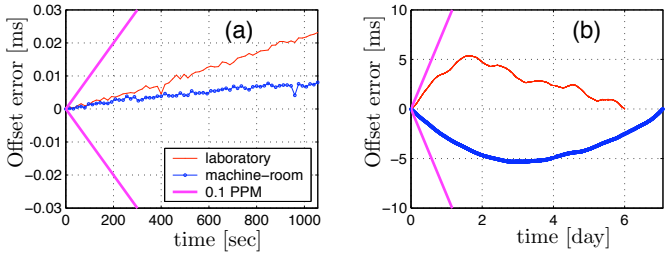

Fig. 2. The offset error $\theta(t)$ of $C_u(t)$, in two temperature environments, falls within the skew cone defined by $\pm\gamma = 0.1$PPM. (b) over 1 week. (a) a zoom over the first 1000 seconds.

Occasionally spikes (isolated outliers) occur in system noise. These can be reliably detected and filtered out, producing 'corrected timestamps'. This is important both for reliable validation and Allan deviation measurement.

III. Timestamp Data Characterization

In this section we study the raw timestamp data and derive models of it on which the TSCclock algorithm is built.

A. The Clock

We examine the TSCclock offset of a 600Mhz host, *green*, in two different temperature environments, *laboratory*: an open plan area in a building which was not airconditioned, and *machine-room*: a closed temperature controlled environment.

It is convenient to examine the drift of the TSC oscillator via that of the uncorrected TSCclock $C_u(t) = \text{TSC}(t)\hat{p} + K$, however to do so we must first supply a value of $\hat{p}$. In Figure 2 we use $\hat{p} = \bar{p} = 1.82263812 * 10^{-9}$ (548.65527 Mhz) for measurements made in the laboratory, and $\bar{p} = 1.82263832 * 10^{-9}$ (548.65521 Mhz) in the machine-room, and then estimate the offset via $\hat{\theta}(t_{f,i}) = T_{f,i} * \bar{p} - T^g_{f,i}$ for each. These $\bar{p}$ estimates are obtained by a simple average using the first and last packets.

The above uncorrected, SKM based clocks, allow the non-linear drift of the TSC to be inspected. From the right plot in Figure 2 it is clear that the SKM model fails over day timescales, as the offset error is far from linear, although the variations fall within the narrow cone emanating from the origin defined by $\gamma = \pm 0.1$PPM. In the left plot however we see that over smaller time scales the offset error grows approximately linearly with time. We found these observations to hold for all traces collected over many months in 2003. In [4] the same result was reported for a host in an airconditioned (but not temperature controlled) office environment over a continuous 100 day period. In 2006-7, using the same hardware, then over 6 years old, values above 0.1PPM but below 1PPM were sometimes encountered.

The above discussion is only one illustration. Depending on the timescale and $\hat{p}$ value chosen, $\theta(t)$ can take on very different appearances. To examine offset over all scales simultaneously, and to avoid the need for a prior rate estimation, we return to the concept of oscillator stability (4). A particular estimator of the variance of $y_\tau(t)$, known as the *Allan variance* (essentially a Haar wavelet spectral analysis [12]), calculated over a range of τ values, is a traditional characterization

of oscillator stability [7]. We term its square root the *Allan deviation*, and interpret it as the typical size of variations of time scale dependent clock rate. A study over a range of timescales is essential as the source and nature of timing errors vary according to the measurement interval. At very small timescales, γ will not be readily visible in $y_\tau(t)$ as the 'rate' error will essentially correspond to system noise affecting timestamping. At intermediate timescales γ may seem well defined and constant with some measurement noise, as in the left plot in Figure 2. At large scale where daily and weekly cycles enter, the issue is not noise in estimates of γ but rather variations in γ itself.

Four Allan deviation plots are given in Figure 3, for traces taken under different conditions ranging from 1 to 3 weeks in length. One is when the host was in the laboratory, and uses *ServerInt*. The others are from the machine room, using each of the 3 servers. Corrected $T_{f,i}$ timestamps were used here, as otherwise the timestamping noise outliers add considerable spurious variation at small scales.

Over small scales the plots show a consistent $1/\tau$ decrease, consistent with the results of [4]. This is exactly what we would expect if the SKM were true with $\omega(t)$ in (3) representing white system noise, and the plots agree as the hardware, operating system, and timestamping solution are the same in each case, and the dominant noises arise from them. The plots diverge, and all rise, at larger scales as new sources of variation, such as diurnal temperature variations, enter in. They begin to flatten as major new sources of variation cease at the weekly timescale, whilst always remaining below the horizontal line marking 0.1 PPM.

In the machine-room the environmental control bounds temperature variations within a 2^oC band. We therefore expect that the laboratory data would be more variable, and therefore that the corresponding curve will lie above each of those from the machine-room. This is indeed the case at large scales, but not always at intermediate scales, due to the presence of a low amplitude (≈ 0.05PPM) but distinct oscillatory noise component of variable period between 100 to 200 minutes (clearly visible in Figure 8). This was confirmed as being driven by the airconditioning cycle through checking with a digital temperature logger.

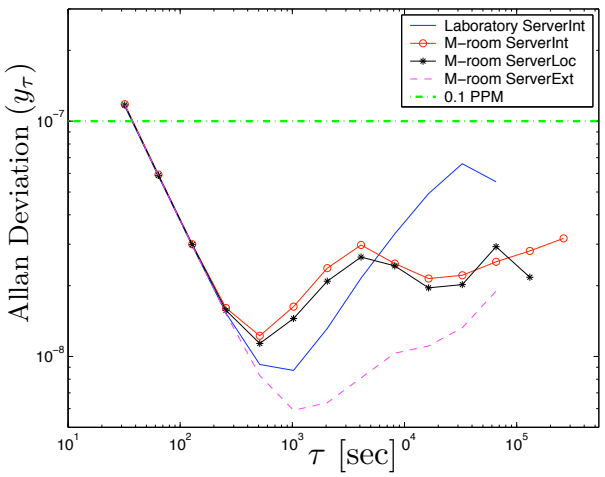

Fig. 3. Allan variation plots. From small scales to around $\tau = 1000$ seconds, the SKM applies, and rate estimates are meaningful down to 0.01PPM.

In conclusion, in three different temperature environments the SKM model holds over timescales up to 1000 seconds. Henceforth we use *SKM scale*, or τ^*, to refer to this value. Below the SKM scale local rate is meaningful but may only be measured to an accuracy given by the value of the Allan deviation at that scale. This is bounded by the minimum value attained at $\tau = \tau^*$, which here is of the order of $\gamma^*_\epsilon = 0.01$PPM. It is not meaningful to speak of rate errors smaller than this, as the validity of the SKM model itself cannot be verified to this level of precision. Over larger timescales the model fails but the rate error remains below the *rate error bound* γ^* $=0.1$ PPM. Indeed, to within this level of accuracy we can say that the SKM model holds over all time scales. These measurements are consistent with the results of [7] stating that the clock stability of commercial PCs is typically of the order of 0.1 PPM. The oscillator metrics above appear as parameters in the TSCclock algorithm, which can therefore be used, for example, for less stable oscillator classes by making the appropriate calibration.

To characterise rate beyond τ^*, one cannot hope to measure an expected or stationary value, as it does not exist. We do **not** attempt to measure a (meaningless) 'long term rate' as such in this paper, however we **do** make use of estimates made over large time intervals, corresponding to an average of *meaningful local rates*, as a means of reducing errors due to timestamping and network congestion. Such average rates may be used as surrogates for local rates, with an error which is bounded by 0.1 PPM. We return to this topic in section V-C where we discuss local rates in more detail.

B. Network and Server Delay

Following Figure 1, we decompose packet i's journey as:

$$\begin{aligned} \text{Forward network delay} &: \ d_i^{\uparrow} = t_{b,i} - t_{a,i} \\ \text{Server delay} &: \ d_i^{\rightarrow} = t_{e,i} - t_{b,i} \\ \text{Backward network delay} &: \ d_i^{\downarrow} = t_{f,i} - t_{e,i} \\ \text{Round Trip Time} &: \ r_i = t_{f,i} - t_{a,i} = d_i^{\uparrow} + d_i^{\rightarrow} + d_i^{\downarrow}. \end{aligned}$$

Figure 4 gives representative examples of 1000 successive values of the backward network delay and server delay for the host in the machine-room, using *ServerLoc*, calculated as $d_i^{\downarrow}(t_{e,i}) = T^g_{f,i} - T_{e,i}$ and $d_i^{\rightarrow}(t_{e,i}) = T_{e,i} - T_{b,i}$ respectively.

