The Nanokernel[1]

David L. Mills[2]
University of Delaware
and
Poul-Henning Kamp
FreeBSD Project

Abstract

Internet timekeeping has come a long way since first demonstrated almost two decades ago. In that era most computer clocks were driven by the power grid and wandered several seconds per day relative to UTC. As computers and the Internet became ever faster, hardware and software synchronization technology became much more sophisticated. The Network Time Protocol (NTP) evolved over four versions with ever better accuracy now limited only by the underlying computer hardware clock and adjustment mechanism.

The clock frequency in modern workstations is stabilized by an uncompensated quartz or surface acoustic wave (SAW) resonator, which are sensitive to temperature, power supply and component variations. Using NTP and traditional Unix kernels, incidental timing errors with an uncompensated clock oscillator is in the order of a few hundred microseconds relative to a precision source. Using new kernel software described in this paper, much better performance can be achieved. Experiments described in this paper demonstrate that errors with a modern workstation and uncompensated clock oscillator are in the order of a microsecond relative to a GPS receiver or other precision timing source.

1. Introduction

Several years ago the software algorithms to discipline the Unix system clock were overhauled to provide improved accuracy, stability and resolution [5]. In addition, means were added to discipline the clock directly from a precision timing source, such as a GPS receiver or cesium oscillator. The software was integrated with several operating system kernels of the day and eventually adopted as standard in Digital Tru64 (Alpha), Sun Solaris, Linux and FreeBSD. The best performance achieved with workstations of the day was a few hundred microseconds in time and a few parts-per-million (PPM) in frequency, so a clock resolution of one microsecond seemed completely adequate.

With workstations and networks of today reaching speeds in the gigahertz range, it is clear the solution of several years ago is rapidly becoming obsolete. Improved modelling techniques have resulted in better discipline algorithms which are more responsive to phase and frequency characteristics of computer clocks [3]. Faster processors and a standardized application program interface (API) allow more flexible and precise timing of external signals [7]. Faster network speeds and lower jitter provide more accurate timekeeping over the Internet [4].

---

1. Sponsored by: DARPA Information Technology Office Contract F30602-98-1-0225 and Digital Equipment Corporation Research Agreement 1417.

2. David L. Mills is with the Electrical and Computer Engineering Department, University of Delaware, Newark, DE 19716, mills@udel.edu, http://www.eecis.udel.edu/~mills; Poul-Henning Kamp is with the FreeBSD Project, Valbygrdsvej 8, DK-4200 Slagelse, Denmark. phk@freebsd.org.

This paper describes new algorithms and kernel software providing much improved time and frequency resolution, together with a more agile and precise clock discipline mechanism. It discusses the analysis and design of the algorithms and the results of proof-of-performance experiments. The software has been implemented and tested in all the kernels mentioned above and is now standard in the Linux and FreeBSD public distributions.

The kernel software replaces the clock discipline algorithm in a synchronization daemon, such as the Network Time Protocol [6], with equivalent functionality in the kernel. It provides a resolution of 1 ns in time and .001 PPM in frequency. While clock corrections are recomputed about once per minute in the daemon, they are recomputed once per second and amortized at every tick interrupt in the kernel. This avoids errors that accumulate between updates due to the intrinsic hardware clock frequency error.

The new software can be compiled for 64-bit machines using native instructions or for 32-bit machines using a macro package for double precision arithmetic. The software can be compiled for kernels where the time variable is represented in seconds and nanoseconds and for kernels in which this variable is represented in seconds and microseconds. In either case the resolution of the clock is limited only by the resolution of the clock hardware. Even if the resolution is only to the microsecond, the software provides extensive signal grooming and averaging to minimize reading errors.

The remaining sections of this paper are organized as follows. Section 2 describes the characteristics of typical computer clock oscillators, which are based on the Allan deviation statistic used in the most recent NTP algorithms. Section 3 describes the software design, which is based on two interacting hybrid phase-lock/frequency-lock (PLL/FLL) feedback loops. Section 4 describes the software implementation, which is integrated in the kernels mentioned above. Section 5 summarizes the results of proof-of-performance experiments which validate the claims in this paper. Section 6 concludes with suggestions for further improvements.

