

The key to synchronizing clocks over networks is taming delay variability.

BY JULIEN RIDOUX AND DARRYL VEITCH

Principles of Robust Timing over the Internet

EVERYONE, AND MOST everything, needs a clock, and computers are no exception. However, clocks tend to ultimately drift off, so it is necessary to bring them to heel periodically through synchronizing to some other reference clock of higher accuracy. An inexpensive and convenient way to do this is over a computer network.

Since the early days of the Internet, a system collectively known as NTP (Network Time Protocol) has been used to allow client computers, such as PCs, to connect to other computers (NTP servers) that have high-quality clocks installed in them. Through an exchange of packet timestamps transported in NTP-formatted packets over the network, the PC can use

the server clock to correct its own clock. As the NTP clock software, in particular the `ntpd` daemon, comes packaged with all major computer operating systems, including Mac OS, Windows, and Linux, it is a remarkably successful technology with a user base on the order of the global computer population.

Although the NTP system has operated well for general-purpose use for many years, both its accuracy and robustness are below what is achievable given the underlying hardware, and are inadequate for future challenges. One area where this is true is the telecommunications industry, which is busy replacing mobile base-station synchronous backhaul systems (which used to provide sub-microsecond hardware-based synchronization as a by-product) with inexpensive asynchronous Ethernet lines. Another is high-speed trading in the finance industry, where a direct relationship exists between reducing latencies between exchanges and trading centers, and the ability to exploit these for profit. Here accurate transaction timestamps are crucial. More generally, timing is of fundamental importance because, since the speed of light is finite, the latencies between network nodes are subject to hard constraints that will not be defeated by tomorrow's faster processors or bandwidth increases. What cannot be circumvented must be tightly managed, and this is impossible without precise synchronization.

The Clockwork

When the discussion turns to clocks, confusion is often not far behind. To avoid becoming lost in the clockwork, let's define some terms. By t we mean true time measured in seconds in a Newtonian universe, with the origin at some arbitrary time point. We say that a clock C reads $C(t)$ at true time t . Figure 1 shows what some example clocks read as (true) time goes on. The black clock $C_p(t)$ is perfect: $C_p(t) = t$, whereas the blue clock $C_s(t) = C_0 + (1 + x)t$ is out by C_0 when $t = 0$. In fact it keeps getting worse as it runs at a constant but overly



fast rate, the rate error being the *skew* x . The red clock $C_d(t) = t + E(t)$ is more realistic, its error $E(t)$ varies with time, and the clock is said to *drift*. Far from aimless, drift is in fact very closely related to temperature changes. Figure 6 (discussed later in greater detail) gives a close-up view of the drift $E(t)$ (black curve) for an unsynchronized clock in a Pentium PC over a two-day period.

At the heart of every clock is hardware. In PCs this is an oscillator found on the motherboard. The oscillator “ticks” at a nearly constant rate—in fact typically varying on average by only 0.1 ppm (parts per million). Hardware counters such as the HPET (High Performance Event Timer) count these ticks, giving the operating system access to a slowly drifting source of timing. From such a counter, a clock can be defined that, roughly speaking, converts counter tick units into seconds and adds a constant to set the time origin:

$$C(t) = C_0(t) + p(t) \cdot \text{HPET}(t)$$

where $p(t)$ is the (slowly time varying) period in seconds of the HPET counter. The role of the clock synchronization algorithm is to set and regularly update the values of C_0 and $p(t)$ to reduce the drift, or error $E(t) = C(t) - t$, as much as possible.

Enter the Internet. The sync algorithm has no way of correcting the drift of $C(t)$, inherited from the counter, without calling on some independent expert: a reference timing source, or master. Attaching extra hardware is possible but expensive, and is pointless unless it is reliable. A good-quality GPS with consistently good satellite visibility can cost many thousands of dollars by the time you have persuaded the building supervisor to mount it on the roof and run a cable to your office. Oh, and it will take you six months to make it happen—minimum. An atomic clock is even more expensive and still

requires GPS to avoid drift on weekly timescales and beyond. In contrast, synchronizing over the network can be done with no lead time, no dollars, and not much effort.

The Key Challenge: Delay Variability

Clock synchronization is trivial in principle. You ask the expert what the time is, he looks at his clock, tells you, and you set your watch. Simple. The problem is that each of these steps takes time, and on the Internet, these delays can be large, unpredictable, and highly variable. The key challenge—almost the only one—for a synchronization algorithm, is to be able to cope with these variations, to robustly and precisely negate the errors caused by delay variability.