These time series appear stationary, with a marginal distribution consistent with a deterministic minimum value plus a positive random component. These observations make physical sense. The minimum in network delay corresponds to propagation delay plus minimum system noise, and the random component to queueing in network switching elements and the operating system. Not unexpectedly, the latter are very small for such a short route, but can take 10's of milliseconds during periods of congestion, For the server, there will be a minimum processing time and a variable time due to timestamping issues both in the μs range, and rare delays due to scheduling in the

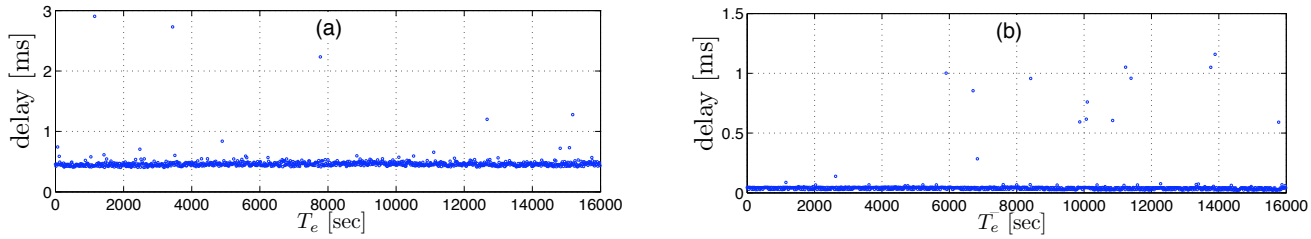

Fig. 4. Example time series: (a) backward network delay $d_i^{\downarrow}$, (b) server delay $d_i^{\rightarrow}$. They are well modelled by a constant + positive noise.

millisecond range. We formalise these observations in

$$\begin{aligned} \text{Forward network delay} &: \; d_i^{\uparrow} = d^{\uparrow} + q_i^{\uparrow} \\ \text{Server Delay} &: \; d_i^{\rightarrow} = d^{\rightarrow} + q_i^{\rightarrow} \\ \text{Backward network delay} &: \; d_i^{\downarrow} = d^{\downarrow} + q_i^{\downarrow} \\ \text{Round Trip Time} &: \; r_i = r + (q_i^{\uparrow} + q_i^{\rightarrow} + q_i^{\downarrow}), \end{aligned} \tag{6}$$

where $d^{\uparrow}$, $d^{\rightarrow}$, and $d^{\downarrow}$ are the respective minima and $q_i^{\uparrow}$, $q_i^{\rightarrow}$ and $q_i^{\downarrow}$ are the positive variable components. The minimum RTT is therefore $r = d^{\uparrow} + d^{\rightarrow} + d^{\downarrow}$. These simple models provide the basic conceptional framework for what follows.

IV. Synchronization: naive SKM Clocks

In this section we examine simple 'naive' synchronization ideas based on the SKM, and detail their weaknesses. In section V we show how they can be overcome using the TSCclock algorithms. We use the first day of the same 7 day machine-room data set (July 4–10) used previously.

A. Rate Synchronization

We wish to exploit the relation $\Delta(t) = \Delta(\text{TSC}) * p$ to measure p. More precisely, assuming the SKM the following relation holds for the forward path:

$$p = \frac{t_{b,i} - t_{b,j} - (q_i^{\uparrow} - q_j^{\uparrow})}{\text{TSC}(t_{a,i}) - \text{TSC}(t_{a,j})}$$

where $i > j$. This inspires the naive estimate

$$\hat{p}_{i,j}^{\uparrow} = \frac{T_{b,i} - T_{b,j}}{T_{a,i} - T_{a,j}} \tag{7}$$

which suffers from the neglect of the queueing terms and the presence of timestamping errors. An analogous expression provides an independent estimate $\hat{p}_{i,j}^{\downarrow}$ from the backward path. In practice we average these two to form our final estimate: $\hat{p}_{i,j} = (\hat{p}_{i,j}^{\uparrow} + \hat{p}_{i,j}^{\downarrow})/2$.

In Figure 5 backward estimates normalised as $(\hat{p}_{i,j}^{\downarrow} - \bar{p})/\bar{p}$ (where $\bar{p}$ denotes the 'detrending' estimates from section III-A) are given for all packets collected. The i-th estimate compares the i-th packet against the first ($j = 1$), and is plotted against the timestamp $T_{e,i}$ of its departure from the server. Thus $\Delta(\text{TSC}) = T_{a,i} - T_{a,j}$ steadily increases as more packets are collected. Superimposed are the corresponding reference rate values, calculated as $\hat{p}_g = (T_{f,i}^g - T_{f,j}^g)/(T_{f,i} - T_{f,j})$ which show some timestamping noise ($T_{f,i}$ is not corrected here), but are not corrupted by network delay. We immediately see that the bulk of the estimates very quickly fall within 0.1 PPM of the reference curve, as the size of errors due to both network delay and timestamping noise are damped at rate $1/\Delta(t)$. The estimates from packets which experienced high network delay can nonetheless be very poor. Table I tells us that even when measured over a timescale of a day, the bound of 0.1 PPM will be broken when network queueing delay exceeds only 8.6 ms.

If the SKM held exactly, these errors would eventually be damped as much as desired, however, this is not the case. We wish $\Delta(t)$ to grow large, so that the estimates will become increasingly immune to both network delay and timestamping errors. However, we cannot let it grow without bound, as changes in the rate would then be masked. For example, there is always the possibility that the local environment will change, and ultimately, the CPU oscillator is also subject to aging. Thus some kind of windowing must be employed which enables the past to be forgotten, which limits the degree of error damping available from a large $\Delta(t)$. The conclusion is that the naive estimates are unreliable, as their errors, although *likely* to be small, can not be controlled or bounded.

B. Offset Synchronization

We wish to exploit the fact that the SKM holds over small timescales to simplify the measurement of $\theta(t)$. Since we can assume that $\gamma < \gamma^* = 0.1$ PPM, the increase in offset error over a host-server round-trip time of 1ms is under 0.1ns (see table I). Even if the RTT was huge, such as 1 second, the error increase would be under 0.1μs, which is well below timestamping noise.

Two important observations follow from the above. First, at RTT timescales we can assume that rate is constant and therefore use a 'global' estimate $\bar{p}$ measured over a large $\Delta(t)$ to convert TSC timestamps to time and thereby calculate offset. We do not have to try to calculate a local estimate, which is far more complex. Second, offset error accumulates so slowly that we can associate to each packet i a single constant value θ_i. From packet i we have two relations

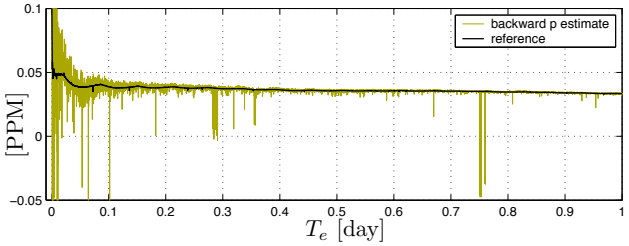

Fig. 5. Naive per-packet rate estimates compared with DAG reference measurements as a function of $\Delta(\text{TSC})$.

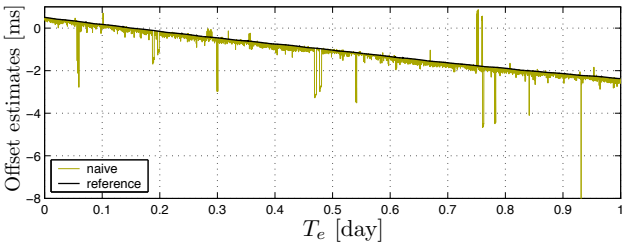

Fig. 6. Naive per-packet offset estimates θ_i compared to DAG reference.

involving θ_i: $\theta_i = C_u(t_{a,i}) - t_{a,i} = C_u(t_{a,i}) - (t_{b,i} - d_i^{\uparrow})$, and $\theta_i = C_u(t_{f,i}) - (t_{e,i} + d_i^{\downarrow})$, from which θ_i cannot be recovered as the one-way delays cannot be independently measured. If we add these:

$$\theta_i = \frac{1}{2}(C_u(t_{a,i}) + C_u(t_{f,i})) - \frac{1}{2}(t_{b,i} + t_{e,i}) + \frac{1}{2}A + \frac{1}{2}(q_i^{\uparrow} - q_i^{\downarrow}), \tag{8}$$

the problem remains that the *path asymmetry*,

$$\text{Path Asymmetry:} \quad A = d^{\uparrow} - d^{\downarrow} \tag{9}$$

is not measurable. There is in fact a fundamental ambiguity here, *the same timestamp observations are consistent with many different* (θ_i, A) *combinations*, given by $\theta_i + c/2$ and $A + c$ for a constant c scanning over a continuous range. Happily, this range is finite because the 'causality' bound $A \in (-(r - d^{\rightarrow}), (r - d^{\rightarrow})) \subset (-r, r)$ holds, i.e. we require the packet events at the server to occur between those at the host. Note that r and $d^{\rightarrow}$ *can* be measured as they each are time differences measured by a single clock under conditions where drift may be neglected.

