1. Introduction

This is one of a series of reports on the technology of synchronizing clocks in computer networks. Previous papers and reports have described The Network Time Protocol (NTP) used to synchronize computer clocks in the Internet [MIL91a, MIL92a], modelling and analysis of computer clocks [MIL92c], the chronology and metrology of network timescales [MIL91b], and measurement programs designed to establish the accuracy, stability and reliability in service [MIL90]. This report presents a series of design improvements in interface hardware, input/output driver software and Unix operating system kernel software which improve the accuracy and stability of the local clock, especially when directly synchronized via radio or satellite to national time standards. Included are descriptions of engineered software refinements in the form of modified driver and kernel sources that reduce jitter relative to a precision timing source to the order of a few tens of microseconds and timekeeping accuracy for workstations on a common Ethernet to the order of a few hundred microseconds.

This report begins with an introduction describing the NTP architecture and protocol and the local clock, which is modelled as a disciplined oscillator and implemented as a phase-lock loop (PLL). It continues with a description of several methods designed to reduce clock reading errors due to various causes at the hardware, driver and operating system level. Some of these methods involve new or modified device drivers which reduce latencies well below the original system design. Others allow the use of special PPS and IRIG signals generated by some radio clocks, together with the audio codec included in some workstations, to avoid the latencies involved in reading ASCII timecodes. Still others involve surgery on the timekeeping software of three different Unix kernels for Sun Microsystem and Digital Equipment machines.

The report continues with descriptions of several experiments intended to calibrate the success of these improvements with respect to accuracy and stability. They establish the latencies in reading the local clock, the errors accumulated in synchronizing one computer clock to another and the errors due to the intrinsic instability of the local clock oscillator. The report concludes that it is indeed possible to achieve reliable synchronization to within a few hundred microseconds on an Ethernet or FDDI network using fast, modern workstations, and that the most important factor in limiting the accuracy is the stability of the local clock oscillator.

There are four appendices to this report. Appendix A describes the programming model and interface for a software driver that reads IRIG signals generated by some radio clocks using the audio codec native to some workstations. Appendix B describes the programming model and interface for the kernel PLL facility, including two new Unix system calls. Appendix C presents details of the PLL implementation, including the source code for a simulator designed to verify correct operation and calibrate performance. Appendix D is a description of our laboratory used for computer network timekeeping research and, in particular, the DCnet master clock.

2. Network Time Protocol

The Network Time Protocol (NTP) is used by Internet time servers and their clients to synchronize clocks, as well as automatically organize and maintain the time synchronization subnet itself. It is evolved from the Time Protocol [POS83] and the ICMP Timestamp Message [DAR81b], but is specifically designed for high accuracy, stability and reliability, even when used over typical Internet paths involving multiple gateways and unreliable networks. This section contains an overview of the architecture and algorithms used in NTP. A detailed description of the NTP architecture and

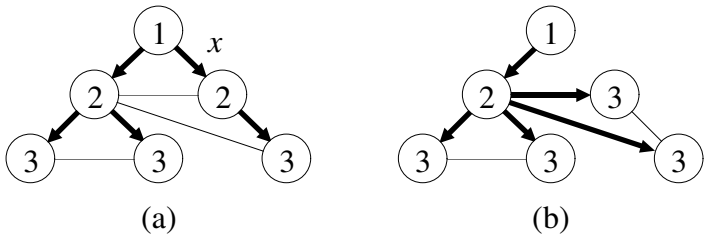

Figure 1. Subnet Synchronization Topologies

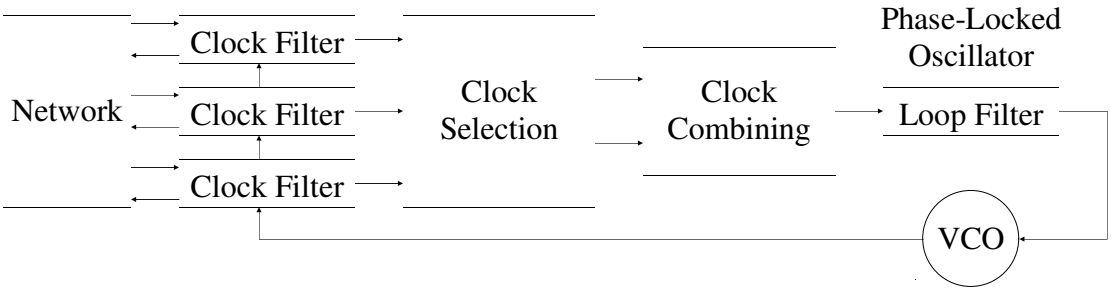

Figure 2. Network Time Protocol

protocol is contained in [MIL91a], while a summary of operational experience and performance is contained in [MIL90] and a detailed discussion on timescales is contained in [MIL91b]. A survey of implementation techniques and a detailed analysis of timekeeping errors is contained in [MIL92c]. The current version of the protocol is designated NTP Version 3 as defined by the specification document RFC-1305 [MIL92a]. A subset of the protocol, designated Simple Network Time Protocol (SNTP), is described in RFC-1361 [MIL92d].