The delays suffered by timing packets are generated in both the host computer and the network (and the server, but we will be generous and assume a

perfect reference clock). In the host, delays arise from the NIC (network interface card) behavior, operating-system process scheduling, and the architecture of the packet timestamping code, and are on the order of several tens of microseconds. In the network, they are caused by queuing in network elements such as IP routers and Layer 2 switches, and vary from a large fraction of a millisecond over a Gigabit Ethernet LAN to possibly hundreds of milliseconds

Figure 1. The time according to some simple clocks as a function of true time.

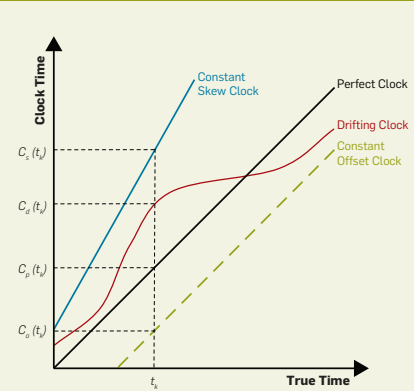


Figure 2. Bidirectional exchange of timing packets between host and server. A symmetric route would have $A = d^{\uparrow} - d^{\downarrow} = 0$, but that is not the case here. The RTT is $r = d^{\uparrow} + d^{\downarrow}$.

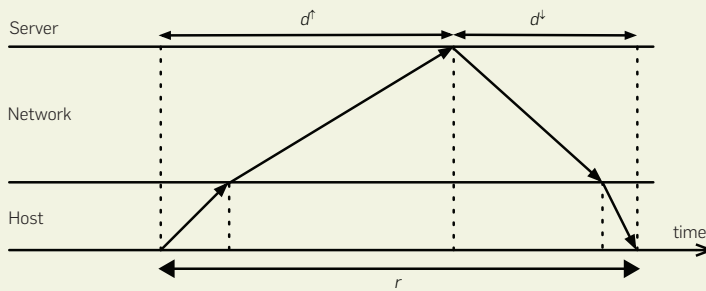
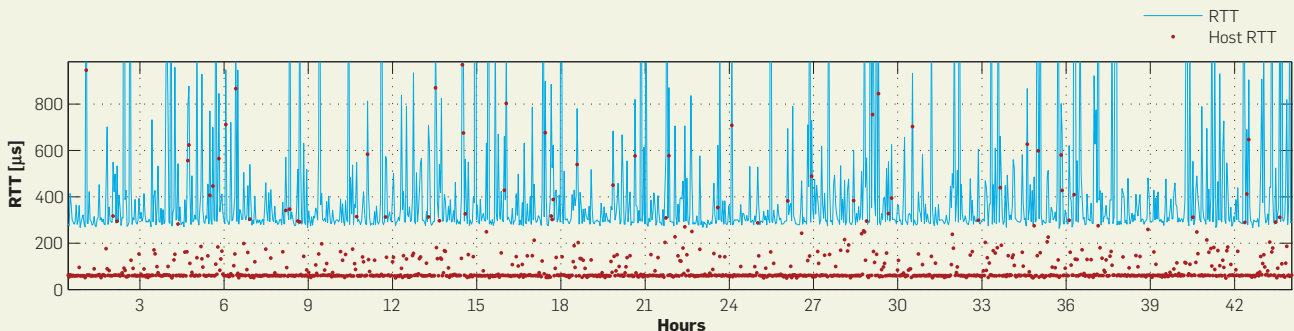


Figure 3. The RTT of timing packets over a congested LAN, and the host component separated out. Each component can be modeled as a minimum value plus a positive noise.



when network congestion is high and the server more than a few hops away.

In a bidirectional synchronization paradigm, which we focus on here (the alternative is to broadcast from the server only), timing packets are exchanged in both directions (see Figure 2). The OWD (one-way delay) in both the forward (host to server: d^{\uparrow}) and backward (server to host: d^{\downarrow}) directions have their own separate host and network components. Adding the OWDs in each direction yields the host→server→host RTT (round-trip time).

Figure 3 provides an example of the RTTs for timing packets when both the host and server are on a LAN, as well as the host component by itself. Each varies well above its minimum value, which translates to the addition of a large “noise” that the sync algorithm must somehow see through.