2. Computer Clock Characterization

In order to understand how the new kernel algorithms operate, it is necessary to understand the design of a typical computer clock and how the time and frequency is controlled. The accuracy attainable with NTP, or any other protocol that provides periodic offset measurements, depends strongly on the stability of the clock oscillator and the precision of its adjustment mechanism. The clock frequency in modern workstations is stabilized by an uncompensated quartz or surface acoustic wave (SAW) resonator, which are affected by temperature, power supply and component variations. The most significant affect is the temperature dependency, which is typically in the order of one PPM in frequency per degree Celsius.

In typical computer clock designs the clock oscillator drives a counter that produces processor interrupts at fixed tick intervals in the range 1-20 ms. At each tick interrupt a software clock variable is updated by the number of microseconds or nanoseconds in the tick interval. The means used by the traditional Unix kernel to adjust the clock time is the `adjtime()` kernel routine, which causes a fixed value, typically 5 µs, to be added to or subtracted from the clock time at each tick interrupt. The `adjtime()` function computes how long these increments must be continued in order to amortize the adjustment specified. In order to provide a frequency offset, the NTP daemon calls the `adjtime()` routine at intervals of one second. Since the intrinsic clock oscillator frequency error can range to several hundred PPM, this can result in sawtooth-like time errors ranging to several hundred microseconds. This was the prime motivation to avoid the `adjtime()` routine and implement the clock discipline directly in the kernel.

Almost all modern processors provide means to measure intervals for benchmarking and profiling. These means typically take the form of a processor cycle counter (PCC), which can be read by a machine instruction. Upon receiving a request to read the clock, the kernel uses the PCC to compute the number of microseconds or nanoseconds since the last tick interrupt. Since the PCC and clock oscillator may not run at the

same frequency and, in the case of multiprocessor systems, there may be more than one PCC, the kernel must carefully mitigate the differences and develop a stable, monotonically increasing timescale.

It is well known that the behavior of an oscillator can be characterized in terms of its Allan deviation, which is a function of stability, interpreted as first-order frequency differences, and averaging interval [1]. In order to determine this statistic for a typical uncompensated computer oscillator, sample offsets relative to a cesium standard were measured with the computer oscillator allowed to free-run over periods ranging from 1.5 to 10 days. These data were saved in files and later used to construct plots in log-log coordinates showing stability versus averaging interval.

In [3] a simple model is developed which characterizes the performance of each individual time server. The model characterizes each combination of synchronization source and clock oscillator by two intersecting straight lines in log-log coordinates. In general, network and computer latency variations produce jitter, which is modelled as white phase noise and appears as a straight line with slope −1 on the plot. On the other hand, oscillator frequency variations produce wander, which is modelled as random-walk frequency noise and appears as a straight line with slope +0.5. The intersection of the two straight lines is called the Allan intercept, which serves to characterize the particular combination of source and oscillator. It represents the optimum averaging interval for the best oscillator stability. If the averaging interval is less than this, errors due to source jitter dominate, while if greater, errors due to oscillator wander dominate.

The averaging interval is roughly equal to the frequency time constant used in the clock discipline algorithm, and this is related to the interval between NTP poll messages sent across the network. With a minimum poll interval of 16 s in the current NTP design, the averaging interval is about 4,000 s, which is on the high side of the optimum range, and the match gets worse with larger poll intervals. Thus, the best accuracy is achieved at the minimum poll interval, but this may result in unacceptable network overhead. Therefore, when the NTP daemon is started, it uses a relatively small poll interval in order to respond quickly to the particular oscillator frequency offset, then gradually increases the interval to an upper limit. Depending on desired accuracy and allowable network overhead, the upper limit could be a small as a few seconds or as large as a day or more.