NTP and its implementations have evolved and proliferated in the Internet over the last decade, with NTP Version 2 adopted as an Internet Standard (Recommended) [MIL89] and its successor NTP Version 3 adopted as a Internet Standard (Draft) [MIL92a]. NTP is built on the Internet Protocol (IP) [DAR81a] and User Datagram Protocol (UDP) [POS80], which provide a connectionless transport mechanism; however, it is readily adaptable to other protocol suites. The protocol can operate in several modes appropriate to different scenarios involving private workstations, public servers and various subnet configurations. A lightweight association-management capability, including dynamic reachability and variable poll-interval mechanisms, is used to manage state information and reduce resource requirements. Optional features include message authentication based on crypto-checksums and provisions for remote control and monitoring.

In NTP one or more primary servers synchronize directly to external reference sources such as radio clocks. Secondary time servers synchronize to the primary servers and others in the synchronization subnet. A typical subnet is shown in Figure 1a, in which the nodes represent subnet servers, with normal level or stratum numbers determined by the hop count from the primary (stratum 1) server, and the heavy lines the active synchronization paths and direction of timing information flow. The light lines represent backup synchronization paths where timing information is exchanged, but not necessarily used to synchronize the local clocks. Figure 1b shows the same subnet, but with the line marked *x* out of service. The subnet has reconfigured itself automatically to use backup paths, with the result that one of the servers has dropped from stratum 2 to stratum 3. In practice each NTP

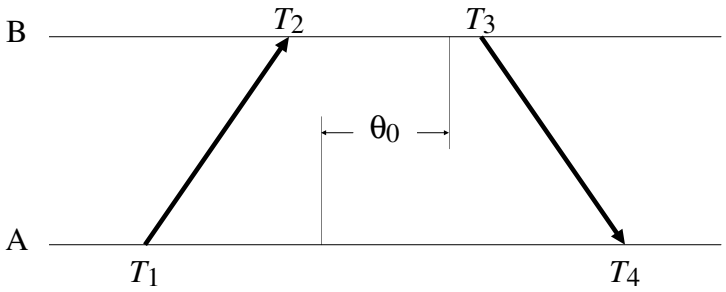

Figure 3. Measuring Delay and Offset

server synchronizes with several other servers in order to survive outages and Byzantine failures using methods similar to those described in [SHI87].

Figure 2 shows the overall organization of the NTP time-server model, which has much in common with the phase-lock methods summarized in [RAM90]. *Timestamps* exchanged between the server and possibly many other subnet peers are used to determine individual roundtrip delays and clock offsets, as well as provide reliable error bounds. As shown in the figure, the computed delays and offsets for each peer are processed by the clock-filter algorithm to reduce incidental timing noise. As described in [MIL92a], this algorithm selects from among the last several samples the one with minimum delay and presents the associated offset as the output.

The clock-selection algorithm determines from among all peers a suitable subset of peers capable of providing the most accurate and trustworthy time using principles similar to those described in [VAS88]. This is done using a cascade of two subalgorithms, one based on interval intersections to cast out faulty peers [MAR85] and the other based on clustering and maximum-likelihood principles to improve accuracy [MIL91a]. The resulting offsets of this subset are first combined on a weighted-average basis using the algorithm described in [MIL92a] and then processed by a phase-lock loop (PLL) using the algorithms described in [MIL92c]. In the PLL the combined effects of the filtering, selection and combining operations are to produce a phase-correction term, which is processed by the loop filter to control the voltage-controlled oscillator (VCO) frequency. The VCO is implemented as an adjustable-rate counter using a combination of hardware and software components. It furnishes the phase (timing) reference to produce the timestamps used in all timing calculations.

2.1. Calculating Offsets, Delays and Error Bounds

Figure 3 shows how NTP timestamps are numbered and exchanged between peers *A* and *B*. Let T_1, T_2, T_3, T_4 be the values of the four most recent timestamps as shown and, without loss of generality, assume $T_3 > T_2$. Also, for the moment assume the clocks of *A* and *B* are stable and run at the same rate. Let

$$a = T_2 - T_1 \quad \text{and} \quad b = T_3 - T_4 \ .$$

If the delay difference from *A* to *B* and from *B* to *A*, called *differential delay*, is small, the roundtrip delay δ and clock offset θ of *B* relative to *A* at time T_4 are close to

$$\delta = a - b \quad \text{and} \quad \theta = \frac{a+b}{2}.$$

Each NTP message includes the latest three timestamps T_1, T_2 and T_3, while the fourth timestamp T_4 is determined upon arrival of the message. Thus, both peers *A* and *B* can independently calculate delay and offset using a single bidirectional message stream. This is a symmetric, continuously sampled, time-transfer scheme similar to those used in some digital telephone networks [LIN80]. Among its advantages are that errors due to missing or duplicated message are avoided (see [MIL92c] for an extended discussion of these issues and an analysis of errors).