The Right Clock for the Job

The three main uses of a clock are to:

- Establish a strict order between events,
- Measure time durations between events, and

► Define a universally comparable and triggerable event label, the “time of day.”

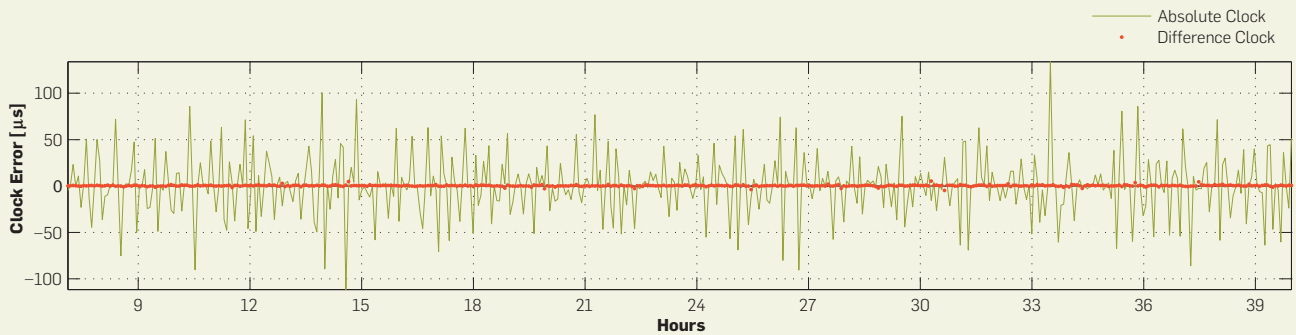
A perfect clock is ideally suited for each of these purposes, but in the real world, hardware and software limitations mean that the best performance is achieved by specialist clocks. The need to choose the right clock for the job is crucial and is not widely understood.

When one thinks of a clock, it is the third use that typically comes to mind: a clock that tells the *absolute* time. Synchronizing such a clock, however, is inherently difficult in the face of delay variability. As a result, the accuracy of an absolute clock $C_a(t)$ can be low, highly variable over time, and a slave to network conditions. Absolute clocks must also support the ability to be jump-reset, including backward. Though this may be rare, it makes them less than ideal for use 1.

A better choice for establishing temporal order is a well-behaved raw counter such as HPET(t), which increases monotonically. Equivalently, one can scale such a counter to form a simple causal clock calibrated in seconds: $C_c(t) = p_0 \cdot \text{HPET}(t)$ where p_0 is a constant that is never updated. Such a clock tells the wrong time and drifts, but does an excellent job of use 1 and is completely independent of network conditions, in fact not requiring server timestamps at all!

The errors of an absolute clock $C_a(t)$ also make it unsuitable for use 2. This is best seen through an example: if a counter has a rate error of 0.1 ppm (this is quite typical in a PC under reasonable temperature environments), then over a one-second interval the drift

Figure 4. In-kernel measurement of the one-second gap between pulses from a GPS receiver using a RADclock difference and RADclock absolute clock. Here the polling period to the server is 256 seconds. A single point per period is shown corresponding to the worst absolute error over the period.



amounts to an error of only $10^{-7} \cdot 1 = 0.1$ μs . On the other hand, errors in $C_a(t)$, and hence in the duration measurement, could be anything from several μs to several ms to larger than a second in extreme cases, depending on factors such as network congestion and algorithm robustness.

A better choice for the purpose of measuring time differences is to use a *difference* clock—that is, one that is not corrected for drift. A simple example is:

$$C_d(t) = C_0 + p_{av} \cdot \text{HPET}(t)$$

where C_0 is a constant (never updated), and p_{av} is an estimate of long-term average rate, which we can think of as a constant (we are simplifying here slightly for clarity; see Veitch et al.⁴ for details). Such a clock tells only approximately the right time and drifts, but does an excellent job of use 2 provided that p_{av} can be robustly and accurately estimated (which it can). It is reliant on server timestamps, but in a much less sensitive and more robust way than $C_a(t)$, as it takes direct advantage of the high stability (rate constant up to 0.1 ppm) of the local hardware.

Figure 4 compares the error in the time between pulses entering the serial port of a PC from a GPS receiver (nominally precisely one second apart) as measured by both an accurate absolute clock and an accurate difference clock. They are orders of magnitude different, even though the server was close by (minimum RTT of around 1ms). In fact, the operating system/serial port adds a noise of around 2 μs to these measurements and so actually swamps the error in $C_d(t)$: the differ-

ence clock is so good at the one-second scale that it is a serious challenge to measure its error!

