Robust Synchronization of Software Clocks Across the Internet

Darryl Veitch
ARC Special Research Center for Ultra Broadband Information Networks
Dept. of Electrical and Electronic Engineering
The University of Melbourne, Australia
dveitch@unimelb.edu.au

Satish Babu
Indian Institute of Technology
New Delhi, India
k_satish@msn.com

Attila Pàsztor
Ericsson Hungary R&D
Budapest, Hungary
Attila.Pasztor@ericsson.com

ABSTRACT

Accurate, reliable timestamping which is also convenient and inexpensive is needed in many important areas including real-time network applications and network measurement. Recently the TSC register, which counts CPU cycles in popular PC architectures, was proposed as the basis of a new software clock which in terms of rate performance performs as well as more expensive GPS alternatives. Smooth and precise clock rate is essential to measure time *differences* accurately. We show how to define a TSC based clock which is also accurate with respect to *absolute* time. The clock is calibrated by processing, in a novel way, timestamps contained in the usual flow of Network Time Protocol (NTP) packets between a NTP server and the existing software clock, and TSC timestamps made independently on the host side. Using real measurements over 4 months, validated with a GPS synchronized hardware timing solution, the algorithm measured absolute time with a median error of only 30 microseconds when using a nearby stratum-1 NTP server. Results for two other servers are given. We also provide new algorithms for the robust determination of clock rate. We exploit the reliability of the available hardware to design synchronization algorithms which are inherently robust to many factors including packet loss, server outages, route changes, temperature environment, and network congestion.

Categories and Subject Descriptors

C.2.m [**Computer-Communication Networks**]: Miscellaneous; D.4.m [**Operating Systems**]: Miscellaneous

General Terms

Algorithms, Reliability

Keywords

timing, synchronization, software clock, NTP, GPS, network measurement, round-trip time

IMC'04, October 25–27, 2004, Taormina, Sicily, Italy.

1. MOTIVATION

The availability of an accurate, reliable, and high resolution *clock* is fundamental to most complex devices, and computers running multi-tasking operating systems such as Unix are far from the exception. Ongoing synchronization to a time standard is necessary to keep the *offset* of such a clock, that is its departure from the true time* at a given time, to within acceptable bounds. 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 NTP packets containing timestamp information to be exchanged between NTP time server(s) and the host across a network. The algorithms filter and process these timestamps, determine the offset, and deliver rate and offset adjustment recommendations to the SW clock.

For many purposes this *SW-NTP* clock and synchronization 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 as low as 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 larger reset adjustments which can in extreme cases be of the order of seconds. 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. Whilst this conveniently preserves 'causality' (assuming no resets), it 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*. Taking differences of timestamps is basic to most applications, and in the case where they are made by the same clock, any constant error in offset is cancelled, leaving rate as the key clock characteristic.

One application where these issues are important is inexpensive measurement of packet switched 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, 5], are widely recognised in the network measurement community. They have led many networking researchers to return to local rather than remote clock synchronization. The Test Traffic Measurement network of RIPE NCC for example [6], consisting of over 100 customised PC's across Europe and elsewhere, uses

*Newtonian space-time will be assumed in this paper.

locally attached Global Positioning System (GPS) based disciplining of the standard SW clock, which improves synchronization to around 10μs. The simple active probing based measurements of this network do not require high accuracy, and nominally the SW-NTP clock would have been sufficient (for example to give one-way delays to a few milliseconds). Instead, the effort was made to install GPS synchronization in every node. 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 may not be possible), and potentially high installation costs, making a multi-node measurement effort such as RIPE NCC's extremely ambitious, and even modest measurement efforts problematic. The exploitation of radio based alternatives for synchronization relies on the presence of the appropriate network and also implies additional hardware. To avoid these problems, it is desirable to provide improved network based synchronization with 'GPS-like' reliability, and increased accuracy, using inexpensive PC's with no additional hardware.

In [5] a new clock was proposed which made significant progress towards this aim. It is based on the *TimeStamp Counter* (TSC) register, found in Pentium class PCs and other architectures, which counts CPU cycles[†]. The essence of this clock is very simple. The TSC register is used to keep track of time at very high resolution, for example 1 nanosecond for a 1 gigahertz processor. Updating of this register is a hardware operation, and reading it and storing its value involves only very fast memory accesses. Provided that we have an accurate estimate of the true duration, p, of a clock cycle, time differences measured in TSC units can readily be converted to time intervals in seconds: $\Delta(t) = \Delta(\text{TSC}) * p$. This simple idea is feasible because of the fact that CPU oscillators have high *stability*, so that the cycle period is, to high precision, constant over quite long time periods, and residual offset errors accumulate only slowly. Note that the TSC is already routinely employed in software clock solutions, although typically its role is to interpolate between the timer interrupts generated by another hardware timing source on the motherboard, rather than being the primary source. By 'TSC clock' we are really referring to the overall solution described here, its principles and methods, and in particular to the synchronization algorithms used and the techniques insuring robustness, and *not simply to the fact that the TSC is employed*.

There are two areas in which our clock differs markedly from the SW-NTP alternative, even though, as we describe in detail later, we propose a synchronization technique which also makes use of the NTP protocol and server network. The first is that in SW-NTP the emphasis is firmly on offset, rate performance is not independently considered, whereas in our TSC clock, rate is seen not only as the more important of the two, but also the logical foundation stone on which to construct a robust clock. Another way of expressing this key difference is that we keep frequency and offset calibration as decoupled as possible, instead of closely integrating them into a single dynamical system. The second is that SW-NTP implicitly follows the philosophy that the host clock is inherently inaccurate, so that it must be constantly disciplined by a reference source. In our approach the view is that, on the contrary, the host clock is inherently 'good', it simply must be calibrated. This dramatic change of viewpoint strongly influences the filtering and estimation procedures used to extract the reference timing information from the timestamps received across the network, and both motivates and justifies a revisiting of this question.

The first aim of this paper is to address in detail the question of robust absolute or *offset* synchronization in the context of the TSC clock. Offset synchronization is a very different, and more difficult problem than that of *rate* synchronization/calibration, which was not addressed at all in our previous work [5]. Here we describe principles, a methodology, and filtering procedures which make accurate offset measurement and thereby reliable absolute synchronization possible. We work within the context of the TSC based clock, however the algorithms could be applied more generally to other timing sources. We make use of the timestamps contained in the normal flow of NTP packets to and from the SW-NTP clock and a nearby NTP server. This allows the system and applications to run as normal, and no change to the NTP protocol itself is required. In parallel, TSC timestamps of the NTP packets are also made, and processed in a new way to produce the *TSC-NTP* clock. In fact we explain the need for, and propose, two clocks: a *difference clock* used for measuring time differences (up to a certain scale), and an *absolute clock* when absolute timestamps are required, for example when measuring one-way delay in networks. We believe that pointing out the need for this differentiation, which arises directly from a rate centric view, is in itself of some importance. The SW-NTP clock is an absolute clock only, and is therefore fundamentally unsuitable for many applications.

Two methods of remote calibration over a network were given in our earlier work [5] for determining the period p of the TSC cycle. One of these was based on NTP servers, and shares some features with the method we propose here (software is available at [7]), however neither were robust enough for unsupervised use under real-world conditions, and neither dealt with offset measurement/calibration. The second aim of this paper is to provide algorithms both for rate synchronization (p measurement), and offset synchronization, which are not just accurate, but also highly robust. By this we understand an ability to provide accurate results even under less than ideal conditions, as well as reasonable performance even under 'catastrophic' conditions. In this regard the 'local clock is good' viewpoint is invaluable. If something goes wrong with the remote timestamp data, our reaction can legitimately be 'change nothing', rather than continuing to adjust the clock according to faulty or extremely variable timestamps. The algorithms we describe here take into account a variety of factors which our earlier work did not, and are substantially different, although based on the same core ideas.

The end result is a set of algorithms which, provided basic but reasonable conditions are satisfied, give reliable rate performance to better than 1 part in 10^7, or 0.1 *Parts Per Million* (PPM), and offset accuracy which under achievable conditions can be of the order of 0.1 ms. Using a nearby NTP server, over a period of 3 months a median offset error of around 0.03 ms was obtained, with an inter-quartile range of only 15μs. Although not sufficient to address all needs, particularly for high performance and high rate network monitoring where a specialist solution such as of use of locally synchronised DAG cards [8] is required, it considerably raises the bar for the accuracy, and more importantly, the reliability, of synchronization achievable inexpensively across a network. It is more than sufficient, for example, to enable the removal of the GPS receivers from the measurement machines in the RIPE NCC testbox network. Other potential applications abound. In networking they include network tomography, enhancing the performance of services and protocols based on one-way or round-trip delay, and improving the performance of adhoc and sensor networks.