In the absence of independent knowledge of A, a natural naive estimate based on (8) is

$$\hat{\theta}_i = \frac{1}{2}(C_u(t_{a,i}) + C_u(t_{f,i})) - \frac{1}{2}(T_{b,i} + T_{e,i}), \tag{10}$$

which implicitly assumes that $A = 0$, and is equivalent to aligning the midpoints $(t_{b,i} + t_{e,i})/2$ and $(C_u(t_{a,i}) + C_u(t_{f,i}))/2$ of the server and host event times respectively. In Figure 6 estimates obeying (10) are shown, along with reference values calculated as in section III-A. Errors due to network delay are readily apparent, but are more significant than in the naive rate estimate case because they are not damped by a large $\Delta(t)$ baseline.

The value of A places a hard limit on the accuracy of offset measurement. The choice of server is therefore very important. A nearby server will have a smaller RTT, and therefore a tighter bound. More importantly however, a nearby server is likely to have a path which is symmetric or close to it, which would result in $A \ll r$. This is in fact the case for *ServerLoc* and *ServerInt*, which we measured (see table II) to be of the order of $50\,\mu$s. Estimating A however is non-trivial. When only incoming DAG timestamps were available (2003 data), we used $A = d^{\uparrow} - d^{\downarrow} = r - d^{\rightarrow} - 2d^{\downarrow}$ which in terms of available timestamps reduces to $\hat{A}_i = (T_{f,i} - T_{a,i})\hat{p} - 2T^g_{f,i} + T_{b,i} + T_{e,i}$, and obtained estimates based on the packet i which minimizes r_i (see [4] for more details). In section VI we describe an improved method using bidirectional DAG monitoring.

V. Synchronization: the TSCclocks

Here we define the core components of the TSCclock algorithms. They differ in two key ways from the naive ones of the previous section: they deal with drift, and they are robust. The central obstacle to achieving these is the successful filtering of network and host delays. It is intuitively clear from Figures 5 and 6 that the packets carrying large delays can be detected, and so dealt with. The TSCclock algorithms succeed in doing this reliably even when delays are small.

A. Approach to Filtering

We need to measure the degree to which, for each packet i, the available timestamps are affected by network queueing and other factors. To do so we work with the round-trip time series $\{r_i\}$, which has a number of important intrinsic advantages over the one-way delays, $\{d_i^{\uparrow}\}$ and $\{d_i^{\downarrow}\}$.

As discussed above, since $T_{a,i}$, $T_{f,i}$, are measured by the same clock, and since round-trip times are very small, neither the unknown $\theta(t)$ nor local rates are needed to accurately measure r_i. The same is true for determining the *quality* of r_i, only a reasonable estimate such as an average $\bar{p}$ is required. This creates a near complete decoupling of the underlying basis of filtering from the estimation tasks, thus avoiding the possibility of undesirable feedback dynamics.

The *point error* of a packet is taken to be simply $r_i - r$. The minimum can be effectively estimated by $\hat{r}(t) = \min_{i=1}^{\lfloor t \rfloor} r_i$, leading to an estimated error $E_i = r_i - \hat{r}(t)$ which is highly robust to packet loss. Error will be calibrated in units of the maximum timestamping latency δ at the host (we use $\delta = 15\mu$s).

Whereas round-trip times can effectively ignore drift, one-way delays are measured by different machines, and must therefore be calculated using absolute clocks. As a result, the uncorrected one-way delay $T_{b,i} - C_u(t_{a,i})$ inherits the drift of $C_u(t)$ (recall Figure 2), greatly complicating assessments of quality. On the other hand RTT based filtering has its own key disadvantage. Consider that with independent symmetric paths, if the probability that one-way quality exceeds a given level is q, and q' for server delay, then the corresponding probability drops below $q'q^2$ for the RTT, which can be much smaller than q under congested conditions. Thus quality packets are rarer when judged by the RTT alone, making accurate estimation more challenging.

B. Rate Synchronization

To bound the error on the estimate $\hat{p}(t)$, we use Equation (7) but restrict ourselves to packets with bounded point error. The base algorithm is simple. To initialise, set j and i to be the first and second packets with point errors below some threshold E^*. Equation (7) then defines the first value of $\hat{p}(t)$ which we assign to $t = t_{f,i}$. This estimate holds for $t \geq t_{f,i}$ up until i is updated at the next accepted packet, and so on. An estimate of the error of the current estimate is $(E_i + E_j)/((T_{f,i} - T_{f,j})\bar{p})$ and should be bounded by $2E^*/((T_{f,i} - T_{f,j})\bar{p})$. As before the above procedure is independently applied to both the forward and backward paths, and the results averaged.

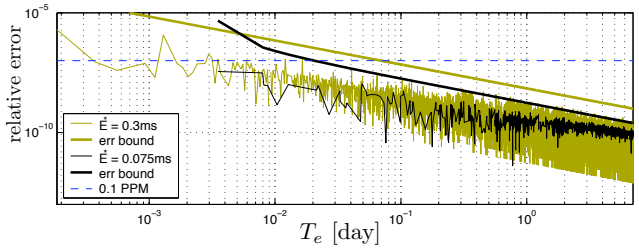

Fig. 7. Relative error in $\bar{p}$ estimates for $E^* = [20, 5] \cdot \delta = [0.3, 0.075]$ ms. Errors fall below $\gamma^* = 0.1$ PPM and remain there.

This scheme is inherently robust, since even if many packets are rejected, error reduction is guaranteed through the growing $\Delta(t) = T_{f,i} - T_{f,j}$, without any need for complex filtering. Even if connectivity to the server were lost completely, the current value of $\hat{p}$ remains entirely valid for filtering, allowing estimation to recommence at any time free of 'warm-up dynamics'.

Figure 7 plots the relative error of the resulting estimates with respect to the corresponding DAG reference rates for those i selected. Two sets of results are given, for $E^* = 20\delta$ and 5δ (resulting in 72% and 3.9% of packets being selected respectively), to show the insensitivity of the scheme to E^*. In each case errors rapidly fall below the desired bound of $\gamma^* = 0.1$ PPM and do not return, in contrast to Figure 5 based on the same raw data. The solid lines give expected upper bounds on the error based on $2E^*/(T^g_{f,i} - T^g_{f,j})$. To put this performance into context, note that for the measurement of time differences over a few seconds and below, which is relevant for inter-arrival times, round-trip times, and also delay variation, the estimate $\hat{p}$ above gives an accuracy better than 1μs, an order of magnitude better than a GPS synchronized absolute software clock, after only a few minutes of calibration.

To ensure that any unexpected failures of the estimation procedure cannot force the rate estimates to contradict the known physical behaviour of the hardware, before a candidate update is accepted it is sanity checked as follows: if the relative difference between two successive rate estimates exceeds some small multiple of the rate bound (we used $3\gamma^*$), then the update will be blocked, and a warning logged (a similar rate bound assumption was used in [13]). This guarantees that the estimate cannot vary wildly no matter what data it receives. One situation where this is needed is when the server timestamps themselves are in error. We have observed many instances of this, and give an example from the data presented here in the next section.

C. Local Rate Estimation

It is important to understand that the estimate $\hat{p}$ above is really that of the average rate $\bar{p}$ over a large $\Delta(t) \gg \tau^*$ window, and is thus an average of many different local or 'true' rates in the sense of the SKM. From Figure 3, true local rates can be meaningfully defined down to accuracies of $\gamma^*_\epsilon = 0.01$ PPM, over scales below τ^*. However, there is no need for local rate estimates in order to obtain $\bar{p}$, and $\bar{p}$ is sufficient to support filtering and both the difference and absolute clocks. This is a huge advantage since the estimation of local rates is much more difficult as there are only a small number of timestamps available with which to 'differentiate' a non-stationary drift. However, there are two important reasons why the measurement of local rates is worthwhile: (i) they extend the scales over which the difference clock $C_d(t)$ can be used to measure time differences, and (ii) to optimize the performance of $\hat{\theta}(t)$ and hence that of $C_a(t)$.