A final crucial point: a difference clock works by strategically ignoring drift, but over longer durations the resulting error grows, and so the difference clock loses accuracy. A good rule of thumb is that the crossover occurs at $\tau = 1,000$ seconds. Above this scale durations should instead be calculated using an absolute clock. This crossover value can be calculated by finding the minimum in an *Allan Deviation* plot, which measures oscillator stability (variability as a function of timescale).

Time to Paradigm: Feedback or Feed-Forward?

Consider the following approach to synchronizing an absolute clock $C_a(t)$. Timing packets are timestamped in the host using C_a . These timestamps are compared with those taken by the server, and an assessment is made of the clock error. The rate of C_a is then slightly adjusted to bring that error to zero over time. This is an example of a *feedback* approach, because the previous round of clock corrections feeds directly back as inputs to the algorithm, because it is the clock C_a itself that is used to generate the host timestamps.

An alternative is to use the underlying counter to make *raw* packet timestamps (that is, counter readings) in the host. The clock error is then estimated based on these and the server timestamps, and “subtracted out” when the clock is read. This is a *feed-forward* approach, since errors are corrected based on post-processing outputs, and these are not themselves

fed back into the next round of inputs. In other words, the raw timestamps are independent of clock state. One could say that the hardware reality is kept in direct view rather than seeing it through algorithmic glasses.

The NTP synchronization algorithm is a feedback design. It uses a combination of PLL (phase-locked loop) and FLL (frequency locked loop) approaches to lock onto the rhythm of the server clock.² An example of a feed-forward design is that of the RADclock, a bidirectional, minimum RTT-filtered, feed-forward-based absolute and difference synchronization algorithm, that lies at the heart of our RADclock project at the University of Melbourne.⁴

The feedback approach has two significant disadvantages in the context of the Internet, where the delays are large and highly variable, and the feedback rate is low (the period between timing packets is typically between 64 and 1,024 seconds, since, for scalability, we cannot saturate the network with timing packets). The first disadvantage is that stability of classical control approaches such as PLL and FLL cannot be guaranteed. In other words, they can lose their lock if conditions aren’t nice enough, resulting in shifts to high-error modes that may last for long periods or even be permanent. Figure 5 gives an example of this, comparing the errors in `ntpd` and the absolute RADclock (sharing exactly the same timing packets to a server on an uncongested LAN) over two weeks. The feedback stability of `ntpd` is lost on a number of occasions, resulting in larger errors.

The second disadvantage is that difference clocks *cannot even be de-*

fixed in a feedback framework, so we lose their benefits, which include not only much higher accuracy, but also far higher robustness. It is worth noting that in networking, time differences are arguably more commonly used than absolute times. Delay jitter, RTTs, inter-arrival times, and code execution times are all examples of time differences that are best measured with a difference clock.

One disadvantage of a feed-forward approach is that it does not in itself guarantee that the clock never moves backward; however, a causality enforcing clock-read function can fix this without compromising the core design.

A Question of Support

The NTP system is the incumbent, supported by all major operating systems. Let us look at some kernel support issues, focusing on FreeBSD and Linux.

The system clock maintained by the kernel (which supplies the well-known `gettimeofday()` function call), and the kernel support for algorithms to discipline it, have historically been, and remain, closely tied to the needs of the `ntpd` daemon. In particular, the counter abstractions (`timecounter` in FreeBSD and `clocksource` in Linux), which give processes access to different hardware counters available in the system, fail to provide *direct* access to the raw counter values. Instead, these are accessible only via timestamps taken by the system clock that uses the counters under the hood. In other words, the kernel APIs support the feedback paradigm only; feed-forward algorithms are, quite simply, barred from participation.

Another important consequence

of the `ntpd`-oriented nature of the existing system clock synchronization architecture is that the feedback loop encourages a separation of the algorithm “intelligence” between user space and the kernel. The `ntpd` daemon operates in the former, but the system clock has its own adaptive procedures that are, in effect, a secondary synchronization system. Two feedback systems linked via a feedback loop are difficult to keep stable, maintain, and even understand. In contrast, by making raw counter values accessible from user space, feed-forward algorithms can avoid the secondary system altogether and concentrate the algorithm intelligence and design in a single, well-defined place. The division of labor is then clean: the synchronization algorithm is in charge of synchronization; the kernel handles the timestamping.