It is a simple exercise to bound the possible error of one peer relative to another when the roundtrip delay δ between the two is known. The true offset of *B* relative to *A* is called θ_0 in Figure 3. Let x denote the actual time between the departure of a message from *A* and its arrival at *B*. Therefore, $x + \theta_0 = T_2 - T_1 \equiv a$. Since x must be positive in our universe, $x = a - \theta_0 \geq 0$, which requires $\theta_0 \leq a$. A similar argument requires that $b \leq \theta_0$, so surely $b \leq \theta_0 \leq a$. This inequality can also be expressed

$$b = \frac{a+b}{2} - \frac{a-b}{2} \leq \theta_0 \leq \frac{a+b}{2} + \frac{a-b}{2} = a \ ,$$

which is equivalent to

$$\theta - \frac{\delta}{2} \leq \theta_0 \leq \theta + \frac{\delta}{2} \ .$$

In other words, the true offset θ_0 of peer *B* relative to *A* must lie somewhere in the interval of size δ with the apparent offset θ measured by *A* as its center. This is the best error bound obtainable, unless additional information about the differential delay is known.

The above development does not consider the effect of errors due to measurement precision, frequency tolerance and statistical delay variations. It is necessary at the outset to identify two types of error quantities. *Absolute error bounds* establish the maximum (worst-case) errors in a subnet where all time servers are operating correctly and synchronized to Coordinated Universal Time (UTC). For instance, the inequality above establishes the maximum error due to differential path delays. *Statistical error bounds* establish the estimated errors expected under nominal operating conditions and are computed from measured statistical variations. They include systematic quantities due to the local clock precision and frequency tolerance, as well as random quantities due to network queueing delays. Knowledge of estimated errors is useful in order to calibrate the performance in actual operation and mitigate among the several synchronization sources usually available at each level in the subnet hierarchy.

Let θ and δ be a sample of offset and delay computed as above. Associated with this sample is an estimated error statistic called the *dispersion* ε. It is determined by comparing successive sample offsets from a single peer and comparing offsets between possibly several peers used as synchronizing sources. There are three components of ε:

1. The maximum error ρ in reading the local clock, which depends on the clock resolution and method of adjustment.

2. The maximum error $\varphi\tau$ due to the frequency tolerance φ of the local clock times the period τ since last set.

3. An estimate of maximum probable error due to various delay variations in the network and statistical latencies in the operating systems.

In practice, errors due to network delays usually dominate the dispersion budget. However, it is not possible to characterize these delays as a stationary random process, since network queues can grow and shrink in chaotic fashion and packet arrivals are frequently bursty. However, the method of calculating ε as defined in the NTP Version 3 specification, represents a conservative estimate of the maximum error due to the above causes.

A statistic called the *synchronization distance* $\lambda = \frac{\delta}{2} + \varepsilon$ represents the maximum error contribution due to all causes. Thus, the above inequality becomes

$$\theta - \lambda \le \theta_0 \le \theta + \lambda .$$

By construction, the true offset θ_0 of a server relative to its client must lie somewhere in the interval of size 2λ with the apparent offset θ measured by the client as its center.

The above development considers only the error accumulation between two peers sharing an arc in the synchronization subnet. The accumulation of ε along the arcs of the subnet to the primary server is called the *root dispersion*, while the accumulation of λ is called the *root synchronization distance*. The intricate details on how to budget the error accumulation along these arcs are given in [MIL92c].

2.2. The Unix Local Clock Model

A computer clock includes some kind of reference oscillator stabilized by a quartz crystal or some other means, such as the power grid. The oscillator frequency is usually divided by a prescaler to a convenient frequency, such as 1 MHz or 100 Hz. This is followed by a clock counter, implemented in hardware, software or some combination of the two, which can be read by the operating system. For systems intended to be synchronized to an external source of standard time, there must be some means to correct the time and frequency by occasional vernier adjustments produced by a synchronization protocol such as NTP. Special care is necessary in all timekeeping system designs to insure that the clock indications are always monotonic increasing; that is, system time never "runs backwards." This is called the *monotonic requirement*.

The simplest computer clock consists of a hardware latch which is set by prescaler overflow and causes a processor interrupt. The latch is reset when acknowledged by the processor, which then increments the value of a software clock counter. The clock time is adjusted by adding corrections to the software counter as necessary. The clock frequency is adjusted by changing the value of the increment, in order to make the counter run faster or slower. The precision of this simple clock model, which is a software emulation of the *phase accumulation method* described in [WIL90], is limited to the value of the increment, usually in the range 1-10 ms. Some workstations, including the Fuzzball [MIL88] and Sun SPARCstation, include a high resolution counter which can be read by the kernel and used to interpolate between timer interrupts.