We denote our local period estimates by $p_l(t)$, measured over a timescale τ_l. Ideally $\tau_l < \tau^*$, however larger values are typically needed to control estimation variance. The algorithm calculates a local value for each packet k over a window of effective width τ_l. Unlike for $\bar{p}$ where packets were selected based on a fixed quality, here it is essential to maintain the timescale of the estimate fixed. The actual window is therefore divided into near, central, and far subwindows of width τ_l/W, $\tau_l(W-2)/W$, and $2\tau_l/W$ respectively. In each of the near (index i) and far (index j) windows, the packets with the lowest point errors are selected, and used in (7) to calculate a candidate estimate $\hat{p}_l(t_{f,k})$. As before, a bound on the error of the estimate is calculated as $(E_i + E_j)/((T_{f,i} - T_{f,j})\bar{p})$. If it lies under a target quality value γ_l (which we choose to be $\gamma_l = 0.05$ PPM > 0.01 PPM to allow for estimation error) we accept the estimate, else we are conservative and set $\hat{p}_l(t_{f,k}) = \hat{p}_l(t_{f,k-1})$. We then set $\hat{p}_l(t) = \hat{p}_l(t_{f,k})$, where packet k is the most recent packet arriving before time t. We apply a sanity check for candidates in a similar way to $\hat{p}$. If it fails, the previous trusted value is used.

As it is important that the estimate be local to the packet k, W should be chosen small. On the other hand W should be large enough so that packets of reasonable quality will lie within it. By selecting the best candidates in the near and far windows, we guarantee that there is an estimate for each k. Good quality is designed into the scheme through the width of the central window. Robustness to outliers is provided by the monitoring of the expected quality of the candidate estimate, and the sanity checking. Consequently, we found that the results are insensitive to W (we use $W = 30$).

The algorithm closely tracks the corresponding reference rate values made over the same time-scale. Using the same data as in Figure 7, with $\gamma_l = 0.05$ PPM, $\tau_l = 5\tau^*$ and $W = 30$, over 99% of the relative discrepancies from the reference were contained within 0.023 PPM. The outliers were due mainly to errors in the reference rates, not instabilities in the estimation algorithm. Only 0.6% of values were rejected by the quality threshold, and the sanity check was not triggered.

D. Offset Synchronization

Our aim is to estimate $\theta(t)$ for arbitrary t, using the naive $\hat{\theta}_i$ estimates from *past* packets as a basis. Note that for many applications, post processing of data would allow both future and past values to be used, which improves performance, particularly following long periods of congestion or sequential packet loss. We focus on causal filtering as required by a general purpose system clock operating on-line.

In this section we use data collected continuously in the machine-room over the last 3 weeks of September 2003. The host was connected to *ServerInt*. We also present comparative

results from a week long trace using *ServerLoc* and a trace 2.7 weeks long using *ServerExt*.

When estimating $\hat{p}$, large $\Delta(t)$ values were an asset. In contrast, since $\theta(t)$ must be tracked, large time intervals between quality packets would imply that the accepted $\hat{\theta}_i$ would be out of date. This fundamental difference suggests a paradigm of using estimates derived for *each* packet. Our approach consists of four stages: (i) determining a total per-packet error E_i^T which combines point error and packet age, (ii) assigning a weight w_i based on the total error, (iii) combining the weighted point estimates to form $\hat{\theta}(t)$, and (iv) a sanity check to ensure that $\hat{\theta}(t)$ will not evolve faster than the known hardware performance for any reason.

(i) Based on the last packet i arriving before time t, the simplest approach is simply to set $\hat{\theta}(t) = \hat{\theta}_i$. The magnitude of the resulting error can be estimated by inflating the point error by a bound on its growth over time: $E_i^T = E_i + \gamma^*(C_d(t) - C_d(T_{f,i}))$. This may be overly pessimistic as the residual rate error (from the $\hat{p}$ used to calculate $\hat{\theta}_i$) may be as low as γ_ϵ^* (from section V-B). We therefore estimate the total error as $E_i^T = E_i + \gamma_\epsilon^*(C_d(t) - C_d(T_{f,i}))$.

(ii) First we consider only those packets which fall into a SKM related *offset window* τ' seconds wide before t, as we only know how to relate current and past offset values within the context of the SKM. For each packet i within the window we penalise poor total quality very heavily by assigning a quality weight via $w_i = \exp(-(E_i^T/E)^2) \leq 1$, which becomes very small as soon as the total quality lies away from a band defined by the size of $E > 0$. The graphs below justify the particular choices $\tau' = \tau^*$ and $E = 4\delta$.

(iii) An estimate can now be formed through a normalised weighted sum over the offset window:

$$\hat{\theta}(t) = \sum_i w_i \hat{\theta}_i / \sum_i w_i,$$

which amounts to a constant predictor on a packet by packet basis. The local rate estimates can be used to introduce linear prediction instead:

$$\hat{\theta}(t) = \Big(\sum_i w_i (\hat{\theta}_i + \hat{\gamma}_l \, (C_d(t) - C_d(T_{f,i}))) \Big) / \sum_i w_i$$

where $\hat{\gamma}_l = 1 - \hat{p}_l(t_{f,i})/\bar{p}$ is the estimate of the residual skew relative to $\hat{p} = \bar{p}$ (the value used in $C_u(t)$ to calculate $\hat{\theta}_i$).

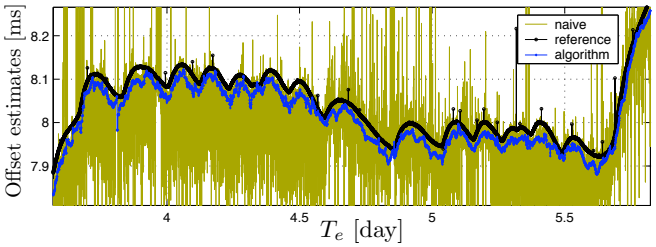

Fig. 8. Time series of $\hat{\theta}_i$ using the algorithm (without local rates) against reference values θ_i^g, with naive estimates in the background. The oscillations are due to the airconditioning in the machine room. The (rare) spikes in the reference values are examples of corruption by timestamping noise (corresponding $T_{f,i}$ are corrected using neighbouring values).

If all the packets in the window have poor quality then even the weighted estimate can perform poorly. Indeed, under periods of high congestion we may find that $\sum_i w_i = 0$ to machine precision. To avoid being influenced in such cases, when $\min_i(E_i^T) > E^{**}$, we instead base the estimate on the last weighted estimate accepted (say at packet i), which gives

$$\hat{\theta}(t) = \hat{\theta}(t_{f,i}) \quad (11)$$

$$\hat{\theta}(t) = \hat{\theta}(t_{f,i}) + \hat{\gamma}_l * (C_d(t) - C_d(T_{f,i})), \quad (12)$$

depending upon whether the local rate correction is used or not We set $E^{**} = 6E$, or about 3 'standard deviations' away in the Gaussian-like weight function, so that the estimate will only be abandoned when quality is extremely poor.

(iv) Just as for the rate estimates, we put in place a high level sanity check to ensure that the offset estimate cannot vary in a way which we know is impossible, no matter what data it receives. If successive offset estimates differ by more than a given function of τ^* and γ^* then the most recent trusted value will simply be duplicated. In this section we use simple thresholding, set at 1ms, which is orders of magnitude beyond the expected offset increment between neighboring packets. It is very important that such a sanity check be just that, for example in this case that the threshold be set very high. Attempting to reduce this value to 'tune' its performance would be tantamount to replacing the main filtering algorithm with a crude alternative dangerously subject to 'lock-out', where an old estimate is duplicated ad infinitum. An instance when the sanity check was needed is given later.