Note that, exactly as in the causal clock $C_c(t)$, a counter can be scaled by a constant so that it reads in more convenient and universal units, and this will not affect the feed-forward algorithm’s ability to synchronize based on it. Linux’s `CLOCK_MONOTONIC_RAW`, for example, provides such a scaled counter, which ticks in (approximate and drifting) nanoseconds.

Currently, the RADclock gets around the lack of feed-forward support by providing patches that extend these mechanisms in FreeBSD and Linux in minimal ways to allow raw counter access to both the kernel and user space. The RADclock API includes difference and absolute clock reading functions based on direct counter timestamping, combined with the latest clock parameters and drift estimate.

The Forgotten Challenge

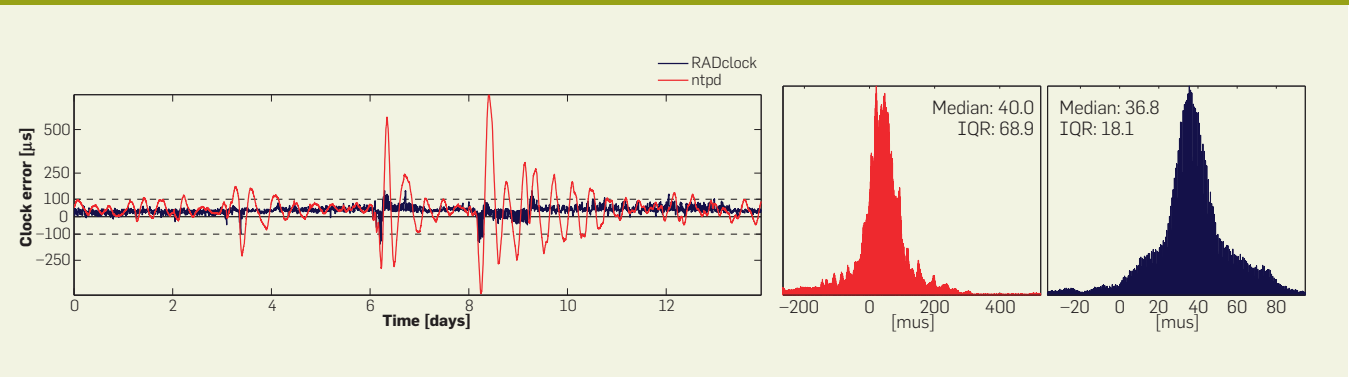
Before continuing, we should point out an annoying truth: *synchronization over networks is actually impossible*. To understand this, let’s reduce the problem to its essence by assuming zero network congestion and system load, so that all delays take their constant, minimal values.

Let $A = d^\uparrow - d^\downarrow$ denote the true path asymmetry, where d^\uparrow and d^\downarrow are the true minimum one-way delays to and from the server, respectively; and let $r = d^\uparrow + d^\downarrow$ be the minimal RTT (see Figure 2). The problem is the existence of a fundamental ambiguity because path asymmetry cannot be measured independently of clock error. The essence of this is the following: if I receive a timestamp T from the server at $t = 1.1$ reading $T = 1$, I cannot tell if I am perfectly synchronized to a server $d^\downarrow = 0.1$ seconds away or, alternatively, if I am 1 second behind true time (packet actually arrived at $t = 2.1$) and the server is 1.1 seconds away.

The asymmetry ambiguity cannot be circumvented, even in principle, by any algorithm. There is, however, some good news. First, this is a problem that plagues absolute clocks only; difference clocks are unaffected. Second, bidirectional exchanges allow constraints flowing from causality (packets can’t arrive before they are sent) to act. Hence, the asymmetry and the associated clock error are boxed in, even if they can’t be known precisely.

The question remains, how are absolute clocks achieving the impossible? What are they returning? In practice, in the absence of any external side-information on A , we must guess a value, and $\hat{A} = 0$ is typically chosen corresponding

Figure 5. Performance of RADclock and ntpd over 14 days synchronizing to a Stratum-1 server on the LAN with a 1024 s polling period; time series (left) and histograms (right). Stability issues are encountered for the feedback control of `ntpd`, resulting in an IQR (Inter-Quartile Range) which is four times wider.



to a symmetric path. This allows the clock to be synchronized, but only up to an unknown error E lying somewhere in the range $[-r, r]$. This range can be tens of milliseconds wide in some cases and can dwarf other errors.