A phase-lock loop (PLL) functions best with poll intervals below the Allan intercept where jitter predominates, while a frequency-lock loop (FLL) functions best above the intercept where wander predominates. As the result of previous research [2][3], a hybrid PLL/FLL clock discipline algorithm has been designed, implemented and tested in the NTP version 4 software for Unix, Windows and VMS. A kernel implementation based on this design is described in the following section.

3. Software Design

The nanokernel software design is based on the NTP implementation, but includes two separate but interlocking feedback loops. The PLL/FLL discipline operates with periodic updates produced by a synchronization daemon such as NTP, while the PPS discipline operates with an external PPS signal and modified serial or parallel port driver. Both algorithms include grooming provisions that significantly reduce the impact of source selection jitter or *clockhopping* and network delay transients. In addition, the PPS algorithm can continue to discipline the clock frequency even if other synchronization sources or the daemon itself crash.

3.1 PLL/FLL Discipline

The PLL/FLL kernel discipline is specially tailored for typical Internet delay jitter and clock oscillator wander. However, the kernel embodiment provides better accuracy and stability than the NTP discipline, as well as a wider operating range. Both the kernel discipline and NTP discipline operate in the same manner except for one important detail. The NTP discipline uses the kernel `adjtime()` system call, which has an inherent resolution of 1 μs in time and 5 PPM in frequency and amortizes adjustments once every

second. The kernel discipline has an inherent resolution of 1 ns in time and .001 PPM in frequency and amortizes adjustments at every tick interrupt.

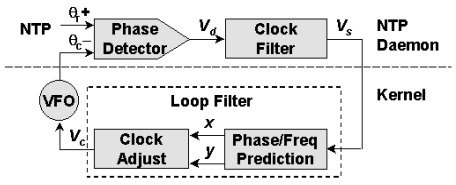

Figure 1. Clock Discipline Feedback Loop

Both the kernel discipline and NTP discipline operate as a hybrid of phase-lock and frequency-lock feedback loops. Figure 1 shows the functional components of the kernel discipline. In the NTP discipline the components below the dotted line are implemented in the daemon. The phase difference V_d between the reference source θ_r and clock θ_c is determined by the NTP daemon. The value is then groomed by the NTP clock filter and related algorithms to produce the phase update V_s used by the loop filter in the kernel to produce the phase prediction *x* and frequency prediction *y*. These predictions are used to produce clock adjustment updates at intervals of 1 s which result in the correction term V_c. This value represents the increment in time necessary to correct the clock at the end of the next second. The various performance data displayed later were derived from the phase update V_s, since this is a common measuring point for both the daemon and kernel.

The *x* and *y* predictions are developed from the phase update V_s as shown in Figure 2. As in the NTP algorithm, the phase and frequency are disciplined separately in both PLL and FLL modes. In both modes *x* is the value V_s, but the actual phase adjustment is calculated by the clock adjust process using an exponential average with an adjustable weight factor. The weight factor is calculated as the reciprocal of the time constant specified by the API. The value can range from 1 s to an upper limit determined by the Allan intercept. In PLL mode it is important for the best stability that the update interval does not significantly exceed the time constant for an extended period.

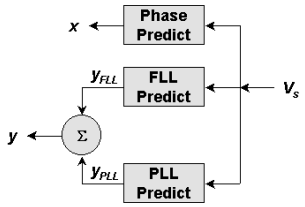

Figure 2. FLL/PLL Prediction Functions

In PLL mode, *y* is computed using an integration process as required by PLL engineering principles; however, the integration gain is reduced by the square of the time constant, so adjustments become essentially ineffective with poll intervals above 1024 s. In FLL mode, *y* is computed directly using an exponential average with weight 0.25. This value, which was determined from simulation with real and synthetic data, is a compromise between rapid frequency adaptation and adequate glitch suppression. In operation, PLL mode is preferred at small update intervals and time constants and FLL mode at large intervals and time constants. The optimum crossover point between the PLL and FLL modes, as determined by simulation and analysis, is the Allan intercept. As a compromise, the PLL/FLL algorithm operates in PLL mode for update intervals of 256 s and smaller and in FLL mode for intervals of 1024 s and larger. Between 256 s and 1024 s the mode is specified by the API.