This paper reexamines the synchronization problem in considerable depth and detail. Although at a high level the topic is not new by any means, and solutions exist, we believe that a thorough treatment, with careful attention to detail at each stage, was essential to arrive at a solution which delivers a significant step up in

[†]A more specific term which is sometimes used is CPU Clock Counter (CCC)

robustness and accuracy. The rate centric treatment is new and is the cornerstone of this ability.

2. PRELIMINARIES

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

2.1 Terminology

The natural or true clock, denoted simply by t, runs at a rate of 1 second per second, and has an origin $t = 0$ at some arbitrary instant. In practice one must use an imperfect clock, which we refer to as 'the clock', which reads $C(t)$ at the true instant t. The *resolution* of $C(t)$ is the smallest unit by which it is updated. The *offset* $\theta(t)$ of the clock is its error at true time t:

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

The *skew* γ is roughly 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

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

To refine the concept of skew consider the following general model

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

where the 'simple skew' γ is just the coefficient of the deterministic linear part, $\omega(t)$ being a remainder with no linear component obeying $\omega(0) = 0$ which encapsulates the deviations from the SKM. It can contain both deterministic and random components.

The *oscillator stability* [9] partially characterizes $\omega(t)$ via the family, indexed by time-scale 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 total rate error at time t when measured over time scale τ. The series $y_\tau(t)$ can be thought of as the mean skew γ arising from the SKM plus those random variations which impact at time scale τ.

Significant Time Interval	Interval Duration	Error rate, PPM 0.02	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 1: Absolute errors at key error rates and intervals. The most important examples are in bold.

To discuss the size of relative offset errors (rate errors), we use the dimensionless unit of *Parts Per Million* (PPM). Table 1 translates this into absolute error over key time intervals: $\Delta(\text{offset}) = \Delta(t) * (\text{rate-error})$. The significance of the error rates in the table, discussed in detail below, are i) target accuracy of 'local' rate estimates: 0.02PPM, ii) bound on clock stability: 0.1 PPM, For comparison, the typical skew of CPU oscillators from nominal rate is around 50PPM [9].

2.2 The TSC Clock

As described in the introduction, we propose a clock based on the TSC register which counts CPU cycles. The value of this register at time t is denoted by $\text{TSC}(t)$, and we set $\text{TSC}_0 = \text{TSC}(0)$. The construction of the clock $C(t)$ from the counter is based on the intuition of the simple skew model, for which the oscillator has a constant period p, implying that $t = (\text{TSC}(t) - \text{TSC}_0)p$. In practice we must obtain estimates, $\hat{p}$ of p, and $\widehat{\text{TSC}_0}$ of TSC_0. The definition of the clock $C(t)$ is therefore

$$\textbf{S}\text{KM:} \quad C(t) \equiv (\text{TSC}(t) - \widehat{\text{TSC}_0})\hat{p} = \text{TSC}(t)\hat{p} + C, \tag{5}$$

where the constant $C \equiv -\widehat{\text{TSC}_0}\hat{p}$ tries to align the origins of $C(t)$ and t, but with some error. It is easy to show that the error $p_\epsilon \equiv \hat{p} - p$ in the period estimate and $\text{TSC}_\epsilon \equiv \widehat{\text{TSC}_0} - \text{TSC}_0$ in the origin estimate leads to an offset of $\theta(t) = p_\epsilon/p \cdot t - \hat{p}\,\text{TSC}_\epsilon$, which, comparing to equation (2), identifies $\gamma = p_\epsilon/p = \hat{p}/p - 1$ and $\theta_0 = C(0) = \text{TSC}_0\,\hat{p} + C$.

As we explore below in detail, the SKM idea 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 function of γ, and so must be measured independently, that is the clock drift must be tracked. In practice we therefore use two forms of a corrected clock, depending on whether time differences are needed (valid up to SKM timescales, but also useful well beyond it, as described below), or absolute time:

$$\textbf{d}\text{ifference:} C_d(t) \equiv \text{TSC}(t)\hat{p}(t), \tag{6}$$

$$\textbf{a}\text{bsolute:} C_a(t) \equiv \text{TSC}(t)\hat{p}(t) + C - \hat{\theta}(t) = C(t) - \hat{\theta}(t), \tag{7}$$

where $\hat{p}(t)$ is the current period estimate, and $\hat{\theta}(t)$ is the current estimate of the offset of $C(t)$, which we then correct for. Only by defining two clocks in this way can we provide an absolute or 'offset calibrated' clock without disturbing the smooth rate of the underlying TSC clock, which is the basis of its excellent rate performance as described in [5]. The less accurate absolute clock should only be used for applications which truly require it. The difference clock, used to measure time differences, is much more accurate when used to measure intervals $\Delta(t)$ 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 however, clock drift is significant and the time difference will be more accurately measured using the absolute clock.

In implementations, care must be taken to avoid losing precision when recording timestamps. For example, the TSC register is typically 64 bits. If a 32 bit counter is used to manipulate it, overflow can occur after around 4 seconds on a 1Ghz machine (see [5] for a discussion on this point).

2.2.1 Timestamping

Even a perfect clock is of little use if one is unable to read it close to when the event of interest occurs. This is the issue of *timestamping*, and its optimisation is application dependent. Here, for the purpose of remote clock synchronization, the application is the timestamping of the arrival of NTP packets.

In [5] (see also [10]), this issue was explored in detail for the application of timing the arrival of packets on the Ethernet network interface (a 600Mhz Pentium PC was used). The timestamping solutions described range from using purely user-space code, through to the use of RT-Linux under Linux, a real-time operating system [11], which entirely avoids the scheduling problems which plague precision timing under multi-tasking operating systems. It was found that timestamping early in the driver-code for the network interface

card provided an excellent compromise of almost no scheduling problems (1 timestamp per 10,000, and then usually with an error under 1ms), and with timestamping noise (dominated by interrupt latency) of the order of at worst 15μs, whilst keeping the code simple and almost entirely at the user-level. In contrast, the standard `gettimeofday` system call to obtain timestamps from the SW clock under Linux suffers from much higher 'system' noise due to the above and other effects.

We adopt the same driver based timestamping approach here. The kernel-level code required consists of just a few lines to enable a raw timestamp (the TSC counter value) to be passed up from the driver to a user process, where it can be stored, processed, or converted to a time in seconds as required without hard time constraints. In [5] this was done under Linux by exploiting the existing API, in this paper it is done via a modified Berkely Packet Filter data structure under BSD Unix. The timestamping errors which remain, especially if of significant size (such as larger than 0.1 ms) will be seen as network delay by the generic filtering mechanisms we develop here, and thereby rejected or damped. For the same reason, if instead user-level timestamping were used, the algorithms would still work, albeit with higher estimation variance, as the errors will always increase round-trip times and therefore be seen as positive network 'noise'.

2.3 The NTP Server

Network Time Protocol (NTP) servers are networked computers whose clocks are deemed to be well synchronized. Different levels of synchronization accuracy are defined. We will be concerned only with *stratum-1* servers, whose clocks are synchronized by a local reference time source (using the GPS timescale, which can be converted to UNIX time). Three such are used in this paper. *ServerLoc* is in our laboratory on the same local network as the host. *ServerInt* is located in the same organization, but in a distinct network and GPS receiver in a different building. Finally, *ServerExt* is located over a thousand kilometres away in another city, and is synchronized by atomic clock. The distances between the host and the servers are given in table 2 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 Δ, which as discussed in detail in section 4.2 is the difference of the minimum one-way delays to and from the server.

Server	Reference	Distance	RTT	Hops	Δ
ServerLoc	GPS	3 m	0.38 ms	2	50μs
ServerInt	GPS	300 m	0.89 ms	5	50μs
ServerExt	Atomic	1000 km	14.2 ms	≈ 10	500μs

Table 2: Characteristics of the stratum-1 NTP servers.