There is another key point to remember here. It is that any change in either the true asymmetry (say because of a routing change), or the estimate of it used by the algorithm, makes the clock jump. For example, based on some knowledge of the routing between the host and the server $r = 15$ ms away, an intrepid administrator may replace the default $\hat{A} = 0$ with a best guess of $\hat{A} = 3$ ms, resulting in a jump of $3/2 = 1.5$ ms. Another example is a change in choice of server, which inevitably brings with it a change in asymmetry. The point is that jumps, even if they result in improvements in synchronization, are an evil unto themselves. Such *asymmetry jitter* can confuse software not only in this host, but also in others, since all OWD's measured to and from the host will also undergo a jump.

To summarize, network synchronization consists of two very different aspects. The synchronization algorithm's role is to see through and eliminate delay variability. It is considered to be accurate if it does this successfully even if the asymmetry error and, therefore the final clock error, is large, as it cannot do anything about this. The asymmetry jitter problem is not about variability but an unknown constant. This is so much simpler; however, it is inherently hard as it cannot be circumvented, even in principle. So here is the challenge: although it cannot be eliminated, *the practical impact of asymmetry depends strongly on how it is managed*. These two very different problems cross paths in a key respect: both benefit from nearby servers.


Robust Algorithm Design

Here is a list of the key elements for reliable synchronization.

Don't forget physics. The foundation of the clock is the local hardware. Any self-respecting algorithm should begin by incorporating the essence of its behavior using a physically meaningful model. Of particular importance are the following simple characterizations of its stability: the large-scale rate error bound (0.1 ppm) and the timescale



With feedforward, the hardware reality is kept in direct view rather than seeing it through algorithmic glasses.



where the oscillator variability is minimal ($\tau = 1,000$ seconds). These characteristics are remarkably stable across PC architectures, but the algorithm should nonetheless be insensitive to their precise values.

The other fundamental physics component is the nature of the delays. A good general model for queuing delays, and hence OWD and RTT, is that of a constant plus a positive random noise. This constant (the minimum value) is clearly seen in Figure 3 in the case of RTT.

Use feed-forward. A feed-forward approach offers far higher robustness, which is essential in a noisy unpredictable environment such as the Internet. It also allows a difference clock to be defined, which in turn is the key to robust filtering for the absolute clock, as well as being directly valuable for the majority of timing applications including network measurement (OWDs aside).

The difference clock comes first. Synchronizing the difference clock equates to measuring the long-term average period p_{av} . This “rate synchronization” is more fundamental, more important, and far easier than absolute synchronization. A robust solution for this is a firm foundation on which to build the much trickier absolute synchronization.

Note that by *rate synchronization* we mean a low noise estimate of *average long-term rate/period*, not to be confused with short-term rate, which reduces essentially to (the derivative of) drift, and hence to absolute synchronization.

Use minimum RTT-based filtering. If a timing packet is lucky enough to experience the minimum delay, then its timestamps have not been corrupted and can be used to set the absolute clock directly to the right value (asymmetry aside). The problem is, how can we determine which packets get lucky?

Unfortunately, this is a chicken-and-egg problem, since to measure OWD, we have to use the absolute clock $C_a(t)$, which is the one we want to synchronize in the first place. Luckily, the situation is different with RTTs, which can be measured by the difference clock $C_d(t)$. The key consequence is that a reliable measure of packet quality can be obtained without the need to first absolutely synchronize the clock. One just measures by how much the RTT

of the given timing packet exceeds the minimum RTT: the smaller the gap, the higher the ‘quality’ of the packet.

The sharp-eyed reader would have noticed that a chicken-and-egg problem remains. The difference clock also needs filtering in order to synchronize it, so how can we use it before it is synchronized? The answer is that even a very poor estimate of p_{av} is sufficient to perform meaningful filtering. For example, imagine that p_{av} is known to an accuracy of 100 ppm (which is truly atrocious) and that $RTT=10ms$. Then the error in the RTT measurement (including that resulting from drift) is of the order of $1\ \mu s$, which is well below the noise caused by the operating system (around $10\ \mu s$).

Rate synchronization. Let’s assume that our filtering is successful, so we can reliably measure the level of quality of timing packets.