3.2 PPS Discipline

In order to reduce incidental errors to the lowest practical value, it is necessary to use a precision source, such as a GPS receiver or precision oscillator. The kernels mentioned above have been modified for this purpose. For serial drivers the PPS signal is connected to the DCD pin via a level converter; for parallel drivers the signal is connected directly to the ACK pin. A comprehensive API has been designed and implemented for this function. It is currently the subject of a Internet Engineering Task Force proposed standard [7].

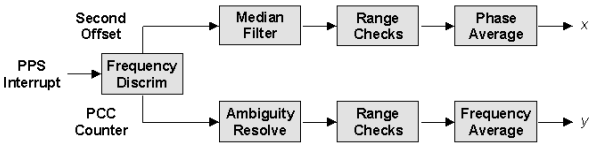

Figure 3. PPS Discipline

The PPS algorithm shown in Figure 3 is functionally separate from the PLL/FLL discipline; however, the two disciplines have interlocking control functions designed to provide seamless switching between them as necessary. The PPS discipline is called at each PPS on-time signal transition with arguments including a clock timestamp and a virtual nanosecond counter sample. The virtual counter can be implemented using the PCC in modern computer architectures or a dedicated counter in older architectures. The intent of the design is to discipline the clock phase using the timestamp and the clock frequency using the virtual counter. This makes it possible, for example, to stabilize the clock frequency using a precision PPS source, while using an external time source, such as a radio or satellite clock or even another time server, to discipline the phase. With frequency reliably disciplined, the interval between updates from the external source can be greatly increased. Also, should the external source fail, the clock will continue to provide accurate time limited only by the accuracy of the precision source.

At each PPS on-time transitional the offset in the second is determined relative to the clock phase. A range gate rejects errors more than 500 μs from the nominal interval of 1 s, while a frequency discriminator rejects errors more than 500 PPM from the nominal frequency of 1 Hz; however, the design tolerates occasional dropouts and rejects noise spikes. The virtual counter samples are processed by an ambiguity resolver that corrects for counter rollover and certain anomalies when a tick interrupt occurs in the vicinity of the second rollover or when the PPS interrupt occurs while processing a tick interrupt. The latter appears to be a feature of at least some Unix kernels which rank the serial port interrupt priority above the tick interrupt priority.

PPS samples are then processed by a 3-stage shift register. The median value of these samples is the raw phase signal and the maximum difference between them is the raw jitter signal. The PPS phase correction is computed as the exponential average of the raw phase with weight equal to the reciprocal of the calibration interval described below. In addition, a jitter statistic is computed as the exponential average of the raw jitter with weight 0.25 and reported as the jitter value in the API. Occasional electrical transients due to light switches, air conditioners and water pumps are a principal hazard to PPS discipline performance. A spike (*popcorn*) suppressor rejects phase outlyers with amplitude greater than 4 times the jitter statistic. This value, as well as the jitter averaging weight, was determined by simulation with real and synthetic PPS signals.

The PPS frequency is computed directly from the difference between the virtual counter values at the beginning and end of the calibration interval, which varies from 4 s to a maximum specified by the API. When the system is first started, the clock oscillator frequency error can be quite large, in some cases 200 PPM or more. In order to avoid ambiguities, the counter differences must not exceed the tick interval, which can be less than a millisecond in some kernels. The choice of minimum calibration interval of 4 s insures that the differences remain valid for frequency errors up to 250 PPM with a 1-ms tick interval.

The actual PPS frequency is calculated by dividing the virtual counter difference by the calibration interval in seconds. In order to avoid divide instructions and intricate residuals management, the calibration interval is always a power of 2, so division reduces to a shift. However, due to signal dropouts or noise spikes, either the length may not be a power of 2 or the signal may appear outside the valid frequency range, so the interval is ignored. The required frequency adjustment is computed and clamped not to exceed 100 PPM. This acts as a damper in case of abrupt changes that can occur at reboot, for example.