Hosts wishing to synchronize their clocks can do so by running a NTP application which communicates with a NTP server via NTP packets. These are User Datagram Packets (UDP) with a 48 byte payload including four 8-byte Unix timestamp fields (90 bytes in total for the Ethernet frame which transports the datagram). The usual exchange works as follows (refer to figure 1). The ith NTP packet is generated in the host. Just 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 timestamps are perfect however due to clock and/or timestamping limitations. The *actual* times of the corresponding events we denote by $\{t_{a,i}, t_{b,i}, t_{e,i}, t_{f,i}\}$. The NTP payload also contains processed data related to estimated clock drift which we do not use, and server identity information which we plan to use as part of route change (level shift) detection in the future.

At the host, we do **not** use the usual timestamps made by the SW-NTP clock, but instead take separate raw TSC timestamps. We denote these by the symbols $T_{a,i}$, $T_{f,i}$ as above even though they are in 'TSC units', rather than seconds.

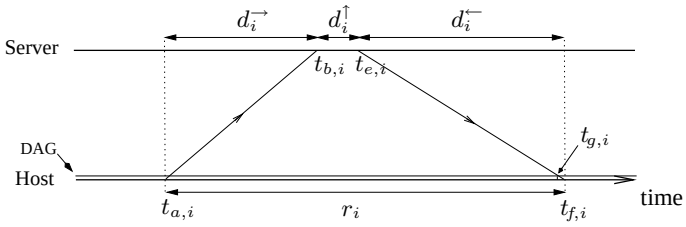

Figure 1: Timeline of the host-server exchange of the ith NTP packet. The components of forward path delay ($d_i^{\rightarrow}$), server delay ($d_i^{\uparrow}$), backward path delay ($d_i^{\leftarrow}$) and their sum, the round-trip time ($r_i = d_i^{\rightarrow} + d_i^{\uparrow} + d_i^{\leftarrow}$) are shown.

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 timestamping errors nonetheless make these unequal even for the server. Indeed, as servers are often just PC's, their timestamping may not have the quality of the driver based TSC timestamping of our host. This is the case here, and we discuss its consequences below.

As stated in the introduction, we rely on the normal flow of NTP packets between host and server, to minimize the disruption to normal system operations. For much of this paper we use a polling rate of 16 seconds, which is higher than the usual default, but which provides a detailed data base from which to examine both the clock and synchronization performance. In the last two sections we give examples of performance using a range of polling rates which includes the usual default values. A conservative polling rate is in keeping with the need to avoid placing excessive load on the network and the NTP server. In a more generic solution where the usual software clock would be entirely replaced by the TSC-NTP clock, the emission of NTP packets could be controlled, which would enable the synchronization performance to be further optimized, and warmup procedures simplified.

We provide a generic synchronization mechanism in this paper, but our work is nonetheless aimed at applications and users for whom accuracy matters. We therefore nominally assume that an initial effort has been made to find a nearby, reliable stratum-1 NTP server. The *ServerInt* is representative of such, as it has a RTT of the order of only 1ms, but is not on the local network. It also has the advantage of a verifiably symmetric route in the forward and reverse directions. This is a very important asset as we explain in detail in section 4.2. We believe that such a server can be readily found for the majority of tertiary educational institutions and commercial organizations 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.

2.4 Reference Timing

Validation of timing methods would not be possible without a reliable timing reference. We used a 'DAG3.2e' series measure-

ment card designed for high accuracy and high performance passive monitoring of 10/100 Mbps Ethernet, yielding a time stamping accuracy around 100ns [8]. The card was synchronized to the Trimbale Acutime 2000 GPS receiver, the same one used by *ServerLoc*, with the antenna permanently mounted on the roof of the building housing the laboratory.

The DAG card was positioned to timestamp the returning NTP packets via a passive tap on the Ethernet cable just before it enters the host's interface card. Unfortunately the DAG and TSC clocks are still timestamping different events, so that $t_{g,i} < t_{f,i}$. One component of this difference is that the DAG timestamps the first bit of each packet, whereas TSC timestamping occurs after a packet has fully arrived. Accordingly by $T_{g,i}$ we denote a DAG timestamp which has been corrected by the addition of the corresponding interval of $90 * 8/100 = 7.2\mu$s. The remaining difference includes the additional length of cable (negligible), the minimum processing time of the card, and the interrupt latency of the host. To estimate the size of these latter effects, we examined a histogram of the difference, with respect to i, of the measured offset discrepancy $T_{f,i} - T_{g,i}$. The (overwhelmingly) dominant mode, centered at zero as expected, has width 5μs. In addition to large departures due to rare scheduling errors, which are easy to detect and exclude if required, there are small but clearly defined side modes symmetrically located at 10 and 31μs from the origin. These smaller yet significant timestamping errors are due to interrupt latencies, and can also be reliably detected and corrected for. The final limit of the verifiability of the offset (but not rate) results is therefore of the order of 5μs.

The DAG timestamps are the basis of all the 'actual performance' results presented here.

3. DATA CHARACTERIZATION

Any synchronization algorithm must begin with a knowledge of the nature of the data collected. In this section we study the basic features of the key quantities, the offset of the TSC clock $C(t)$, the network delay, and also the delay at the NTP server.

3.1 The Clock

We examine the clock offset of the same 600Mhz CPU host 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.

To calculate the offset of the clock from the TSC counter timestamp $T_{f,i}$, we are immediately faced with the issue of prior rate estimation: without a value of $\hat{p}$ the clock $C(t) = \text{TSC}(t)\hat{p} + C$ cannot be calculated. In figure 2 we use $\hat{p} = 1.82263812 * 10^{-9}$ (548.65527 Mhz) for measurements made in the laboratory, and $\hat{p} = 1.82263832 * 10^{-9}$ (548.65521 Mhz) in the machine-room, and then calculate the offset via $\theta(t_{f,i}) = T_{f,i} * \hat{p} - T_{g,i}$ for each. These estimates 'detrend' (in fact they force the first and last offset values to be the same, normalised to be zero), facilitating an initial inspection of the (small) residual clock drifts, which depend on the temperature environment.

From the right plot in figure 2 it is clear that the SKM model fails over day timescales, as the residual errors are 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 residual offset error grows approximately linearly with time, suggesting that the SKM could be accepted and the above 'global' estimates replaced with a considerably more accurate local $\hat{p}$ values (the microsecond scale irregularities are due to timestamping noise in the host, as here corrected $T_{f,i}$ timestamps, described in section 2.4, were not used). We have found these observations to hold for all traces collected over many months. In [5] the same result was reported for a host in an airconditioned (but not temperature controlled) office environment over a continuous 100 day period.

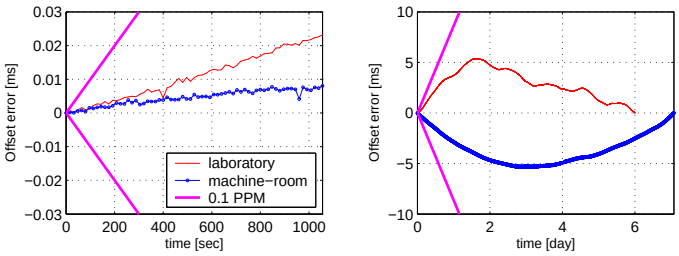

Figure 2: Offset variations $\theta(t)$ of $C(t)$ in two different temperature environments (each set to 0 at $t = 0$ for comparison) always fall within the cone corresponding to a steady error rate of $\gamma = 0.1$PPM. Left: over a 1000 [sec] period, Right: over 1 week (legend applies to both).

The above discussion is only an illustration of the underlying behaviour we must understand and deal with. In fact depending on the time scale 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 somewhat arbitrary prior rate estimate, we return to the concept of oscillator stability (equation (4)). A particular estimator of the variance of $y_\tau(t)$, known as the *Allan variance*[‡], calculated over a range of τ values, is a traditional characterization of oscillator stability [9]. We term the square root of the Allan variance the *Allan deviation*, and interpret it as the typical size of variations of time scale dependent 'rate'. A study over a range of time-scales 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.

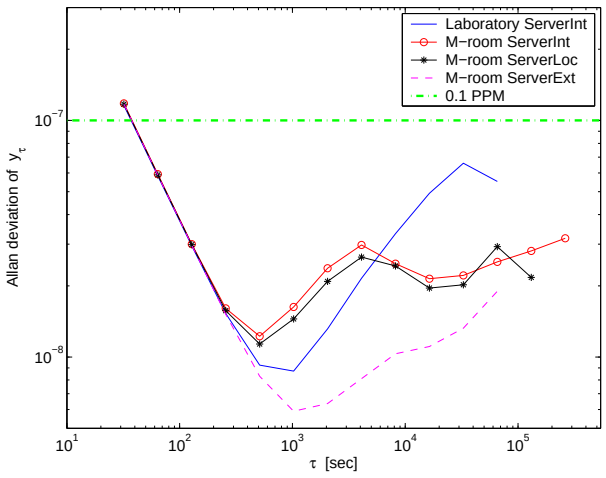

Figure 3: Allan variation plots. From small scales to around $\tau = 1000$ seconds, the SKM applies, and rate estimates are meaningful down to 0.01PPM.