In some applications an external pulse-per-second (PPS) signal is available from a reference source such as a cesium clock or precision timing receiver. Such a signal generally provides much better precision than the serial ASCII character string produced by a radio clock and is typically in the low hundreds of nanoseconds. The PPS signal can be processed by an interface which produces a hardware interrupt coincident with the arrival of the PPS pulse. The processor then determines the

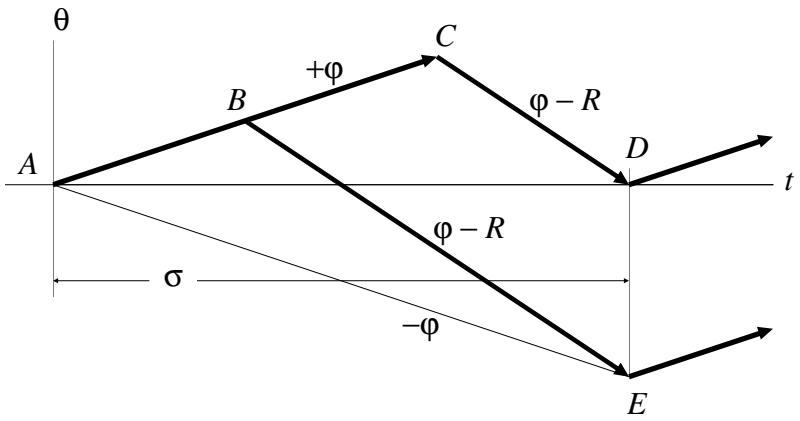

Figure 4. Clock Adjustment Process

current time and computes the residue modulo 1 s. Assuming the seconds numbering of the clock counter has been determined by a reliable source, such as an ordinary radio clock or even NTP itself, the final offset within the second can be determined from the residue.

The Unix 4.3bsd local clock model is based on two system calls, settimeofday() and adjtime(), together with two kernel variables *tick* and *tickadj*. The settimeofday() system call unceremoniously resets the local clock to the value given, while the adjtime() system call slews the local clock to a new value numerically equal to the sum of the present time of day and the (signed) argument given in the adjtime() call. In order to understand the behavior of the Unix clock as incorporated into precision timing systems, it is helpful to explore the operations of adjtime() in more detail.

The Unix clock model requires a periodic timer interrupt produced by an oscillator in the 100-1000 Hz range. Each interrupt causes an increment of tick to be added to the kernel *time* variable. The value of the increment is chosen so that the counter, plus an initial offset established by the settimeofday() call, is equal to the time of day in seconds and microseconds. When the tick does not evenly divide the second in microseconds, an additional variable *fixtick* is added to the kernel time once each second to make up the difference.

The Unix clock can actually run at three different rates, one at the intrinsic oscillator frequency $\frac{1}{tick}$, another at a slightly higher frequency and a third at a slightly lower frequency. Setting the *amortization rate* $R = \frac{tickadj}{tick}$ in parts per million (ppm) for convenience, these three rates correspond to frequency offsets of zero, $+R$ and $-R$ ppm respectively. Normally the zero offset is used; but, if adjtime() is called, the argument δ given is used to calculate the *amortization interval* $\Delta t = \frac{\delta}{R}$, during which either the $+R$ or $-R$ rate is used, depending on the sign of δ. The effect is to slew the clock to a new value at a small, constant rate, rather than incorporate the adjustment all at once, which could violate the monotonic requirement.

In Unix systems the values of tick and tickadj are expressed in integer microseconds. With common values of *tick* = 10,000 μs and *tickadj* = 4 μs, the amortization rate is $R = \pm 400$ ppm. The effective clock frequency can be adjusted by changing the value of tick, thus adjusting the frequency in steps of approximately 100 ppm. The amortization rate can be adjusted by changing the value of tickadj.

Figure 4 shows the offset θ as a function of time t for the Unix clock model. The oscillator has an assumed frequency tolerance of $\pm\varphi$ ppm, a fixed amortization rate $\pm R$ ppm and an adjustment interval σ s. In the diagram the heavy lines show the offsets for two cases, each labelled with the effective frequency offset. If no corrections are applied, the time offset grows at an assumed maximum positive rate φ as shown by the line *AC* in the diagram. The path *ACD* shows the offset for the case where the clock is steered at zero nominal rate and the line *ABE* for the case where it is steered at an assumed maximum negative rate $-\varphi$. For the first case the amortization interval Δt extends from *C* to *D*, when the clock resumes its intrinsic rate; while, in the second case it extends from *B* to *E*.

If θ is the true offset required at the end of the amortization interval, then

$$\theta = \varphi(\sigma - \Delta t) + (\varphi - R)\Delta t \; .$$

If ε is the maximum absolute error allowed, Δt must satisfy the inequality

$$\Delta t \le \frac{\varepsilon}{R - \varphi}$$

in order to steer the clock to a nominal rate of zero, $\theta = 0$. The above expressions yield the inequality

$$\sigma \le \frac{R}{\varphi} \frac{\varepsilon}{R - \varphi} \; .$$

In order to steer the clock to a nominal rate of $-\varphi$, $\theta = -\varphi\sigma$. According to the above, this requires $R \ge 2\varphi$ in order that the adjustments complete before the end of the adjustment interval.