4. Software Implementation and Operation

Figure 4 shows the general organization of the kernel software. Updates produced by the NTP daemon are processed by the `hardupdate()` routine, while PPS signal interrupts are processed by the `hardpps()` routine. The values in both routines are calculated using extended precision arithmetic to preserve nanosecond resolution and avoid overflows over the range of clock oscillator frequencies from 50 Hz to above 1000 Hz. The actual corrections are redetermined once per second and linearly amortized over the second at each hardware tick interrupt. In contrast to the NTP daemon, where most computations use floating-double data types, the kernel is limited to integer data types.

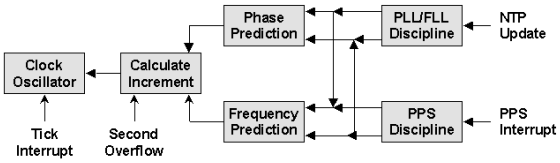

Figure 4. Kernel Clock Discipline

Both the `hardupdate()` and `hardpps()` routines discipline the computer clock in nanoseconds in time and nanoseconds per second in frequency. There are two programs which implement the kernel algorithms, `ktime.c` and `micro.c`. The `ktime.c` program includes code fragments that implement the `hardupdate()` and `hardpps()` routines, as well as the `ntp_gettime()` and `ntp_adjtime()` system calls that implement the API. The `micro.c` program implements a nanosecond clock using the tick interrupt augmented by the virtual counter described above. In its present form, it can be compiled only for 64-bit architectures. In this program the `nano_time()` routine measures the intrinsic processor clock frequency, then interpolates the nanoseconds be scaling the PCC to one second in nanoseconds. The unavoidable divide instruction is the only one in the nanokernel software. The routine also supports a microsecond clock for legacy purposes.

Since the PPS signal is inherently ambiguous, the seconds numbering is established by another NTP server or a local radio clock using the PLL/FLL discipline. The PPS frequency determination is independent of any other means to discipline the clock frequency and operates continuously. When the NTP daemon recognizes from the API that the PPS frequency has settled down, it switches the clock frequency discipline to the PPS signal, but continues to discipline the clock phase using the PLL/FLL algorithm. The sometimes intricate mitigation rules that control the detailed sequencing are beyond the scope of this paper; they are given in the software documentation [8].

5. Performance Evaluation

Following previous practice [3], the `ktime.c` and `micro.c` routines have been embedded in a special purpose, discrete event simulator. In this context it is possible to verify correct operation over a wide range of operating conditions likely to be found in current and future computer systems and networks and which cannot be easily duplicated with in-situ implementations. It operates with internally synthesized data or raw data files produced by the NTP daemon during regular operation. For this purpose raw time offsets are recorded with NTP operating in an open-loop configuration and later played back to the simulator. Synthetic data having similar statistics are generated as described in [3]. The simulator can measure the response to time and frequency transients, monitor for unexpected interactions between the simulated clock oscillator, PCC and PPS signals, and verify correct monotonic behavior as the various counters interact due to small frequency variations.

In order to calibrate the performance of the routines in a functioning system, they were implemented in the kernels for several architectures, including Alpha, Intel and SPARC. Detailed performance data have been collected for three systems: Rackety is a busy SPARC IPC time server running SunOS 4.1.3 and connected to four radio clocks - dual redundant GPS receivers and dual redundant WWVB receivers. The PPS signal is derived from one of the GPS receivers. Churchy is a Digital Alpha 433au personal workstation running

Tru64 4.0d and connected to a GPS receiver with PPS signal. Hepzibah is an Intel Pentium II 233 laboratory machine running FreeBSD 3.4 and connected to a GPS receiver with PPS signal.