[‡]This is essentially a Haar wavelet spectral analysis [12]

Four Allan deviation plots for the host oscillator 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 adds considerable spurious variation at small scales (due to the strong wavelet signatures of discontinuities).

Over small scales the plots show a consistent $1/\tau$ decrease, consistent with the results of [5], where a similar plot showed this linear zone to extend at least down to $\tau = 1$ second. This is exactly what we would expect if the SKM were true in the presence of system noise, and corresponds to equation (3) if $\omega(t)$ was considered to be white noise. 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 enter in, such as temperature variations arising over the day and the diurnal cycle itself. The curves begin to flatten as major new sources of variation cease at the weekly timescale, but remain below the horizontal line marking 0.1 PPM.

From figure 3 the picture emerges that the SKM model holds true for timescales up to around 1000 seconds. For any value within this range, we can interpret the corresponding value on the vertical axis as the level of precision to which the rate can be measured. The greatest precision is obtained at the minimum point, and is of the order of 0.01PPM. 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.

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 lie above each of those from the machine-room. This is indeed the case at large scales, as the temperature variations, and therefore the main source of rate fluctuations, are bounded. At intermediate scales however it was true in only 1 of 3 cases. We found this to be 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) which creates additional variability over a broad range of scales. The possibility of this being due to the airconditioning cycle was investigated through checking with a digital temperature logger making measurements every minute, with mixed results. The true cause (and possible elimination) of this effect is under investigation. It is not expected to be truly server dependent and may be linked to hardware and software controlling cooling fans in the host. The performance of the synchronisation algorithms reported below is *despite* the presence of this (possibly very unusual) noise component.

In conclusion, in three different temperature environments: uncontrolled laboratory, temperature controlled, and from [5], building-wide airconditioning, the SKM model holds over timescales up to 1000 seconds. Henceforth we use *SKM scale*, or τ^*, to refer to this value. Over larger timescales the model fails but the rate error remains bounded by 0.1 PPM. Indeed, to within this level of accuracy we can say that the SKM model holds over all time scales. This fact, and the values of the rate error bound and the SKM scale, are the fundamental hardware characterizations on which the synchronization is based. To the best of our knowledge this is the first time that synchronization algorithms have been built on a well defined hardware abstraction in this way.

The above measurements are consistent with the results of [9] stating that the clockstability of commercial PCs is typically of the order of 0.1 PPM. If a class of oscillators were used which were significantly different (for example less stable) then they would need to be characterised by calculating curves such as those in figure 3, to determine the two key metrics. As these appear as parameters in the synchronization algorithms, our clock solution would continue to work, with altered performance.

To characterise rate beyond τ^*, one cannot hope to measure an expected or stationary value, as it does not exist. If one measured a long term average rate over much larger timescales such as several weeks, its variability would itself be well under 0.1 PPM, however this apparent stability does not reflect a meaningful convergence, and does not enable more accurate synchronisation in any sense. We do **not** attempt to measure a (meaningless) 'long term' rate as such in this paper, however as described below, we **do** make used 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. We denote such estimates in section 5.2 by $\bar{p}$, where we discuss local rates in more detail. Such average rates may be used as surrogates for local rates, with an error which is bounded by 0.1 PPM.

3.2 Network and Server Delay

Following figure 1, we decompose the history of packet i as:

$$\text{Forward network delay } d_i^{\rightarrow} \equiv t_{b,i} - t_{a,i} \quad (8)$$

$$\text{Server Delay } d_i^{\uparrow} \equiv t_{e,i} - t_{b,i} \quad (9)$$

$$\text{Backward network delay } d_i^{\leftarrow} \equiv t_{f,i} - t_{e,i} \quad (10)$$

$$\text{Round Trip Time } r_i \equiv t_{f,i} - t_{a,i} = d_i^{\rightarrow} + d_i^{\uparrow} + d_i^{\leftarrow} \quad (11)$$

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 $d_i^{\leftarrow}(t_{e,i}) = T_{g,i} - T_{e,i}$ and $d_i^{\uparrow}(t_{e,i}) = T_{e,i} - T_{b,i}$ respectively. These time series appear roughly stationary, with a marginal distribution which seems consistent with a deterministic minimum value plus a positive random component. The main difference between them is that the server delay has much lower minimum and average values: microseconds rather than milliseconds. These observations make physical sense. The minimum in network delay could correspond to propagation delay, and the random component to queueing in network switching elements, which is not unexpectedly very small for such a short route, but which can takes 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 millisecond range. We formalise these observations in

$$\text{Forward network delay : } d_i^{\rightarrow} = d^{\rightarrow} + q_i^{\rightarrow} \quad (12)$$

$$\text{Server Delay : } d_i^{\uparrow} = d^{\uparrow} + q_i^{\uparrow} \quad (13)$$

$$\text{Backward network delay : } d_i^{\leftarrow} = d^{\leftarrow} + q_i^{\leftarrow} \quad (14)$$

$$\text{Round Trip Time : } r_i = r + (q_i^{\rightarrow} + q_i^{\uparrow} + q_i^{\leftarrow}) \quad (15)$$

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

4. SYNCHRONIZATION: THE SKM WORLD

In this section we examine simple synchronization ideas based on the SKM. We detail the weaknesses of these 'naive' approachs, which are addressed in subsequent sections. We use the first day of the same 7 day machine-room data set (July 4–10) used in the previous section.

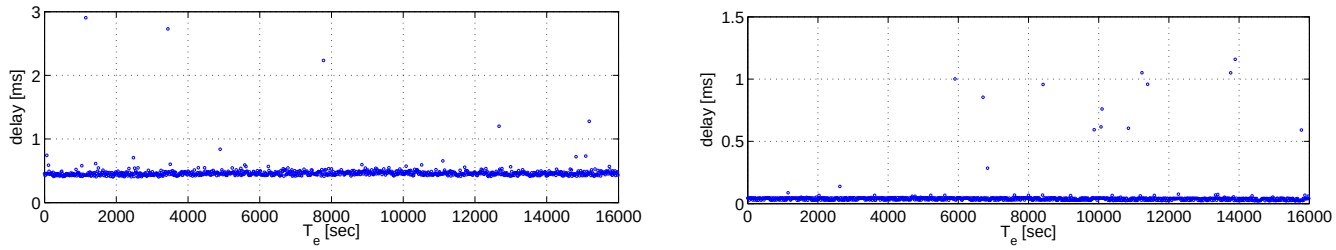

Figure 4: Examples of backward Network delay $d_i^{\leftarrow}$ (left) and Server $d_i^{\uparrow}$ delay (right) time series.

4.1 Rate Synchronization

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

$$p = \frac{t_{b,i} - t_{b,j} - (q_i^{\rightarrow} - q_j^{\rightarrow})}{\text{TSC}(t_{a,i}) - \text{TSC}(t_{a,j})} \tag{16}$$

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

$$\hat{p}_{i,j}^{\rightarrow} \equiv \frac{T_{b,i} - T_{b,j}}{T_{a,i} - T_{a,j}} \tag{17}$$

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

In figure 5 backward estimates normalised as $(\hat{p}_{i,j}^{\leftarrow} - \bar{p})/\bar{p}$ (where $\bar{p}$ denotes the 'detrending' $\hat{p}$ estimates from section 3.1) 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 $\hat{p}_g = (T_{f,i} - T_{f,j})/(T_{g,i} - T_{g,j})$ which show some timestamping noise ($T_{f,i}$ is not corrected here), but are not subject to network delay. We immediately see that the bulk of the estimates very quickly fall

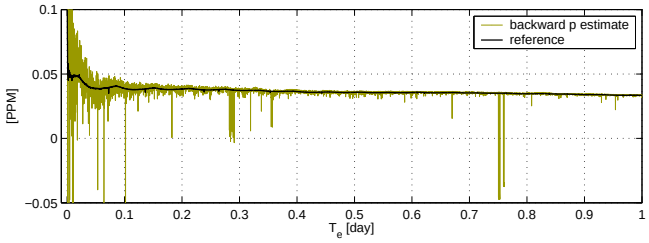

Figure 5: Naive per-packet rate estimates compared with reference measurements with steadily increasing Δ(TSC).