For example, let the maximum absolute error be $\varepsilon = 100$ µs, oscillator frequency tolerance $\varphi = 200$ ppm and amortization rate $R = 400$ ppm. Substituting in the above expressions results in the inequality $\sigma \le 1$ s. The residual error ε inherent in the adjustment process is called *sawtooth error* elsewhere in this report. Obviously, ε can be reduced either by decreasing σ or reducing the frequency tolerance specification φ of the local clock oscillator. It is possible to do both of these things by modifying certain kernel routines, as described later in this report.

Protocols such as the Digital Time Synchronization Service (DTSS) [DEC89], which use adjtime() with adjustment intervals much greater than 1 s, can accumulate considerable sawtooth error between updates, especially if the intrinsic frequency error of the local clock oscillator is relatively large. There are several difficulties in the Unix model in addition to the sawtooth error. One of them is the integer division operation that computes Δt, which can produce adjustments only with a precision of tickadj. Errors due to this problem can accumulate to serious proportions in clock disciplining operations. These and related issues are discussed in a later section of this report and in Appendix B.

2.3. The NTP Local Clock Model

The NTP local clock model described in [MIL92c] incorporates the Unix local clock as a disciplined oscillator controlled by an adaptive parameter, type-II phase lock loop. Its characteristics are determined by the transient response of the loop filter, which for a type-II PLL includes an integrator with a lead network for stability. For use as a disciplining function in a computer clock, the NTP model can be implemented as a sampled-data system using a set of recurrence equations. A capsule overview of the design extracted from [MIL92c] may be helpful in understanding how the model operates.

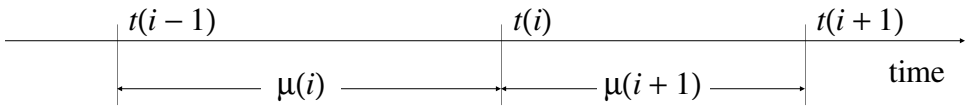

Figure 5. Update Nomenclature

The local clock is continuously adjusted in small increments at fixed adjustment intervals σ. The increments are computed from the values of the frequency error f and phase error g. These variables are computed from the timestamps in messages received at nominal *update intervals* μ, which are variable from about 16 s to over 17 minutes. As part of update processing, the *compliance* h is computed and used to adjust the *time constant* τ. Finally, the *poll interval* ρ for transmitted NTP messages is determined as a multiple of τ. Details on how τ is computed from h and how ρ is determined from τ are given in [MIL92a].

Updates are numbered from zero, with those in the neighborhood of the ith update shown in Figure 5. All variables are initialized at $i = 0$ to zero. After an interval $\mu(i) = t(i) - t(i-1)$ $(i > 0)$ from the previous update the ith update arrives at time $t(i)$ including the time offset $v_s(i)$. Then, after an interval $\mu(i+1)$ the i+1th update arrives at time $t(i+1)$ including the time offset $v_s(i+1)$ and time constant $\tau(i)$. When the update $v_s(i)$ is received, the frequency error $f(i+1)$ and phase error $g(i+1)$ are computed:

$$f(i+1) = f(i) + \frac{\mu(i) v_s(i)}{\tau^2}, \quad g(i+1) = \frac{v_s(i)}{\tau}.$$

The factor τ in the above has the effect of adjusting the bandwidth of the PLL as a function of the prevailing network dispersions determined by the NTP daemon. When the dispersion has been low over some relatively long period, τ is increased and the bandwidth is decreased. In this mode small timing fluctuations due to jitter in the subnet are suppressed and the PLL attains the most accurate frequency estimate. On the other hand, if the dispersion becomes high due to network congestion or a systematic frequency change, for example, τ is decreased and the bandwidth is increased. In this mode the PLL is most adaptive to transients due to these causes and others due to system reboot or a missed timer interrupt.

A model suitable for simulation and parameter refinement can be constructed from the above recurrence relations. It is convenient to set the temporary variable $a = g(i+1)$. At each adjustment interval σ the quantity $\frac{a}{K_g} + \frac{f(i+1)}{K_f}$ is added to the local-clock time and the quantity $\frac{a}{K_g}$ is subtracted from a. For convenience, let n be the greatest integer in $\frac{\mu(i)}{\sigma}$; that is, the number of adjustments that occur in the ith interval. Thus, at the end of the ith interval just before the i+1th update, the VCO control voltage is:

$$v_c(i+1) = v_c(i) + [1 - (1 - \frac{1}{K_g})^n]\, g(i+1) + \frac{n}{K_f} f(i+1).$$

In the original NTP implementation, the NTP daemon simulates the above recurrence relations and provides offsets to the kernel at intervals of $\sigma = 1$ s using the adjtime() system call. With this value of σ and an assumed frequency tolerance $\varphi = 100$ ppm, which is a reasonably conservative

assumption, the maximum sawtooth error is 100 μs, which is a reasonable goal for machines in the DEC Alpha class. This is in fact what the original NTP daemon does. However, provisions have to be made for the additional jitter which results when the timer interval does not divide the second in microseconds. Also, since the adjustment process must complete within 1 s, larger adjustments must be parcelled out in a series of adjtime() calls. Finally, provisions must be made to compensate for the roundoff error in computing Δt. These factors add to the error budget, increase system overhead and complicate the daemon implementation.