Figure 5 shows the typical behavior of hepzibah. In this particular configuration the PPS signal was connected via a parallel port and a special kernel driver. The characteristic is decidedly spikey, in spite of the signal grooming algorithms used in the PPS discipline. The jitter budget includes contributions from the source (less than 100 ns), clock resolution (about 4 ns) and the hardware and software interrupt latencies. The interesting thing about this figure is that the jitter spikes are as often positive as negative. If due only to interrupt latencies, the spikes would be negative. There is no obvious explanation for this behavior other than to remark the standard (RMS) error is less than a microsecond.

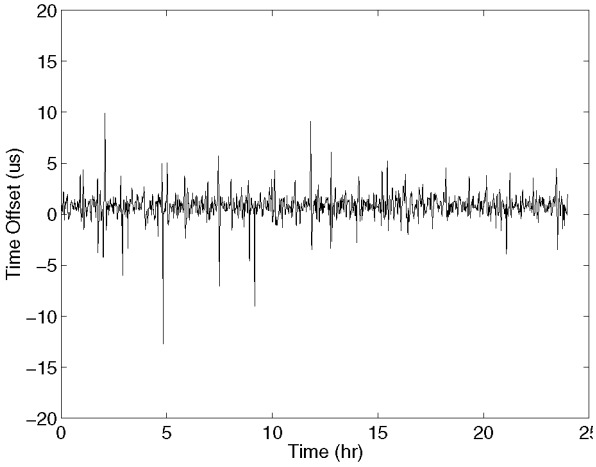

Figure 5. Time Offset for Hepzibah

While hepzibah has no applications or services other than NTP, rackety is a much slower machine dedicated to NTP service. It services an arrival stream of some 15 packets per second from an estimated client population well over 1000. The radio clocks are connected to a an 8-input multiplexor which services other ancillary devices as well. The hardware interrupt load produced by the multiplexor and network interface is severe, especially since the SPARC IPC is only a 25-MHz machine. The large negative time offset spikes shown in Figure 6 are clearly the result of interrupt latencies for the four radio clocks, the PPS signal and the network interface.

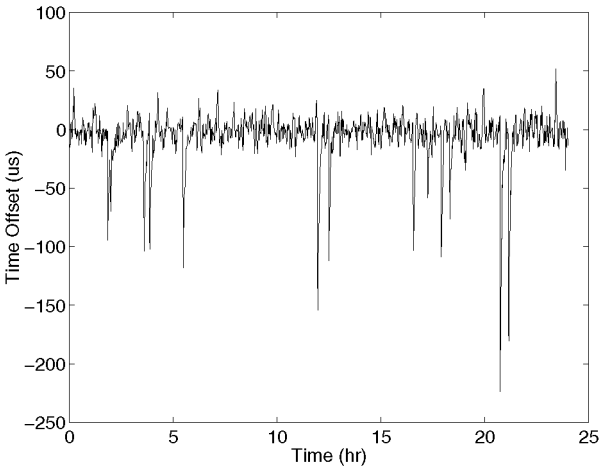

Figure 6. Time Offset for Rackety

Figure 7 shows the typical behavior for churchy, the fastest machine of the bunch. The PCC for this machine is derived from a SAW oscillator. Ordinarily, one would expect low phase noise from this type of oscillator, but the characteristics shown in the figure argue otherwise. To the trained eye, the characteristic is dominated by flicker noise. The source of this unexpected behavior is yet to be determined.