An example of the offset error $\hat{\theta}_i$ of the uncorrected clock $C_u(t)$, estimated by the algorithm at successive packet arrivals, is given in Figure 8. The performance is very satisfactory: the algorithm succeeds in filtering out the noise in the naive estimates (shown in the background), producing estimates which are only around $\hat{\theta}_i - \theta_i^g = -30\mu$s away from the DAG reference values $\theta_i^g = C_u(t_{f,i}) - T_{f,i}^g$. This difference is just the negative of the offset error of $C_a(t)$ as measured by DAG, since $C_a(t_{f,i}) = C_u(t_{f,i}) - \hat{\theta}_i = C_u(t_{f,i}) - \theta_i^g - (\hat{\theta}_i - \theta_i^g)$. However, given that the path asymmetry is estimated as $A \approx 50\mu$s, which implies an ambiguity in offset of $A/2 \approx 25\,\mu$s (Equation (8)), and that timestamping issues limit the verifiability of our results to around 10μs in any case, the total offset of 30μs is seen to be essentially due to asymmetry, showing that the algorithm is working very well in eliminating the errors due to variable network delay.

We now consider the performance of the clock more systematically as a function of key parameters. In Figure 9 the central curve shows the median of $\hat{\theta}_i - \theta_g(t) = -(C_a(t_{f,i}) - T_{f,i}^g)$ as a function of the offset window τ', calculated over the entire 3 weeks. It is around 28μs over a wide range of window sizes, and the inter-quartile range (IQR) is likewise very small, of the order of 11μs for the optimal value at $\tau'/\tau^* = 0.5$, again with low sensitivity to window size. Even the range from the topmost (99th percentile) to the bottommost (1st percentile) curve is only of the order of 50μs. Essentially identical results were obtained over the 3 month period of continuous monitoring (section VI) using *ServerInt*, of which the current 3 week trace is a subset.

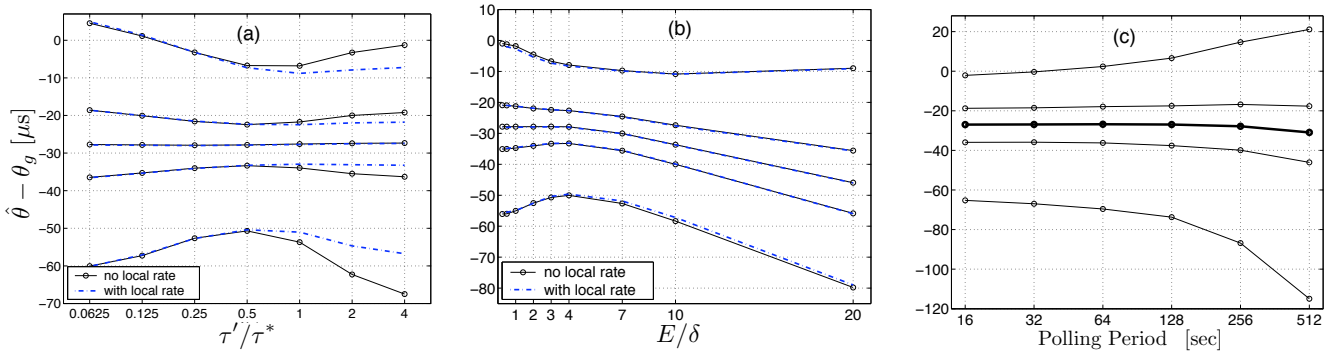

Fig. 9. Sensitivity analysis of clock offset with respect to key parameters: (a) window size τ', ($E = 4\delta$, with and without local rate: $\tau_l = 20\tau^*$) (b) quality assessment E ($\tau' = \tau^*/2$, with and without local rate: $\tau_l = 20\tau^*$), and (c) polling period ($\tau' = \tau^*$, $E = 4\delta$ without local rate). From top to bottom: the 99%, 75%, 50% (the median) 25% and 1% percentiles of the error $\hat{\theta}_i - \theta_g(t) = -(C_a(t_{f,i}) - T^g_{f,i})$ of $\hat{\theta}_i$ The sensitivity is very low in each case.

Figure 9(a) also compares the estimation with and without the use of local rates. The differences are marginal for this trace, with local rate we only gain some immunity to the effects of choosing a window size too large. In either case, the insensitivity of the results to the precise value of τ' is encouraging, and the fact the optimum is close to $\tau' = \tau^*$ is precisely what we would expect from our SKM formulation, and a natural validation of it.

Figure 9(b) examines the results as a function of the quality assessment parameter E. Again very low sensitivity is found, with optimal results being achieved at a small multiples of δ, as one would expect. We also performed sensitivity analyses with respect to the aging rate parameter γ^*_ϵ, and the local rate window width τ_l. For each, the sensitivity is so low for this relatively well behaved data that they could be omitted entirely with little effect. These refinements bring tangible benefits only under certain conditions, such as high loss, where packets in the τ' window may be much further in the past than intended, or when parameters have not been poorly chosen.

We next examined performance with respect to polling period. As they were essentially identical, we omit the results using the local rate correction. We compare the period of 16 seconds, used so far, with others including the usual range of allowed default values: 64 to 256. The sensitivity results with respect to τ' were very similar to those reported in Figure 9(a), although the optimal 'kink' position moves to slightly larger values. The results for E were unchanged beyond a slight spreading of the error distribution.

We now keep the other parameters fixed at $\tau' = \tau^*$, $E = 4\delta$, and $\gamma^*_\epsilon = 0.02\,\text{PPM}$ and vary the polling rate. Figure 9(c) shows again that the sensitivity is very low. In particular the median error only changed by a few microseconds despite a reduction of raw information by a factor of 32 across the plot. This is significant since it is important, in generic applications, that time servers not be excessively loaded.

Finally, we examine the performance of the algorithm over the four different traces, representing different host-server environments, used in Figure 3. We use $\tau' = \tau^*$, $E = 4\delta$, and $\tau_l = 5\tau^*$, and a polling period of 64. We see the reduction in variability when moving from the laboratory into the more stable machine room (MR), and a further improvement when moving from *ServerInt* to the even closer local server. The jump in median error when *ServerExt* is used is due to the much increased path asymmetry, an inescapable phenomenon for *any* remote client-server algorithm which assumes $A = 0$. As before, the error is approximately $A/2$ using the values from table II, much smaller than the RTT of $r = 14.2\,\text{ms}$. The increased variability is due to the higher noise resulting from many hops, making quality packets much rarer. With *ServerExt* we are stress testing the algorithm using a server that is much further away in all senses than necessary.

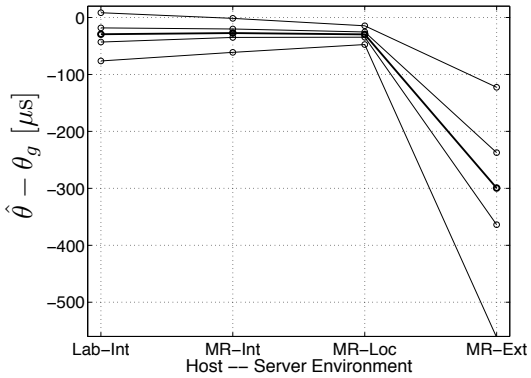

Fig. 10. Performance over four different operating environments (same data sets as Figure 3). Top to bottom: 99%, 75%, 50% 25% and 1% percentiles of the error $\hat{\theta}_i - \theta_g(t)$ of $\hat{\theta}_i$.

VI. A Robust Working System

Here we discuss additional issues that are important in a working system, and examine one of the most important, robustness to routing changes. We then present results from a new C implementation, using several new data sets.

A. Robustness

The challenge for a working system is to adapt core algorithms to a heterogeneous environment, repleat with both foreseen and unforeseen anomalies, without compromising principles or performance. There are many issues here which are addressed in our implementation, including modifications required to algorithms in a warmup phase, on-line formulations that summarise (and forget) the past efficiently, and the avoidance of unwanted dynamic interactions between estimates.

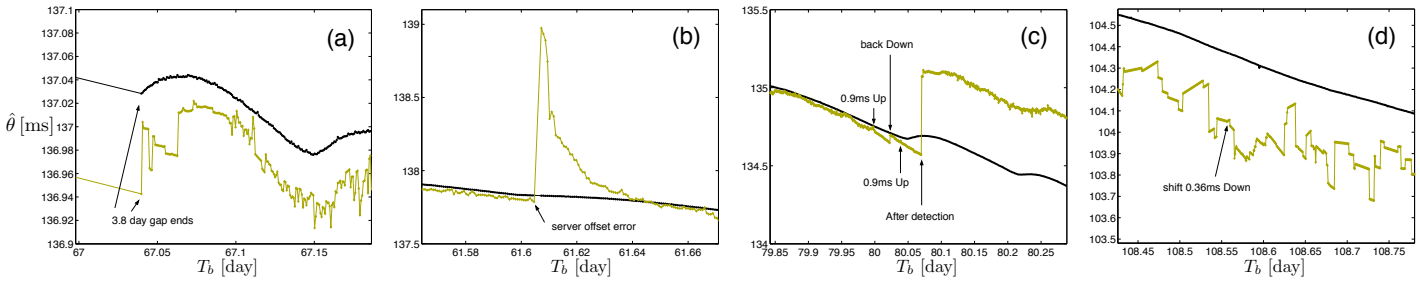

Fig. 11. Performance of the $\hat{\theta}$ algorithm under extreme conditions (smooth thicker curves are reference offsets, variable curves with arrows are estimated offsets): (a) loss of data over 3 days, (b) level shift error of 150 ms in the server clock (triggers sanity check), (c) artificial temporary and permanent upward level shifts inducing change in A, (d) real permanent downward level shift using *ServerExt* with A constant. $\tau' = 2\tau^*$, $\tau_l = 5\tau^*$, $\tau_s = \tau_l/2$.