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 1 tells us that even when measured over a timescale of a day, the bound of 0.1 PPM will be broken when a combined network queueing delay exceeds only 8.6 ms.

If the SKM held exactly, these errors would eventually be damped as much as desired, however, we know that 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 the drift in the rate would then be masked. The estimate would appear to become increasingly stable, but would not be converging to any meaningful value, and both long term and medium term changes could be hidden. 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 naive estimates are therefore unreliable, as their errors, although *likely* to be small, can not be controlled or bounded.

4.2 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 $\gamma < 0.1$ PPM, the increase in offset error over a host-server round-trip time of 1ms is under 0.1ns (see table 1). 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 a far more complex task. 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 involving θ_i: $\theta_i = C(t_{a,i}) - t_{a,i} = C(t_{a,i}) - (t_{b,i} - d_i^{\rightarrow})$, and $\theta_i = C(t_{f,i}) - (t_{e,i} + d_i^{\leftarrow})$, from which θ_i cannot be recovered as the one-way delays cannot be independently measured. If we add these to obtain

$$\theta_i = \frac{1}{2}(C(t_{a,i}) + C(t_{f,i})) - \frac{1}{2}(t_{b,i} + t_{e,i}) + \frac{1}{2}\Delta + \frac{1}{2}(q_i^{\rightarrow} - q_i^{\leftarrow}), \tag{18}$$

the problem remains that the *path asymmetry*, $\Delta \equiv d^{\rightarrow} - d^{\leftarrow}$, is not measurable. There is in fact a fundamental problem of ambiguity here, differences in the θ_i due to $\Delta > 0$ are impossible to distinguish from true offset errors. However, clearly the 'causality' bound $\Delta \in (-(r - d^{\uparrow}), (r - d^{\uparrow})) \subset (-r, r)$ holds, i.e. we require the packet events at the server to occur inbetween those at the host. Note that r and $d^{\uparrow}$ *can* be measured as they each are time differences measured by a single clock.

In the absence of independent knowledge of Δ, a naive estimate based on equation (18) is

$$\hat{\theta}_i = \frac{1}{2}(C(t_{a,i}) + C(t_{f,i})) - \frac{1}{2}(T_{b,i} + T_{e,i}), \tag{19}$$

which implicitly assumes that $\Delta = 0$, and is equivalent to aligning the midpoints $(t_{b,i} + t_{e,i})/2$ and $(C(t_{a,i}) + C(t_{f,i}))/2$ of the server and host event times respectively. In figure 6 estimates obeying equation (19) are shown, along with reference values calculated as in section 3.1. 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. A histogram of the deviations of the estimates from their reference values is essentially

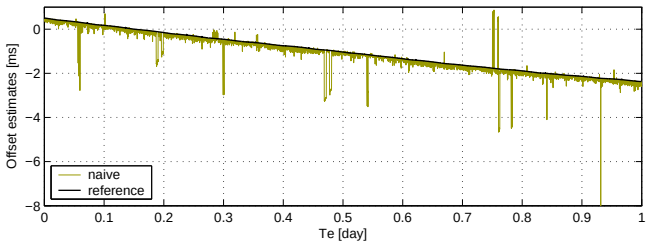

Figure 6: Naive per-packet offset estimates θ_i compared with reference measurements.

identical to a histogram of $(q_i^{\leftarrow} - q_i^{\rightarrow})/2$, and is biased towards negative values in this case because the forward path is more heavily utilised than the backward one.

The value of Δ 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 $\Delta \ll r$. This is in fact the case for *ServerLoc* and *ServerInt*, which we measured (see table 2) to be of the order of 0.05 ms. Estimating Δ however is non-trivial. With only a single reference clock positioned as in figure 1, we use $\Delta = d^{\rightarrow} - d^{\leftarrow} = r - d^{\uparrow} - 2d^{\leftarrow}$ which in terms of available timestamps reduces to $\hat{\Delta}_i = (T_{f,i} - T_{a,i}) * \hat{p} - 2T_{g,i} + T_{b,i} + T_{e,i}$. With i chosen to minimize r_i, we obtain measurements with close to minimal network and server delays, however timestamping errors remain, which are significant compared to the small size of Δ. Since $T_{a,i} < t_{a,i}$ and $T_{f,i} > t_{f,i}$, minimising r_i will also tend to minimize timestamping errors at the host. At the server however, although clearly $T_{b,i} > t_{b,i}$ (recall we assume the server is synchronised), the error $T_{b,i} - t_{b,i}$ is only bounded by the server delay $d^{\uparrow}$ and is not influenced by minimising r_i, and more seriously, the relationship between $T_{e,i}$ and $t_{e,i}$ is a priori unknown. Outliers in the reference backward delay values $\{T_{g,i} - T_{e,i}\}$ suggest that in fact $T_{e,i} > t_{e,i}$, in very rare cases by as much as 1ms, larger even than the RTT! In the future we will make a more reliable determination by repositioning the tap of the reference monitor. The results of the offset estimation algorithm described in section 5.3 provide an alternative, indirect, way of estimating Δ which agrees broadly with the values in table 2).

5. SYNCHRONIZATION: THE REAL WORLD

It is intuitively clear from figures 5 and 6 that the packets carrying large network delays can be detected, and therefore dealt with. We show how this can be achieved even for small network delays, and exploited to evolve the naive estimators into accurate ones. For clarity of exposition, we leave some on the considerations needed for an on-line implementation to the final section, and focus on the principles and performance of the underlying algorithms. In particular, when measuring offset we use a constant rate estimate made over the entire trace, rather than using the on-line evolving rate estimate $\hat{p}(t)$ (this makes little difference in practice), and for convenience we set $\text{TSC}_0 = 0$.

5.1 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, $\{d_i^{\rightarrow}\}$ and $\{d_i^{\leftarrow}\}$.

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 $p(t)$ 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 absolute *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 error at the host, which we take to be $\delta = 15\mu$s.

In contrast, one-way delays are measured by different clocks, so that the 'minimum' inherits the wander of $\theta(t)$ (recall figure 2), greatly complicating assessments of quality. On the other hand, 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. An alternative which retains the inherent advantages of RTT whilst in principle increasing the proportion of quality packets is $\{r_i - d_i^{\uparrow}\}$. However, although the server is synchronized, its timestamping errors serve only to add noise to the more reliable driver based timestamps made at the host from which we derive $\{r_i\}$.

5.2 Rate Synchronization

To bound the error on the estimate $\hat{p}(t)$, we use equation (17) 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 E^*. Equation (17) 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.

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 valid for meaningful filtering, allowing estimation to recommence at any time with no warm-up required.

Figure 7 plots the relative error of the resulting estimates with respect to the corresponding 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 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 $2E^*/(T_{g,i} - T_{g,j})$. To put this performance into context, note that for the measurement of time differences over a few seconds and below, the estimate $\hat{p}$ above gives an accuracy better than 1μs, which is the same order of magnitude as a GPS synchronized software clock, after only a few minutes! For example inter-arrival times, round-trip times, and the delay variation of network packets all fall into this category.

It is important to understand that the estimate $\hat{p}$ above is really that of the average rate 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

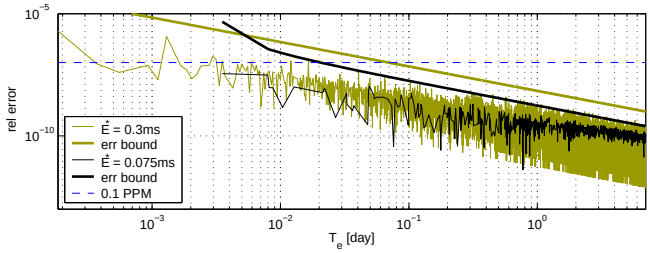

Figure 7: Relative error in $\bar{p}$ estimates for $E^* = [20, 5] \cdot \delta = [0.3, 0.075]$ ms. Errors fall below 0.1 PPM and remain there.

down to accuracies of $\epsilon = 0.01$ PPM, over scales below τ^*. However, there is no need for local rate estimates in order to obtain $\hat{p}$, and $\hat{p}$ is sufficient to support filtering and both the difference and absolute clocks. This is an advantage since the estimation of local rates is much more difficult due to the small number of samples available. However, there are two reasons why their measurement may be worthwhile: (i) they could extend the time intervals 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 the absolute clock $C_a(t)$.