3. Hardware and Software Interfaces for Precision Timekeeping

In general, a real-time, distributed application is driven by events such as the expiry of an interval timer, the arrival of a network message or completion of an input/output operation. These events can result in actions created by the application itself, such as transmitting a message or initiation of an input/output operation or timer operation. Some of these events and actions are internal to the application and their associated event times or epoches may not be visible to other clients in the distributed application. Others may result in externally visible events which must be accurately known relative to the global network time.

The timing accuracy of a distributed application depends on the accuracy and stability with which its local clock can be set, maintained and read. These issues are discussed at length in [MIL92c], which develops a model for the local clock and budgets its error accumulations from the primary server along the arcs of the synchronization subnet. While the report presents a model and analysis, it does not quantify the errors in actual hardware and software systems, nor does it present any discussion on how to minimize these errors. For some sources of error an engineering solution is impractical, such as to eliminate statistically varying network delays, for example. For others, such as interface jitter, interrupt latency and oscillator instability, there are engineering solutions which, if not eliminating the errors entirely, can reduce their effect to minor proportions.

It has been demonstrated in previous reports that it is possible using NTP to synchronize a number of hosts on an Ethernet or a moderately loaded T1 network within a few tens of milliseconds with no particular care in selecting the radio clock or configuring the servers on the network. This may be adequate for the majority of applications; however, modern workstations and high speed networks can do much better than that, generally to within some fraction of a millisecond, by taking special care in the design of the hardware and software interfaces.

The timekeeping accuracy of a NTP-synchronized host depends on two quantities: the fixed delay due to hardware and software processing and the accumulated jitter due to such things as clock reading precision and varying latencies in hardware and software queuing. Processing delays directly affect the timekeeping accuracy, unless minimized by systematic analysis and adjustment. Jitter, on the other hand, can be essentially suppressed, if the statistical properties are unbiased, by the low-pass filtering of the PLL incorporated in the NTP local clock model.

Much of the discussion in this report is on issues related to the design of hardware and software interfaces for external time sources such as radio clocks and associated timing signals. However, there are similar issues related to the design of network interfaces such as for Ethernet or FDDI as well, but these will not be discussed further here. Radio clocks are most often connected to a processor using a serial asynchronous port. Much of the discussion in the following has to do with ways in which the delays incurred in this type of connection can be controlled and ways in which the jitter due to various causes can be minimized.

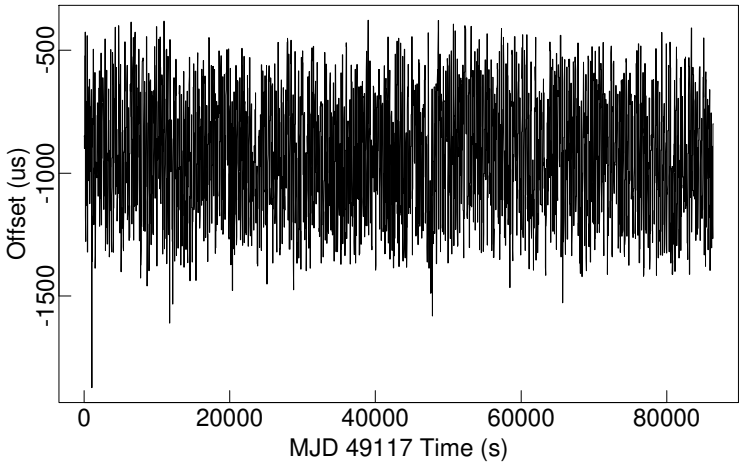

Figure 6. Timing Offsets with Serial ASCII Timecode

There are ways other than serial ports to interface a radio clock, including special purpose hardware devices which can capture special signals generated by some radio clocks and discipline an onboard clock oscillator directly. Many of these methods can yield accuracies much better than attainable with a serial port. For those radio clocks equipped with an IRIG-B signal output, for example, a hardware interface is available for the Sun SBus and perhaps other bus architectures as well, but it is quite expensive ($2,500). Since these devices are expensive and have limited application, they will not be discussed further in this report.

However, there is an inexpensive, elegant solution for precision timekeeping, at least in the case of certain architectures including an integral audio port and codec in the processor circuitry. This scheme uses the audio codec to demodulate the special IRIG signal generated by some radio clocks and use it to discipline the local clock. The design and implementation of this scheme is discussed below, while the programming model and calling sequences are discussed in Appendix A.