The key to rate synchronization (that is, the measurement of p_{av}) is that it is a long-term average, which means that we can afford to be patient! Reliable rate synchronization is then as simple as collecting reasonable-quality timestamps at either end of a large interval (ideally several days, but even much less will work well). Timestamp errors and errors in the quality assessment are then compressed by the size of the interval and enter into the 0.001-ppm zone where they will never bother you again.

Once synchronized, the p_{av} value has a long life. One consequence is that even if connectivity to a server were lost, the difference clock would remain well synchronized. Using a

hardware-validated test bed, we performed tests showing that the RAD-clock difference clock measured one-second intervals to sub- μs accuracy even after 20 days of disconnection from the server. Compare this level of robustness with that of absolute synchronization, where over such an interval the local clock will inevitably drift considerably, or worse.

Absolute synchronization. Drift is constantly happening, and a clock must be ready to be read at any time. It follows that patience is not an option here: even if congestion is high and timing packets delayed, the best must still be made of a bad situation; the drift must be dealt with.

There are two chief considerations. The first is to use the timing packet quality to control its contribution to the clock error estimation. The control must be very strict: if the packet has an RTT that is only a little above the minimum, that gap corresponds directly to an error to be avoided.

The second consideration is time scale. The essential trade-off is the need to have data from as many timing packets as possible to increase the chances of getting lucky, versus the need for them to be recent so that the current error, rather than an outdated error from the past, is estimated. Here the time scale τ plays a key role, as it places a meaningful limit on how far back in the past to look.

The absolute clock can be defined off the difference clock simply as follows:

$$C_a(t) = C_d(t) - \hat{E}(t)$$

where $\hat{E}(t)$ is the estimate of the error of $C_d(t)$, which is removed on the fly when the absolute clock is read by a process. Figure 6 shows $\hat{E}(t)$ as the blue curve, which is calculated based on the noisy unfiltered per-packet error estimates that underlie it (green spikes). The true error $E(t)$ is the black curve, as measured using an external GPS-synchronized DAG capture card¹ in our test bed.

One server is enough. So far we have discussed synchronizing to a single server. Surely one could connect to multiple servers and select among them to obtain even better results? In principle this is true; in Internet practice, at least with the current state of the art, it is most definitely not, for two reasons.

Most fundamentally, switching servers implies a change in path asymmetry and hence a jump in the clock. Imagine a server-selection algorithm that is moving around among candidate servers of similar quality. The result is constant jumping—a classic case of asymmetry jitter. Such jitter is not hard to observe in `ntpd`, where a connection to multiple peers is in fact recommended.

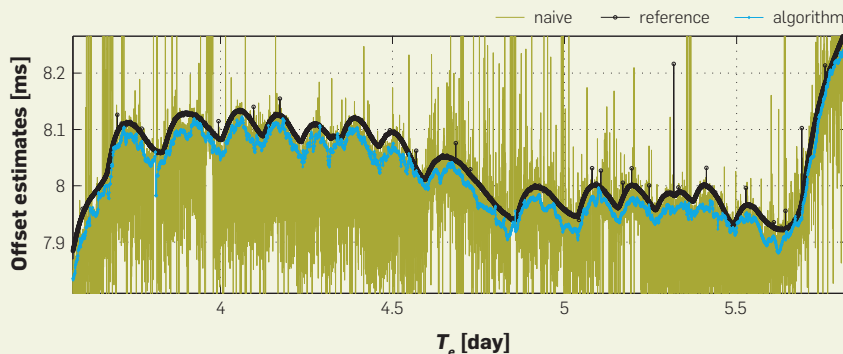
Second, once performance is tuned to close to system and network noise limits, any disruption at all will downgrade it. Ceasing to query a server for a time and then returning to it, for example, qualifies as a moderate to large disruption.

The take-home message for administrators is this: if you want to improve your system clock performance, just make sure the configuration points only to a single (nearby) server.

Watch your route. Servers do change, and even if they don’t, the routing to them will eventually. Not only will this result in an unavoidable jump for asymmetry reasons, it is also a challenge to the synchronization algorithm itself. Careful design is needed so that the rate and absolute synchronization algorithms transition gracefully and robustly to the new environment. This has to begin with a well-thought-out approach to the quality assessment underlying the filtering. If this breaks down, then quality packets can be assessed as good and good as bad, and there is no limit to the potential damage.