With the exception of robustness to changes in r (for example caused by IP layer routing changes), for space reasons we cannot discuss these in detail (see [6]). Instead, we illustrate the performance of the final result on real, heterogeneous data.

We illustrate how the full system (for the moment we use the same C implementation as in [6]), reacts in different circumstances. Figure 11 zooms in on extreme events which occurred during a continuous measurement period which continued the 3 month trace of the previous section to include an additional gap of 6 days, followed by the change to *ServerLoc* for 1 week, and then to *ServerExt*. Figure 11(a) demonstrates the fast recovery of the algorithm even after the 3.8 day gap in data collection Figure 11(b) shows the impact of a server error lasting a few minutes, during which $T_{b,i}$ and $T_{e,i}$ were (inexplicably) each offset by 150ms. As errors in server timestamps do not affect the RTT measurements at the host, this is very difficult to detect and account for. However, the offset (and local rate) sanity check algorithm was triggered, which limited the damage to a millisecond or less.

The remaining examples involve *level shifts*, that is changes in any of the minimum delays $d^{\uparrow}$, $d^{\rightarrow}$ or $d^{\downarrow}$ (see Equation (6)), and hence r. This is primordial as filtering of network delay is based on the estimation of r. On-line algorithms must be able not only to survive such a transition in r but continue to perform well. We now discuss the key issues.

Asymmetry of shift direction:
These are fundamentally distinct:
Down: congestion cannot result in a downward movement, so the two can be unambiguously distinguished → easy detection.
Up: indistinguishable from congestion at small scales, becomes reliable only at large scales → difficult detection.

Asymmetry of detection errors:
The impact of an incorrect decision is dramatically different:
Judge quality packet as bad: an undetected upward shift looks like congestion, to which algorithms are robust → non-critical.
Judge bad quality packet as good: falsely interpreting congestion as an upward shift immediately corrupts estimates, perhaps very badly → critical to avoid.

Asymmetry of offset and rate:
Offset: underlying naive estimates $\hat{\theta}_i$ *remain valid* to the $\hat{\theta}(t)$ algorithm even after a future shift → store past estimates and their point errors relative to the $\hat{r}$ estimate made at the time.
Rate: $\hat{p}$ and $\hat{p}_l$ estimates are made between a *pair* of packets, so must compare them using a common point error base → use point errors relative to current error level (after any shifts).

If the procedures of the last paragraph are followed, few additional steps are needed to assemble a robust detection and reaction scheme for level shifts.

The Level Shift Algorithm:

The two shift directions are treated separately:

Down:
Detection: Automatic and immediate when using $\hat{r}$.
Reaction: Offset: no additional steps required.
Rate: No additional steps required. The algorithms will see the shift as poor quality of past packets and react normally. In time, increasing baseline separation and windowing will improve packet qualities again.

Up:
Detection: Based on maintaining a local minimum estimate $\hat{r}_s$ over a sliding window of width τ_s. Unambiguous detection is difficult and the consequences of incorrect detection serious. We therefore choose τ_s large, $\tau_s = \tau_l/2$, and detect a shift (at $t = C_a(T_{f,i}) - \tau_s$) if $|\hat{r}_s - \hat{r}| > 4E$.
Reaction: First update $\hat{r} = \hat{r}_s$ (and on-line window estimate), and recalculate $\hat{\theta}_i$ values and reassess their point qualities back to the shift point. Otherwise no additional steps required. Before detection, the algorithms will see the packets as having poor quality, and react as normal. Since the window is large, estimates may start to degrade toward the end of the window.

In Figure 11(c) two upward shifts of 0.9ms were artificially introduced. The first, being under τ_s in duration, was never detected and makes little impact on the estimates. The second was permanent. Occurring at 80.04 days, it was detected a time τ_s later, resulting in a jump in subsequent offset estimates (the original on-line estimates, not the recalculated ones, are shown). This jump is not a failure of the algorithm, but is the unavoidable result of the change in A of $0.9/2 = 0.45$ ms, as the shifts were induced in the host→server direction only. The important point concerning the algorithm is that the estimation difficulties resulting from the shift in r have been kept well controlled at around 50μs. Finally, the naturally occuring permanent shift in Figure 11(d) occurs equally in each direction, so that A does not change, and is also downward, so that detection and reaction are immediate. The result is no observable change in estimation quality, and no jump due to A, the shift is absorbed with no impact on estimates.

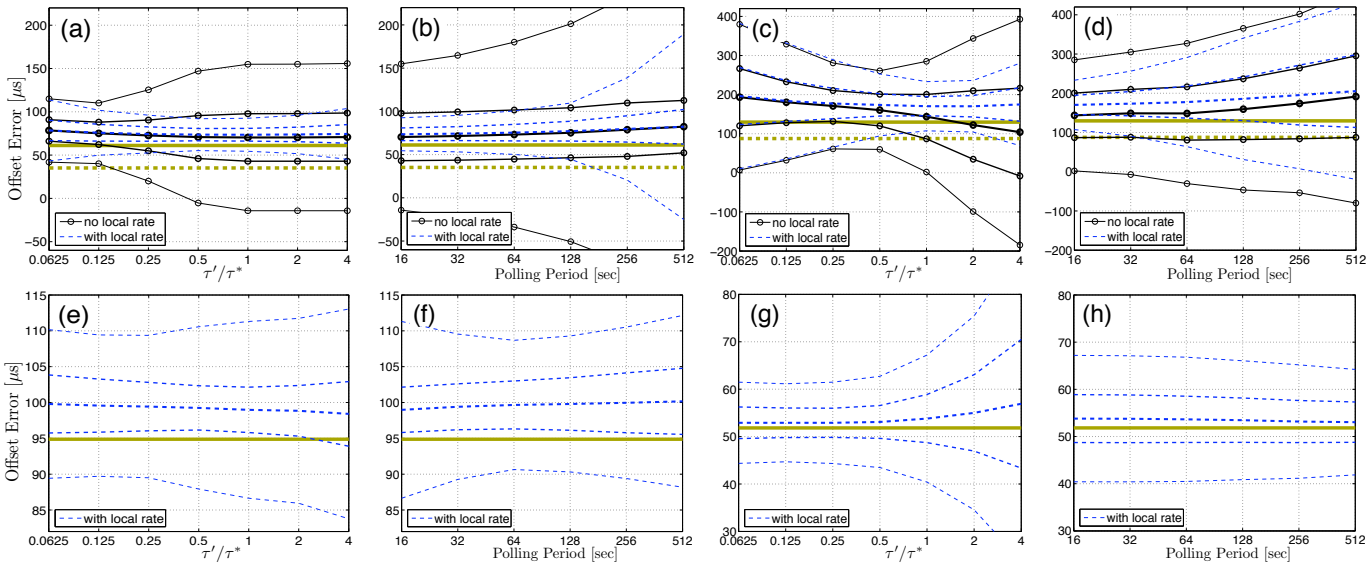

Fig. 12. Sensitivity analysis of offset error $C_a(t_{f,i}) - T^g_{f,i}$, new implementation, for window size and polling period. In each plot from top to bottom the curves are the 95%, 75%, 50% (the median) 25% and 5% percentiles of the offset error. Plots (a) and (b): *wallaby* to *ServerInt2* over 32 days. Plots (c) and (d): *tastiger* to *ServerExt2* over 21 days. Plots (e) and (f): *green* to *ServerLAN* over 64 days. Plots (g) and (h): *tastiger* to *ServerLAN* over 64 days. Full horizontal lines show the error component due to path asymmetry alone. Dashed horizontal lines (top row only) show the network asymmetry A_n.