3.1. Connection via Serial Port

Most radio clocks produce an ASCII timecode with a precision only to the millisecond. This results in a maximum peak-to-peak (p-p) jitter in the clock readings of one millisecond. However, assuming the read requests are statistically independent of the clock update times, the reading error is uniformly distributed over the millisecond, so that the average over a large number of readings will make the clock appear 0.5 ms late. To compensate for this, it is only necessary to add 0.5 ms to the reading before further processing by the NTP algorithms. For example, Figure 6 shows the time offsets between a WWVB receiver and the local clock over a typical day. The readings appear to be uniformly distributed over the interval -400 to -1400 μs, with mean about -900 μs; thus, the true offset of the radio clock would be -400 μs.

Most radio clocks can produce a PPS signal in addition to the ASCII timecode. The precision of this signal is usually much better than the ASCII timecode; however, its accuracy is limited by the nature of the time dissemination service, generally a few milliseconds for WWV/WWVH and CHU receivers, 100 μs for WWVB receivers and better than 1 μs for GOES and GPS receivers. When necessary to distinguish between ordinary radio clocks with millisecond resolution and receivers with substantially better PPS precision, the latter will be called *timing receivers*. Figure 7, which

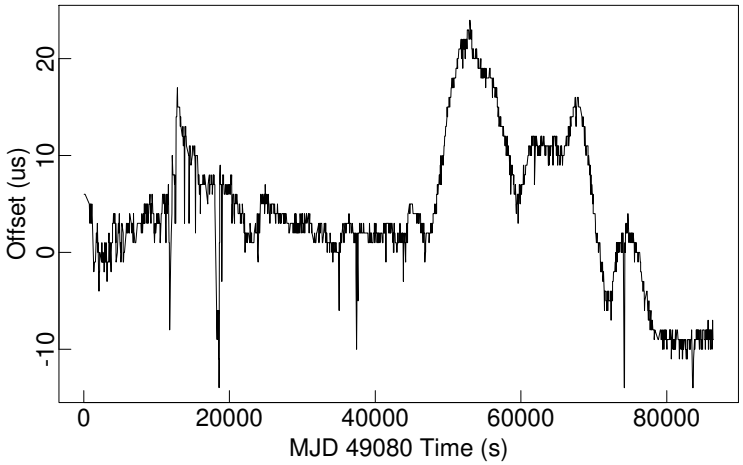

Figure 7. Timing Offsets with PPS Signal

shows the offsets between the local clock and precision PPS signal measured using special kernel modifications described later.

Radio clocks are usually connected to the host computer using a serial port operating at a speed of 9600 bps. The on-time reference epoch for the timecode is usually the start bit of a designated character of the timecode. The UART chip implementing the serial port most often has a sample clock of eight to 16 times the basic bit rate. Assuming the sample clock starts midway in the start bit and continues to midway in the first stop bit and there are eight bits per character, this creates a processing delay of 10.5 bit times, or about 1.1 ms, relative to the start bit of the character. The jitter contribution is usually no more than a couple of sample-clock periods, or about 26 μs p-p. This is small compared to the clock reading jitter and can be ignored. Thus, the UART delay can be considered constant, so the hardware contribution to the total mean delay budget is 0.5 + 1.1 = 1.6 ms.

In some kernel serial port drivers, in particular, the Sun zs driver, an intentional delay is introduced when characters are received after an idle period. A batch of characters is passed to the calling program when either (a) a timeout in the neighborhood of 10 ms expires or (b) an input buffer fills up. The intent in this design is to reduce the interrupt load on the processor by batching the characters where possible. Obviously, this can cause severe problems for precision timekeeping. Judah Levine of the National Institute of Science and Technology (NIST) has developed patches for the zs driver [personal communication] which fixes this problem for the native serial ports of the Sun SPARCstation. The problem does not appear to be present in the Serial/Parallel Controller (SPC) for the SBus, which contains eight serial asynchronous ports along with a parallel port. The measurements referred to below were made using this controller.

Good timekeeping depends strongly on the means available to capture an accurate timestamp at the instant the stop bit of the on-time character is found; therefore, the code path delay between the character interrupt routine and the first place a timestamp can be captured is very important, since on some systems, such as Sun SPARCstations, this path can be astonishingly long. The Unix scheduling mechanisms involve both a hardware interrupt queue and a software interrupt queue. Entries are made on the hardware queue as the interrupt is signalled and generally with the lowest latency, estimated at 20-30 microseconds (μs) for a Sun IPC. Then, after minimal processing, an entry is made on the software queue for later processing in order of software interrupt priority.

Finally, the software interrupt unblocks the NTP daemon, which then calculates the current local clock offset and introduces corrections as required.