In order to measure local rate to an accuracy close to the optimal $\epsilon = 0.01$ PPM however, given that with *ServerInt* point error values descend into noise for $E^* < 0.1$ ms, it is necessary to use intervals of width greater than τ^*, in fact $\bar{\tau} \equiv 5000\text{sec} = 5\tau^*$ in this case (one can even use larger values, $\bar{\tau} = 20\tau^*$, to reduce estimation variance, although results are insensitive). Thus, such estimates are again not truly local in the sense of the SKM but are also averages, this time however over nearby local values. As changes in local rate occur slowly, these 'trends' are nonetheless local in comparison with the time-scales used in the estimate $\hat{p}$. We denote the corresponding quasi-local period values by $p_l(t)$.

The algorithm calculates a local value for each packet k over a window of effective width $\bar{\tau}$ stretching back from $t = t_{f,k}$. 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 $\bar{\tau}/W$, $\bar{\tau}(W-2)/W$, and $2\bar{\tau}/W$ respectively (the far window has length twice the expected $\bar{\tau}/W$ so that the actual window will begin at $t = t_{f,k} - \bar{\tau}$ on *average*). In each of the near (index i) and far (index j) windows, the packets with the lowest point errors are selected, and used in equation (17) 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^* (which we choose to be $\gamma^* = 0.05$ 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.

To ensure that any unexpected failures of the estimation procedure cannot force the rate estimates to contradict the known physical behaviour of the hardware, if the relative difference between two succesive rate estimates ever exceeds some multiple of the 0.1 PPM rate bound, we use $3 * 10^{-7}$, then the previous value will be duplicated as above: $\hat{p}_l(t) = \hat{p}_l(t_{f,k})$. This guarantees that the local rate estimate cannot vary too wildly no matter what data it receives. One situation where this is needed is when the server timestamps themselves are in error. This actually occurred in our data set, as shown in the next section.

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 high level sanity checking. Consequently, we found that the results are not sensitive to the exact value of W (we use $W = 30$).

The algorithm closely tracks the corresponding reference rate values made over the same timescale. Using the same data as in figure 7, with $\gamma^* = 0.05$ PPM, $\bar{\tau} = 5\tau^*$ and $W = 30$, over 99% of the relative descrepancies 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.

5.3 Offset Synchronization

Our aim is to estimate $\theta(t)$ for arbitary 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 to improve estimates. In particular this makes good performance immediately following long periods of congestion or sequential packet loss much easier to achieve.

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*, and 169 of the 113401 packets were lost (or failed to have matching reference timestamps). We also present comparative results from a week long trace using *ServerLoc* where 299 packets were unavailable, and a trace 2.7 weeks long using *ServerExt* where 666 packets were missing.

For the estimate $\hat{p}$, 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 a 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: $E_i^T = E_i + 10^{-7}(C_d(t) - C_d(T_{f,i}))$. This however is overly pessimistic as the residual rate error (from the $\hat{p}$ used to calculate $\hat{\theta}_i$) is more likely to be of the order of ϵ (from section 5.2). We therefore estimate the total error as $E_i^T = E_i + \epsilon(C_d(t) - C_d(T_{f,i}))$.

(ii) First we consider only those packets which fall into a SKM related 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 $w_i = \exp(-(E_i^T/E)^2)$, which has a maximum of 1, and 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 weighted sum

over the window:

$$\hat{\theta}(t) = \sum_i w_i \hat{\theta}_i / \sum_i w_i, \tag{20}$$

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

$$\hat{\theta}(t) = \Big(\sum_i w_i (\hat{\theta}_i - \hat{\gamma}_l \left(C_d(t) - C_d(T_{f,i}) \right)) \Big) / \sum_i w_i \tag{21}$$

where $\hat{\gamma}_l = \hat{p}_l(t_{f,i})/\bar{p} - 1$ is the estimate of the residual rate error relative to $\bar{p}$ (implicitly already present in $\hat{\theta}_i$).

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 taken. In the case where this is at the last packet (we always evaluate the offset at packet times), this gives

$$\hat{\theta}(t) = \hat{\theta}(t_{f,i}) \tag{22}$$

$$\hat{\theta}(t) = \hat{\theta}(t_{f,i}) - \hat{\gamma}_l * (C_d(t) - C_d(T_{f,i})), \tag{23}$$

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

(iv) Just as for local rates, 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 threshold then the most recent trusted value will simply be duplicated, for example $\hat{\theta}(t) = \hat{\theta}(t_{f,i})$ if the last such was at packet i. We set the threshold at $E^s = 1$ms, which is orders of magnitude beyond the expected offset increment between neighboring packets. It is very important that such a simple thresholding be used only as a sanity check, meaning 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 will be given in the next section.

An example of estimates made 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 30μs away from the reference values.

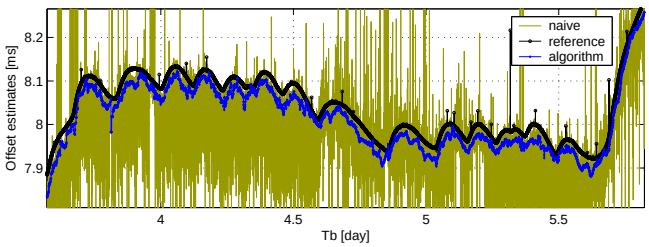

Figure 8: Time series of $\hat{\theta}_i$ using the algorithm (without local rates) against reference values, with naive estimates in the background. The isolated jumps in the reference values are due to timestamping noise in the (uncorrected) $T_{f,i}$.

In figure 9 the central curve shows the median of the difference of the estimates from the corrected reference values $\theta_g(t)$, as a function of τ', calculated over the entire 3 weeks. It is around 28μs over a wide range of window sizes, and the inter-quartile range (the lines above and below) 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. Given that our best current evaluation of the path asymmetry is around $\Delta \approx 50\mu$s, which implies an ambiguity in offset of $\Delta/2 \approx 25\,\mu$s (equation (18)), and that timestamping issues limit the verifiability of our results to around 5μs in any case, figure 9 suggest that the algorithm is working very well in eliminating the variability in network delay. Essentially identical results were obtained over the 3 month period of continuous monitoring (with gaps, see section 6) using *ServerInt*, of which the current 3 week trace is a subset.

Figure 9(a) also compares the estimation with and without the use of local rates. The differences are marginal, 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. Since $\tau' = \tau^*/2$ here, the fact that using local rates makes a negligible difference is consistent with figure 9(a).

We also performed sensitivity analyses with respect to the aging rate parameter ϵ, and the local rate window width $\bar{\tau}$. 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 (see 'lost packets' in section 6.1), or when parameters have not been optimised for a given set of circumstances. For example we found that using local rate also helped stabilise estimates when E was selected too low.

We next examined the performance of the algorithm with respect to polling period. We do not give results using the local rate refinement (they were very similar). 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 $\epsilon = 0.02$ PPM and vary the polling rate. Figure 9(c) shows again that the sensitivity is very low. In particular the median error (thick central line) 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 essential that NTP servers not be excessively loaded. It also reduces the memory and computational burden at the host.

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 $\bar{\tau} = 5\tau^*$, and a polling period of 64. We see the reduction in variability when moving from the laboratory into the more stable machine room, 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. As before, the

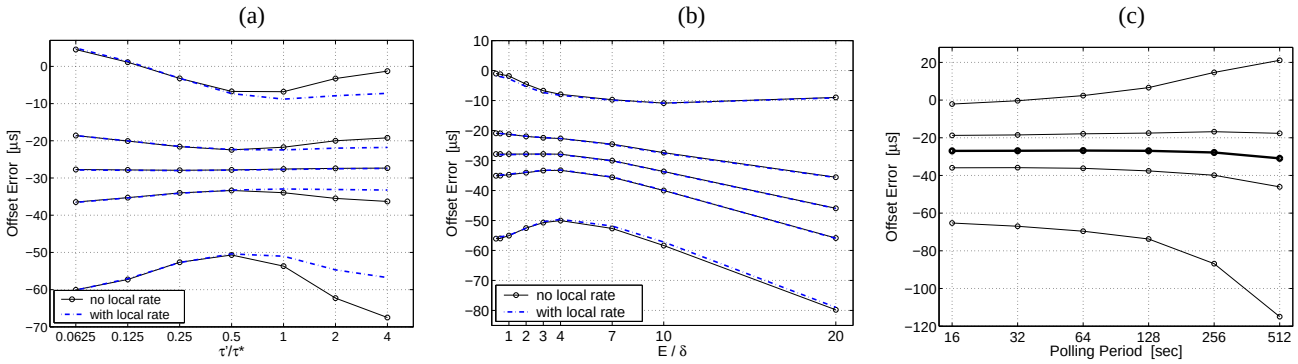

Figure 9: Sensitivity analysis of offset estimate errors with respect to key parameters: (a) window size τ', ($E = 4\delta$, with and without local rate: $\bar{\tau} = 20\tau^*$) (b) quality assessment E ($\tau' = \tau^*/2$, with and without local rate: $\bar{\tau} = 20\tau^*$), and (c) polling period ($\tau' = \tau^*$, $E = 4\delta$ without local rate). From top to bottom the curves are the 99%, 75%, 50% (the median) 25% and 1% percentiles of the empirical errors $\hat{\theta}(t) - \theta_g(t)$. The sensitivity is very low in each case.