B. Experiments With a Production TSCclock

This section documents the performance of an entirely new C implementation. Algorithmically, it incorporates many small improvements to robustness and efficiency. As a system, it has improved kernel timestamping, is modular, uses less memory, is far more configurable, runs as a system daemon, and can be easily installed as a package on recent Linux and BSD systems in parallel with the SW-NTP.

The timestamp data sets featured here were collected on three hosts in 2006-7: *green* and *tastiger* with BSD-6.1 kernels, and *wallaby* running BSD-5.3, using the stratum-1 NTP servers of table III. An important improvement in validation methodology for these new data sets arises through capturing DAG timestamps in both the outgoing ($t^g_{a,i}$) and incoming ($t^g_{f,i}$) directions (Figure 1). First, this allows path asymmetry to be decomposed into host and network components: $A = A_h + A_n$, where A_n is defined analogously to (9) but with DAG replacing host, and $A_h = \min_i(d_i^{h\uparrow}) - \min_i(d_i^{h\downarrow})$. As both the DAG and server are synchronized, A_n can be readily estimated for each server, and values are given in table III.

As mentioned in section II-E, minimum system noise can be of the order of $r^h = 150\mu$s, an appreciable proportion of minimum RTT for nearby servers! In terms of asymmetry in this noise, the same fundamental ambiguity which exists between the host and the server, exists between the host and the DAG. Hence, although r^h places bounds on A_h (just as r does on A), A_h cannot be measured, and therefore neither can A. However, it is nonetheless possible, with bidirectional DAG timestamps, to remove the effect of A completely. Specifically, if we decompose the median total error $\theta = A/2 + \theta'$ into the inherent error $A/2$ due to path asymmetry and the 'true' error θ' representing the performance of the TSCclock algorithm itself, then the latter can be measured as

$$\theta' = \theta - \frac{A}{2} = \text{median}(C_a(t_{f,i}) - T^g_{f,i}) - \frac{1}{2}(A_n + r^h).$$

since $\theta(t_{f,i}) = C_a(t_{f,i}) - t_{f,i} = C_a(t_{f,i}) - (t^g_{f,i} + d_i^{h\downarrow})$, and, assuming that packet i experiences minimum delays everywhere, $d_i^{h\downarrow} = (r^h - A_h)/2$ (a symmetric expression holds using outgoing timestamps, however the system noise is lower with incoming). The correction $(A_n + r^h)/2$, which is specific to the (server, host) pair, is shown as the solid horizontal line in Figure 12 and should be compared to the median error. The upper plots also show the server component $A_n/2$ as the dashed horizontal line.

We first examine the window size and poll period sensitivities for a 32 day trace of the TSCclock on *wallaby* synchronizing to *ServerInt2* (with default values $\tau' = \tau^* = 1024$, poll-period= 16, $E = 6\delta$, γ^*_ϵ =0.1 PPM, $\tau_l = 5\tau^*$). Figures 12(a) and (b) show that, in contrast to Figure 9, there is a significant advantage in using the local rate correction, due to the wider range of drift over this trace, which results in $\bar{p} - \hat{p}_l$ being larger. For example the IQR at $\tau = \tau^*$ in Figure 12(a) is only 15μs using the local correction, compared to 55μs without. In terms of median error, the gap between the total median clock error and the component due to asymmetry (thick horizontal line) is only 12μs when using the local rate correction, compared to 10μs without. In other words, apart from inherent asymmetry issues, the TSCclock (using local rates) has errors which are tightly clustered around a median value which is so low it stresses our ability to measure it given system noise. The IQR values and sensitivity conclusions are comparable to those of Figures 9(a),(c) using a similar server.

Server	Reference	Distance	r (min RTT)	Hops	A_n
ServerLAN	GPS	3 m	0.24 ms	1	24μs
ServerInt2	GPS	300 m	0.61 ms	5	70μs
ServerExt2	GPS	3500 km	37.7 ms	10	175μs

TABLE III
CHARACTERISTICS OF THE STRATUM-1 NTP SERVERS USED IN 2006-7.

Now consider a 20 day trace of the TSCclock on *wallaby* synchronizing to *ServerExt2*. Figures 12(c) shows that, again, the performance is best at around $\tau = \tau^*$ where the IQR using local rates is only 49μs, a good result for such a distant server, as is the excess median error which is only 40μs above the asymmetry component. Plot 12(d) shows that the drop in performance with increasing poll period is graceful.

The bottom row in Figure 12 supports a comparison of *green*, the 600MHz Pentium III machine used above on the 2003 data, and *tastiger*, a 3.4Ghz Pentium Core Duo. Each synchronized to *ServerLAN* in a simultaneous experiment lasting 64 days in the same laboratory temperature environment. Because of the proximity of *ServerLAN*, both A_n (24μs) and r^g (166μs) are very small. Host system effects therefore dominate the network ones, facilitating a host comparison. We only show results using the local rate correction. The results are very good: for default parameters the IQR is below system noise at only 6μs for *green* and 10μs for *tastiger*, and the excess median error above the asymmetry component (horizontal line) is 4μs and 2μs respectively. The Allan plots (not shown) fail to give any clear indication of a difference in TSC oscillator characteristics between the different hardware. However, the system noise was larger for *green* at $r^h = 166\mu$s, compared to $r^h = 78\mu$s for *tastiger*, resulting in a total error $(166 - 78)/2 = 44\mu$s larger for the older machine.

Finally, we checked how the new implementation compares to the old. When applied to the full 3 month trace collected in 2003, [6] reports that the proof of concept implementation (using $\tau' = 2\tau^*$, $E = 4\delta$, $\gamma^*_\epsilon = 0.01$PPM and local rates) gave [median,IQR]= $[31, 15]\,\mu$s compared to $[29, 14]\,\mu$s now, using a poll period of 16 seconds, and $[33, 24]\,\mu$s compared to $[31, 24]\,\mu$s now when polling every 256 seconds.

Thus far we have used stratum-1 servers exclusively, in order to evaluate the TSCclock in isolation, free of errors arising from an inaccurate server. However, in practice servers of lower stratum may be used, and it is important to ensure that the TSCclock reacts stably in this environment. To test this, we first benchmarked *green* using *ServerLAN*, obtaining the best-case $\hat{\theta}$ performance shown in Table IV, and a reference value for period of $\bar{p}^* = 1.822638984$[ns]. We then pointed *green* to two stratum-2 servers, first on the LAN, then another in the same location as *ServerInt2*. Table IV gives the median and IQR of two metrics, the rate error $e_p = (\hat{p} - \bar{p}^*)/\bar{p}^*$ relative to $\bar{p}^*$, and the offset error $e_\theta = C_a(t_{f,i}) - T^g_{f,i}$ (using a poll period of 16[sec] and correcting for asymmetry). The values of $\hat{p}$ are extremely stable despite the lower stratum. The results for e_θ are also very good (better for the closer server as usual), indicating that the TSCclock is tracking its server well. Of course, if the absolute performance of the server is poor, so will be that of the TSCclock. As an additional example, earlier we reported an IQR of $15\,\mu$s for *wallaby* to *ServerInt2*. For *green* to *ServerInt3* this increases, but only to $64\,\mu$s.

Server	Stratum	Med(e_p)	IQR(e_p)	Med(e_θ)	IQR(e_θ)
ServerLAN	1	0 PPM	0.005 PPM	11μs	9μs
tastiger	2	-0.09 PPM	0.004 PPM	16μs	12μs
ServerInt3	2	-0.08 PPM	0.03 PPM	48μs	64μs

TABLE IV
PERFORMANCE COMPARISON WITH STRATUM-2 SERVERS.

CPU Load	State of p_l	Duration	Med(e_θ)	IQR(e_θ)
Very light	Stable	5 hours	3μs	14μs
Saturated	Transitional	2 hours	7μs	33μs
Saturated	Stable	5 hours	3μs	16μs

TABLE V
PERFORMANCE AS A FUNCTION OF CPU LOAD.

Another dimension of robustness is host load, which up to now has been very light. Table V shows results before and after a busy loop was activated which pushed CPU usage up to 100%. As expected, the load had no impact on algorithm operation, which only requires resources once each polling period (here 16 [sec]). One potential impact is of higher system noise on timestamping, however we saw no sign of this neither in the raw timestamp data, nor via algorithm diagnostics. Sustained high load does however result in a considerable temperature increase, and a corresponding unusually large increase in local rate of the TSC, measured as 1.0 PPM. The middle row of table V shows that during a transitional phase, during which the local rate estimate p_l adapts to this new value, e_θ is mildly affected, after which (row three), performance returns to light-load values. Throughout and as expected, $\hat{p}$ was stable and allowed delay filtering as normal despite the radical load increase. Note that if strong fluctuations were endemic, then performance would be downgraded as above consistent with a permanent 'transition phase'.