Trust yourself. Yes, the server is the expert, without which synchronization would be impossible. Still, it should

Figure 6. An unsynchronized clock drifting. The black line is the drift, the green the drift seen through the noise of delay variability, and the blue the estimate of the drift by the RADclock algorithm (it is offset vertically from the black by about $30\ \mu s$ because path asymmetry has not been corrected for here). Source: Veitch et al.⁴



not be trusted. Servers can and do have their bad periods, and blind faith in them can lead to deep trouble. Fortunately another authority is available: the counter model. If the RTT filtering is telling you that congestion is low, yet the server timestamps are saying you are suddenly way out, trust the hardware and use the model to sanity-check the server. Basically, a drift well over 1 ppm is not credible, and the algorithm should smell a rat.

When in doubt, just drift. What if congestion has become so high that none of the available timestamps is of acceptable quality? What if I don't trust the server, or if I have lost contact with it entirely?


There is only one thing to do: sit back and relax. Nothing bad will happen unless the algorithm chooses to make it happen. A reaction of inaction is trivial to implement within the feed-forward paradigm and results in simply allowing the counter to drift gracefully. Remember the counter is highly stable, accumulating only around 1 μ s per second, at worst.

More generally, the algorithm should be designed never to overreact to anything. Remember, its view of the world is always approximate and may be wrong, so why try to be too clever when inaction works so well? Unfortunately, feedback algorithms such as `ntpd` have more reactive strategies that drive the clock more strongly in the direction of their opinions. This is a major source of their nonrobustness to disruptive events.


The Bigger Picture

Thus far we have considered the synchronization of a single host over the network to a server, but what about the system as a whole? In NTP, the main system aspect is the server hierarchy. In a nutshell, Stratum-1 servers anchor the tree as they use additional hardware (for example, a PC with a GPS receiver or a purpose-built synchronization box with GPS) to synchronize locally, rather than over a network. By definition, Stratum-2 servers synchronize to Stratum-1, Stratum-3 to Stratum-2, and so on, and hosts synchronize to whatever they can find (typically a Stratum-2 or a public Stratum-1).

At the system level there are a number of important and outstanding chal-




Yes, the server is the expert, without which synchronization would be impossible. Still, it should not be trusted. Servers can and do have their bad periods, and blind faith in them can lead to deep trouble.



lenges. Stratum-1 servers do not communicate among themselves, but act (except for load balancing in limited cases) as independent islands. There is a limited capability to query a server individually to obtain basic information such as whether it is connected to its hardware and believes it is synchronized, and there is no ability to query the set of servers as a whole. An interconnected and asymmetry-aware Stratum-1 infrastructure could provide a number of valuable services to clients. These include recommendations about the most appropriate server for a client, automatic provision of backup servers taking asymmetry jitter into account, and validated information on server quality. Currently no one is in a position to point the finger at flaky servers, but they are out there.

Building on the RADclock algorithm, the RADclock project³ aims to address these issues and others as part of a push to provide a robust new system for network timing within two years. Details for downloading the existing client and server software (packages for FreeBSD and Linux), documentation, and publications can be found on the RADclock project page.

Acknowledgments

The RADclock project is partially supported under Australian Research Council's Discovery Projects funding scheme (project number DP0985673), the Cisco University Research Program Fund at Silicon Valley Community Foundation, and a Google Research Award. 

References

1. Endace Measurement Systems. DAG series PCI and PCI-X cards; <http://www.endace.com/networkMCards.htm>.
2. Mills, D.L. 2006. *Computer Network Time Synchronization: The Network Time Protocol*. CRC Press, Boca Raton, FL, 2006.
3. Ridoux, J., Veitch, D. RADclock Project Web page; <http://www.cubinlab.ee.unimelb.edu.au/radclock/>.
4. Veitch, D., Ridoux, J., Korada, S.B. Robust synchronization of absolute and difference clocks over networks. *IEEE/ACM Transactions on Networking* 17, 2 (2009), 417–430. doi: 10.1109/TNET.2008.926505.

Julien Ridoux is a Research Fellow at the Center for Ultra-Broadband Information Networks (CUBIN) in the Department of Electrical & Electronic Engineering at The University of Melbourne, Australia.

Darryl Veitch is a Principal Research Fellow at the Center for Ultra-Broadband Information Networks (CUBIN) in the Department of Electrical & Electronic Engineering at The University of Melbourne, Australia.

Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.