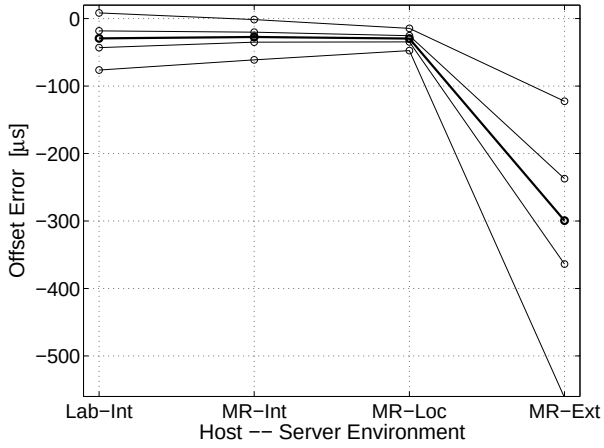

Figure 10: Performance over four different operating environments (same data sets as figure 3). Top to bottom: 99%, 75%, 50% 25% and 1% percentiles of the empirical errors $\hat{\theta}(t) - \theta_g(t)$.

error is approximately $\Delta/2$ using the values from table 2, much smaller than the round-trip time of 14.2 ms. The increased variability is due to the higher noise resulting from the larger number of hops, making quality packets much rarer. We should recall at this point that here we are testing the algorithm in the extreme case of a server that is much further away in all senses than necessary.

6. A ROBUST WORKING SYSTEM

In this section we address additional issues that are important to complete the algorithms, and present results from a working on-line implementation written in C. We also describe additional robustness issues and our solutions to them. For space reasons, we cannot present detailed flow diagrams. Full details will be made available at [7] by way of a heavily structured and commented reference C implementation.

6.1 Additional Issues

Warmup Phase: Each part of the estimation algorithms which uses a window, or requires a minimum quality to first be reached, needs a 'warm up phase' during which the full window of data and/or quality is established for the first time. We describe briefly the main features of our approach, under the assumption that estimates must be made available immediately (offset and the absolute clock from the first packet, and $\hat{p}$ and the difference clock from the second).

- $\hat{r}(t)$: Requires no warmup, but point error estimates based on it should not be trusted for small sample size → define warmup window T_w calibrated in *number of RTT samples*.
- $\hat{p}(t)$: In T_w, use a local rate type algorithm to exploit $\Delta(t)$ increase whilst managing delay errors, where the width of the near and far windows is initially 1 and grows as $\Delta(t)/4$. The first estimate is just the naive estimate $\hat{p}_{2,1}$. After T_w, the initialisation of section 5.2 applies. There are interactions with windowing and level shift detection which must be carefully managed.
- $p_l(t)$: Just a refinement, so no special warmup → only activated once a full window $\bar{\tau}$ available *after* T_w.
- $\hat{\theta}(t)$: In T_w, the quality assessment parameter E is increased, and the SKM window is filled up, otherwise no change. The first estimate is just the server timestamp $T_{b,1}$.

Windowing: As discussed at the end of section 4.1, despite the high stability of the CPU oscillator, conditions may change over time for many reasons, and so one must eventually forget the past. This is also necessary in practice in order to limit the amount of per-packet historical data that *may* be stored. We implement this as a top level 'sliding' window of width T, updated at intervals of $T/2$ to limit the computational burden. We set T to 1 week below.

The top level window only impacts directly on $\hat{p}(t)$ and the minimum estimate $\hat{r}(t)$, as $\hat{p}_l(t)$ and $\hat{\theta}(t)$ are already based on limited (and much smaller) windows. When the window reaches full size, the oldest half of the data is discarded and the updates occur as follows:

- $\hat{r}(t)$: This operation is performed first. A new value is calculated (actually *was* calculated on-line) based on the full set (now $T/2$ wide) of historical data. If an upward level shift(s) occurred during the window (see below), then the new value will be based only on values beyond the last detected shift point.
- $\hat{p}(t)$: If the first packet j of the current pair defining the estimate was discarded, then it is replaced by the first packet in the new window of similar or better point quality. The total quality using the new pair (which will be high as $\Delta(t)$

will be of the order of $T/2$) is then calculated, and $\hat{p}(t)$ is updated if it exceeds the current quality.

Re-evaluation of Point Errors: Whenever the estimates $\hat{p}(t)$ and $\hat{r}(t)$ are updated the past point errors effectively change, which impacts on estimates of all quantities. For the purposes of future estimates the new point errors are used, however for simplicity we do not retrospectively alter estimates already calculated. For rate estimates, this means that the current packets j and i used in the estimate may actually be rejected if they were assessed again. We do not do so: they remain the current indices, however the quality of the rate estimate is reassessed and used as normal in the algorithms.

The above principle holds true regardless of the cause of a change in $\hat{p}(t)$ and $\hat{r}(t)$: normal on-line updating, window change, or level shift (see below).

Clock Offset Consistency: Updating $\hat{p}(t)$ effectively redefines the $C(t)$ and results in a jump in the offset estimate. Although this is cancelled in the absolute clock $C_a(t)$ of equation (7), we preserve the continuity of $\hat{\theta}(t)$ by adding $\text{TSC}(t^-)(\hat{p}(t^-) - \hat{p}(t))$ to the constant C, effectively redefining the clock again so it agrees with the old one just before the update.

Lost Packets: Any lost packets are simply excluded from the analysis. The windows: $(\tau^*, \tau', \bar{\tau}, T, T_s)$, although nominally defined as time intervals, are in practice based on maintaining a fixed number of packets calculated by dividing the nominal interval size by the known polling period. As the proportion of lost packets is typically very low, this results in very little drift in the control of time-scale, and greatly simplifies the details of the algorithms.

To guard against the possibility of loss of time-scale control when using the local rate refinement, the $T_{f,i}$ timestamps are used to monitor the time since the previous packet. If it is too large compared to the local rate scale (we use a threshold *gap* size of $\bar{\tau}/2$) then the local rate is deemed out of date and is not used. Furthermore, if in addition the quality of packets in the window is too low (in the precise sense defined in point (iii) of the algorithm in section 5.3), the estimate is formed slightly differently in order to increase the importance of the new data: a weighted sum is made between the new naive estimate, using its point error, and the most recent $\hat{\theta}(t)$, using an aged form of its estimated error.

Figure 12 summarises the performance of the full on-line algorithm over a continuous 3 month period for standard default polling periods of 64 and 256 seconds. During this time, two major gaps occured in our trace collection, of duration around 1.5 hours and 3.8 days, in addition to a server error event. Despite these anomalies, the performance remained uniformly very good to excellent, and did not change greatly with polling rate.

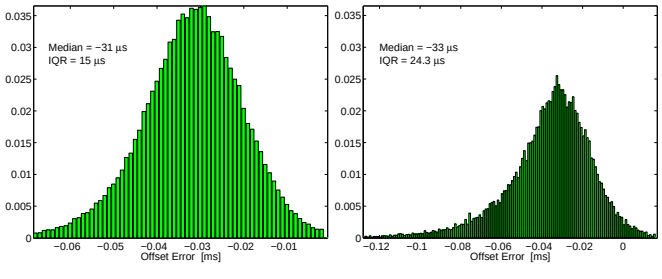

Figure 12: Performance of offset errors over a 3 month period using *ServerInt*, with polling period: 64 (left) and 256 (right) seconds. The histograms shows exactly 99% of all values (note scale change). $\tau' = 2\tau^*$

.