Figure 13 shows a comparison of SW-NTP (synchronized to *ServerLAN* in broadcast mode) against the TSCclock over 14 days. The SW-NTP fluctuates in a 1[ms] band, two orders of magnitude wider than that of the TSCclock. Experiments conducted in the environments of Figure 12 showed similar levels of improvement. An authoritative comparison requires controlled testing over a wide range of conditions and cannot be attempted here. In [14] we describe how such testing can be performed, and in so doing provide a number of comparisons which are consistent with Figure 13. Using this methodology, [15] makes a beginning on a systematic comparison and confirms improvement exceeding 1 order of magnitude over a range of servers and polling period. A detailed benchmarking study, including the influence of hardware, load, temperature, server stratum and distance, congestion, and algorithm parameters, is the subject of another paper.

We do not give explicit results for the difference clock $C_d(t)$ (see [14]). However, in months of operation we observed that

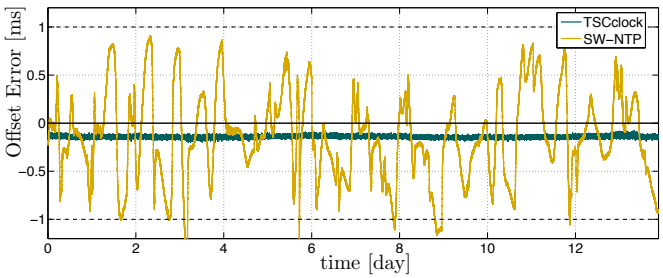

Fig. 13. A sample comparison of SW-NTP and TSCclock performance.

$\hat{p}$ typically varies in a range of a few part in 10^9, with very rare 'jumps' of the order of a few parts in 10^8. The immediate and remarkable implication is that, even if the network connectivity were lost (resulting in $\hat{p}$ being frozen) for extended periods, the difference clock would be essentially unaffected.

VII. Conclusion

We have presented a detailed re-examination of the problem of inexpensive yet accurate clock synchronization for networked devices. It rests on an empirically validated, parsimonious abstraction of the CPU oscillator as a timing source, accessible via the TSC register in popular PC architectures, and then builds on the key observation that the measurement of time differences, and absolute time, requires separate clocks. We showed how this insight infiltrates all levels of the clock design and synchronization algorithm problem, leading in particular to a decoupling of timestamping filtering from synchronization itself, and a decoupling of absolute synchronization from rate synchronization. The result is distinct algorithms: the *difference* and *absolute TSCclock*, with performance and robustness which fulfill the promise of the high stability of the TSC but which are, inherently, qualitatively and quantitatively very different. The status-quo solution, embodied in the *ntpd* daemon, offers an absolute clock only.

Using many months of real data from 6 different network accessible (stratum-1 NTP) time servers, and both 'old' and new host hardware, we demonstrated that the TSCclocks are very accurate, as well as robust to many factors including packet loss and loss of server connectivity, routing changes, network congestion, temperature environment, timestamping noise, and even faulty server timestamps. We showed in detail how the performance of the absolute TSCclock is insensitive to key algorithm parameters, and explained why the difference TSCclock is virtually invulnerable to them. For absolute synchronization, we stressed the need to separate out errors due to the algorithm (of the order of a few 10's of μs under reasonable conditions, close to the system noise limit) from fundamental limitations due to path asymmetry. The impact and magnitude of path asymmetry, notably for applications such as one-way delay measurement, is context dependent, however the TSCclocks should allow many applications requiring accurate and reliable timing to do away with the cost of hardware based synchronization, such as using GPS receivers. In particular, the difference clock is not impacted by path asymmetry, and is robust to extreme events such as weeks of connectivity loss.

We give detailed performance results for an implementation for BSD and Linux platforms. It synchronizes to an NTP server and is capable of being run as a system daemon in parallel with existing system software clocks based on *ntpd*. The algorithms however could easily be adapted for use with other kinds of oscillators, and other kinds of time servers.

VIII. Acknowledgements

The original BSD-2.2.8 timestamping code was developed by Michael Dwyer. Mark Santcroos of RIPE-NCC contributed to the port to BSD-5.3. We thank Henk Uijterwaal at RIPE-NCC for the long term loan of a GPS receiver. The original work was supported by Ericsson.

References

[1] D. Mills, "Internet time synchronization: the network time protocol," *IEEE Trans. Communications*, vol. 39, no. 10, pp. 1482–1493, October 1991, condensed from RFC-1129.

[2] C. L. et al., "Experience with an adaptive globally synchronizing clock algorithm," in *Proc. of 11th ACM Symp. on Parallel Algorithms and Architectures*, June 1999.

[3] V. Paxson, "On calibrating measurements of packet transit times," in *Proc. ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. ACM Press, 1998, pp. 11–21.

[4] A. Pásztor and D. Veitch, "PC based precision timing without GPS," in *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, Del Rey, California, 15-19 June 2002, pp. 1–10.

[5] "RIPE NCC Test Traffic Measurements," http://www.ripe.net/ttm/.

[6] D. Veitch, S. Babu, and A. Pásztor, "Robust Synchronization of Software Clocks Across the Internet," in *Proc. ACM SIGCOMM Internet Measurement Conf.*, Taormina, Italy, Oct 2004, pp. 219–232.

[7] D. Mills, "The network computer as precision timekeeper," in *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, Reston VA, December 1996, pp. 96–108.

[8] A. Pásztor and D. Veitch, "A precision infrastructure for active probing," in *Passive and Active Measurement Workshop (PAM2001)*, Amsterdam, The Netherlands, 23-24 April 2001, pp. 33–44.

[9] E. Corell, P. Saxholm, and D. Veitch, "A User Friendly TSC Clock," in *Passive and Active Measurement Conference (PAM2006)*, Adelaide Australia, March 30-31 2006.

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

[11] J. Micheel, I. Graham, and S. Donnelly, "Precision timestamping of network packets," in *Proc. SIGCOMM IMW*, November 2001.

[12] P. Abry, D. Veitch, and P. Flandrin, "Long-range dependence: revisiting aggregation with wavelets," *Journal of Time Series Analysis*, vol. 19, no. 3, pp. 253–266, May 1998, Bernoulli Society.

[13] K. Römer, "Time synchronization in ad hoc networks," in *MobiHoc'01: Proc. 2nd ACM international symposium on Mobile ad hoc networking & computing*. New York: ACM Press, 2001, pp. 173–182.

[14] J. Ridoux and D. Veitch, "A Methodology for Clock Benchmarking," in *Tridentcom*. Orlando, FL, USA: IEEE Comp. Soc., May 21-23 2007.

[15] ——, "Ten Microseconds Over LAN, for Free," in *Int. IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS '07)*, Vienna, Austria, Oct 1-3 2007.

Darryl Veitch (SM'02) completed a BSc. Hons. at Monash University, Australia (1985) and a mathematics Ph.D. from Cambridge, England (1990). He worked at TRL (Telstra, Melbourne), CNET (France Telecom, Paris), KTH (Stockholm), INRIA (Sophia Antipolis, France), Bellcore (New Jersey), RMIT (Melbourne) and EMUlab and CUBIN at The University of Melbourne, where he is a Principal Research Fellow. His research interests include traffic modelling, parameter estimation, active measurement, traffic sampling, and clock synchronisation. ACM SIG Member 1832195.

Julien Ridoux (S'01-M'06) received the M.Eng. degree in Computer Science (2001) and M.Sc. degree in Telecommunication and Networks (2002) respectively from the Ecole Polytechnique de l'Universite de Nantes (France) and University Paris 6 (France). In 2005 he received the Ph.D. degree in Computer Science from the University Paris 6. Since 2006 he is a Research Fellow at The University of Melbourne where his main research interests are distributed clock synchronization and Internet traffic modeling.

Satish Babu Korada received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Delhi, India in 2004. As an undergraduate he did a summer internship at The University of Melbourne in 2003. He completed pre-doctoral school at the Ecole Polytechnique Fédérale de Lausanne, Switzerland in 2005 and currently is continuing his Ph.D. there. His current research interests are coding theory and applications of statistical physics in communications.