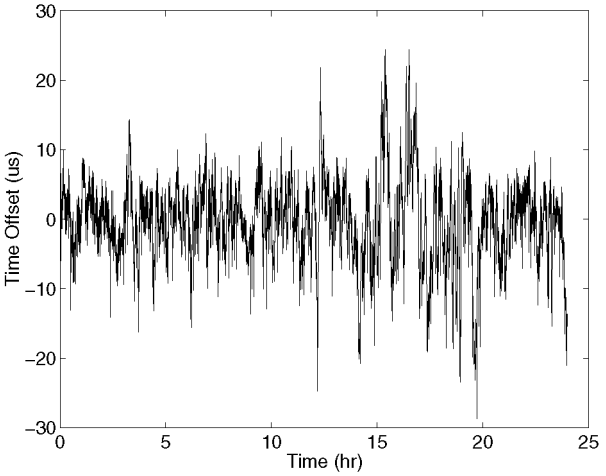

Figure 7. Time Offset for Churchy

6. Summary and Conclusions

This paper demonstrates that modern computers can maintain nominal accuracy relative to precision time sources of a microsecond or two, assuming systematic latencies due to signal conditioning, interrupt processing and timestamp capture can be calibrated out. In order to achieve this level of performance, a

hybrid PLL/FLL discipline loop is used for NTP control together with separate time and frequency loops for PPS discipline. The level of performance is probably near the best that can be achieved with an unstabilized clock oscillator. Where a fast computer with precision hardware clock is available, the performance can be improved to the order of a few tens of nanoseconds at the API. This was verified using a machine where the system clock was derived from a Rubidium oscillator and FPGA counter; however, this setup would not ordinarily be considered practical. The practical accuracy expectations of individual applications will vary depending on the mix of applications and operating system scheduling latencies.

Observations of the kernel disciplines in actual operation suggest a few areas where further improvements may be possible. One of these is the grooming algorithm used in the PPS discipline. The complexity of the median calculation increases rapidly with the number of register stages, which is only three in the current design. However, the NTP discipline operates in user space, so its resource commitments are more flexible. The NTP daemon includes a PPS driver with a 60-stage register. The algorithm sorts the offsets, then iteratively trims off the sample furthest from the median until a prespecified fraction of the original samples are left. Finally, it presents the average of these samples to the kernel PLL/FLL discipline.

The PPS driver provides significantly less jitter than the kernel PPS discipline; however, the performance advantage due to the quick response of the kernel discipline is lost. While the current minimum daemon update interval is currently limited to 16 s in the interest of minimizing kernel overhead, it might be acceptable in fast machines to reduce that interval to 1 s. Should this be done, it would be practical to do almost all discipline loop processing in user space and move the per-second processing to the daemon, where more flexible processor and memory resource commitments are possible.

7. References

Note: Papers and reports by D.L. Mills can be found in PostScript and PDF forma at www.eecis.udel.edu/~mills.

1. Allan, D.W. Time and frequency (time-domain) estimation and prediction of precision clocks and oscillators. *IEEE Trans. on Ultrasound, Ferroelectrics, and Frequency Control UFFC-34, 6* (November 1987), 647-654. Also in: Sullivan, D.B., D.W. Allan, D.A. Howe and F.L. Walls (Eds.). *Characterization of Clocks and Oscillators. NIST Technical Note 1337*, U.S. Department of Commerce, 1990, 121-128.

2. Levine, J. An algorithm to synchronize the time of a computer to universal time. *IEEE Trans. Networking 3, 1* (February 1995), 42-50.

3. Mills, D.L. Adaptive hybrid clock discipline algorithm for the Network Time Protocol. *IEEE/ACM Trans. Networking 6, 5* (October 1998), 505-514.

4. Mills, D.L. The network computer as precision timekeeper. *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting* (Reston VA, December 1996), 96-108.

5. Mills, D.L. Unix kernel modifications for precision time synchronization. Electrical Engineering Report 94-10-1, University of Delaware, October 1994, 24 pp.

6. Mills, D.L. Network Time Protocol (Version 3) specification, implementation and analysis. Network Working Group Report RFC-1305, University of Delaware, March 1992, 113 pp.

7. Mogul, J., D. Mills, J. Brittenson, J. Stone and U. Windl. Pulse-per-second API for Unix-like operating systems, version 1. Request for Comments RFC-2783, Internet Engineering Task Force, March 2000, 31 pp.

8. Network Time Protocol Version 4 software distribution, including sources and documentation. Available via the web at www.ntp.org.