Unforseen Events: There is a need to ensure robustness to unforeseen and extreme events including loss of connectivity, a change in the route to host or a change in server, unexpected change of the temperature environment, or even errors in the server's clock.

As it is impossible to forsee all scenarios, it is important that the robustness is built in in very generic ways, rather than via dedicated algorithmic branches. Many such robustness features have already been described in the previous section. An important additional consideration which we have yet to address is that of level shifts in the data, which can occur due to route or server changes. Examples of each of these occured in our experimental data. Due to its importance and relative complexity we devote the next subsection to this topic.

Figure 11 zooms in on extreme events which occurred during a continuous measurement period which extends that of figure 12 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 (simulating server unavailability). Figure 11(b) shows the impact of a server error lasting a few minutes, during which $T_{b,i}$ and $T_{e,i}$ were 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.

6.2 Robustness to Level Shifts

By *level shift* we mean principally a change in any of the minimum delays $d^{\rightarrow}$, $d^{\uparrow}$ or $d^{\leftarrow}$ (see equation (12)), and hence r, which results in a change in minimum level in some or all of the observed $d_i^{\rightarrow}$, $d_i^{\uparrow}$, $d_i^{\leftarrow}$ or r_i. We will continue to restrict ourselves to RTT as the basis of packet quality measurement, and therefore level shift detection.

We first discuss the key issues governing level shifts.

Asymmetry of shift direction:
These are fundamentally distinct and must be treated differently:
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 the algorithms must already be 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.

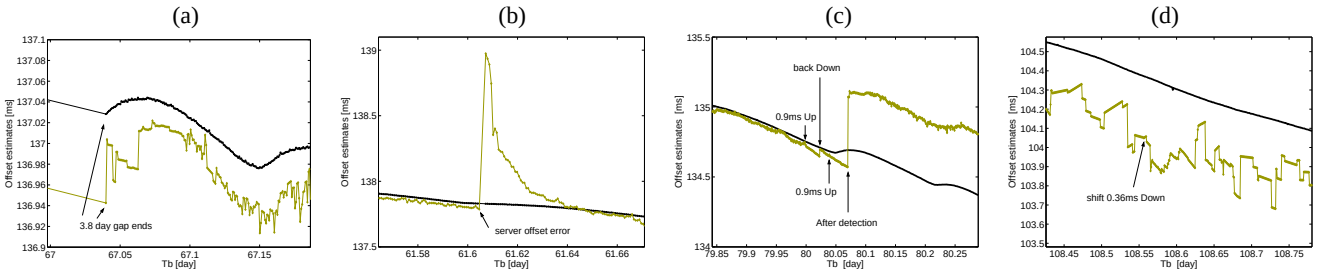

Figure 11: Performance of algorithm under extreme conditions (smooth thicker curves are reference offsets, variable curves with arrows are estimated offsets): (a) loss of data over 3 days, (b) level shift error of 150 ms in the server clock (triggers sanity check), (c) artificial temporary and permanent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

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 $\Delta(t)$ and windowing will improve packet qualities again.

Up:

Detection: Based on maintaining a local minimum estimate $\hat{r}_l$ over a sliding window of width T_s. Unambiguous detection is difficult and the consequences of incorrect detection serious. We therefore choose T_s large, $T_s = \bar{\tau}/2$, and detect a shift (at $t = C(T_{f,i}) - T_s$) if $|\hat{r}_l - \hat{r}| > 4E$.

Reaction: First update $\hat{r} = \hat{r}_l$ (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 T_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 T_s later, resulting in a jump in subsequent offset estimates (the original on-line estimates, not the recalculated ones, are shown). Most of this jump is due not to estimation difficulties resulting from the shift but to the change in Δ of $0.9/2 = 0.45$ ms, as the shifts were induced in the host→server direction only. In constrast, the natural permanent shift in figure 11(d) occurs equally in each direction, so that Δ does not change, and is also downward, so that detection and reaction are immediate. The result is no observable change in estimation quality, the shift is absorbed with no impact on estimates and with no level-shift specific actions being taken.

7. CONCLUSION

We have presented a detailed reexamination of the problem of inexpensive, convenient yet accurate clock synchronization for networked PCs. It is based on a thorough understanding of the stability of the CPU oscillator as a timing source, accessible via the TSC register. Using the NTP server network together with TSC timestamps, we showed how to calibrate a clock based on the TSC both in terms of rate, essential to the measurement of time differences, and offset, that is absolute time. We explained the importance of maintaining distinct software clocks for each of these distinct tasks. Our approach is new in several respects, notably in its being explicitly rate rather than offset-centric, grounded in a meaningful time-scale analysis of the drift in the underlying hardware, and through the development of filtering algorithmns built explicitly on the above with relatively few additional ad-hoc elements. Together these allow considerably higher levels of accuracy and more importantly, reliability.

Using months of real data from 3 different NTP servers, we provided a systematic and thorough testing of the algorithm, its absolute performance and sensitivity to parameters. Using a nearby server, we were able to reliably absolutely synchronize to the order of 30μs, and obtain rate accuracy of around 0.02 PPM. We demonstrated the robustness of the techniques to such effects as packet loss and loss of server connectivity, changes in server, network congestion, temperature environment, timestamping noise, and even faulty server timestamps. The approach should allow many applications requiring accurate and reliable timing, in particular network measurement, to do away with the cost of hardware based synchronization, such as using GPS receivers. The stability analysis, filtering principles, and algorithms discussed here also provide a firm basis for a new generation of software clock, with NTP server based synchronization, for networked PCs and other devices. They could easily be adapted for use with other kinds of oscillators, and other kinds of reference time servers. They could also be used to improve the existing SW clock, however the simplest way to do so is to simply replace it with a complete TSC-NTP clock solution.

Both the RIPE NCC Test Traffic Measurement project [6], and CAIDA's Skitter project [13], have agreed to trial the methods described here, the former to enable the expensive GPS component to be replaced (or made more reliable by replacing the SW-GPS with a 'TSC-GPS' clock), and the latter to replace the existing SW-NTP solution which was found to be unreliable. In future work we hope to collaborate with RIPE NCC and CAIDA to benchmark the performance of their implementations, and comprehensively compare them to those of the original clocks. Software will be made available for download, for trial and use by the community at [7]. It is expected that a version will be available in time for IMC-2004.

8. ACKNOWLEDGEMENTS

The BSD timestamping code was developed and implemented by Michael Dwyer. We thank David Batterham and Karl Cirulis at the University of Melbourne for their assistance with the temperature logging. This work was supported by Ericsson.

9. REFERENCES

[1] D.L. 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. Liao et al., “Experience with an adaptive globally synchronizing clock algorithm,” in *Proc. of 11th ACM Symp. on Parallel Algorithms and Architectures*, June 1999.

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

[4] Jonas Andren, Magnus Hilding, and Darryl Veitch, “Understanding end-to-end internet traffic dynamics,” in *IEEE Global Telecommunications Conference (Globecom’98)*, Sydney, Australia, Nov. 1998, vol. 2, pp. 1118–1122.

[5] Attila Pásztor and Darryl Veitch, “PC based precision timing without GPS,” in *Proceeding of ACM Sigmetrics 2002 Conference on the Measurement and Modeling of Computer Systems*, Del Rey, California, 15–19 June 2002, pp. 1–10.

[6] “RIPE NCC Test Traffic Measurements,” `http://www.ripe.net/ttm/`.

[7] Attila Pásztor and Darryl Veitch, “Software infrastructure for accurate active probing,” `http://www.cubinlab.ee.mu.oz.au/probing/software.shtml/`.

[8] Jörg Micheel, Ian Graham, and Stephen Donnelly, “Precision timestamping of network packets,” in *Proc. of the SIGCOMM IMW*, November 2001.

[9] D.L. 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.

[10] Attila Pásztor and Darryl Veitch, “A precision infrastructure for active probing,” in *Passive and Active Measurement Workshop (PAM2001)*, Amsterdam, The Netherlands, 23–24 April 2001, pp. 33–44.

[11] Victor Yodaiken, “The RTLinux Manifesto,” Tech. Rep., Department of Computer Science, New Mexico Institute of Technology, 1999, available at http://www.rtlinux.org.

[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] “Cooperative Association for Internet Data Analysis (CAIDA) Skitter project,” `http://www.caida.org/tools/measurement/skitter/`.