Opportunities exist to capture timestamps at the hardware interrupt time, software interrupt time and at the time the NTP daemon is activated, but these involve various degrees of kernel trespass and hardware gimmicks. To gain some idea of the severity of the errors introduced at each of these stages, measurements were made using a Sun SPARCstation IPC and a test setup that results in an error between the host clock and a precision timing source (calibrated cesium clock) no greater than 0.1 ms. The total delay from the on-time epoch to when the NTP daemon is activated was measured at 8.3 ms in an otherwise idle system, but increased on rare occasion to over 25 ms under load, even when the NTP daemon was operated at a relatively high software priority level. Since 1.6 ms of the total delay is due to the hardware, the remaining 6.7 ms represents the total code path delay accounting for all software processing from the hardware interrupt to the NTP daemon.

It is commonly observed that the latency variations (jitter) in typical real-time applications scale very roughly as the nominal processing delay. In the case above, the ratio of the maximum observed delay (25 ms) to the baseline code path delay (8.3 ms) is about three. It is natural to expect that this ratio remain the same or less as the code path between the hardware interrupt and where the timestamp is captured is reduced. However, in general this requires trespass on kernel facilities and/or making use of features not common to all or even most Unix implementations. In order to assess the cost and benefits of increasingly more aggressive insult to the hardware and software of the system, it is useful to construct a budget of the code path delay at each of the timestamp opportunity times. For instance, on Unix systems which include support for the SIGIO facility, it is possible to intervene at the time the software interrupt is serviced. The NTP daemon code uses this facility, when available, to capture a timestamp and save it along with the timecode data in a buffer for later processing. This reduces the total code path delay from 6.7 ms to 3.5 ms on an otherwise idle system. This design is used for all input processing, including network interfaces and serial ports.

3.1.1. The CLK Facility

By far the best place to capture a serial-port timestamp is right in the kernel interrupt routine, but this generally requires intruding in the kernel code itself, which can be intricate and architecture dependent. The next best place is in some routine close to the interrupt routine on the code path. There are two ways to do this, depending on the ancestry of the Unix operating system variant. Older systems based primarily on the original Unix 4.3bsd support *line discipline* modules, which are hunks of code with more-or-less well defined interface specifications that can get in the way, so to speak, of the code path between the interrupt routine and the remainder of the serial port processing. Newer systems based on System V STREAMS can do the same thing using *streams* modules.

Both approaches are supported in the NTP Version 3 distribution, as described in the documentation included therein. In either case, header and source files have to be copied to the kernel build tree and certain tables in the kernel have to be modified. In neither case, however, are kernel sources required. In order to take advantage of this, the clock driver must include code to activate the feature and extract the timestamp. At present, this support is included in several clock drivers, including the Spectracom WWVB receiver, the PSTI/Traconex WWV/WWVH receiver and the PPS signal described later. If justified, support can be easily added to most other clock drivers as well. This support is called the CLK facility.

The CLK line discipline and streams module operate in the same way. They look for a designated character, usually <CR>, and stuff a Unix *timeval* timestamp in the data stream following that character whenever it is found. Eventually, the data arrive at the particular clock driver configured in the NTP Version 3 distribution. The driver then extracts the timestamp as a precise reference epoch, subject to the earlier processing delays and jitter budget, for future reference. In order to gain some insight as to the effectiveness of this approach, measurements were made using the same test setup described above. The total delay from the on-time epoch to the instant when the timestamp is captured was measured at 3.5 ms. Thus, the code path delay is this value less the hardware delay 3.5 - 1.6 = 1.9 ms.

While the reduction in code path delay to 1.9 ms from the baseline delay of 8.3 ms is significant, there is another factor, at least in Sun systems, that makes it even more worthwhile. When processing the code path up to the CLK module, the priority is apparently higher than for processing beyond it. In case of heavy processor activity, this can lead to relatively long tails in the processing delays for the driver, which of course are avoided by capturing the timestamp early in the code path.

3.1.2. The PPSCLK Facility

Many radio clocks produce a PPS signal of considerably better precision than the ASCII timecode. Using this signal, it is possible to avoid the 1-ms p-p jitter and 1.6 ms hardware timecode adjustment entirely. However, a device called a *gadget box* is required to interface this signal to the hardware and operating system. The gadget box includes a level converter and pulse generator that turns the PPS signal on-time transition into a valid character. A circuit schematic, circuit board artwork and construction details for such a device are included in the NTP Version 3 distribution. Although many different circuit designs could be used as well, this particular device generates a single 26-μs start bit for each PPS signal on-time transition. This appears to the UART operating at 38.4K bps as an ASCII DEL (hex FF).

Now, assuming a serial port can be dedicated to this purpose, the PPS character interrupts can be used to provide a precision reference. The NTP Version 3 daemon can be configured to utilize this feature. The character resulting from each PPS signal on-time transition is intercepted by the CLK facility and a timestamp inserted in the data stream. An interrupt is created for the device driver, which reads the timestamp and discards the DEL character. Since the timestamp is captured at the on-time transition, the seconds-fraction portion is the offset between the local clock and the on-time epoch less the UART delay. If the local clock is within ±0.5 s of this epoch, as determined by other means, the local clock correction is taken as the offset itself, if between zero and 0.5 s, and the offset minus one second, if between 0.5 and 1.0 s.

In the NTP daemon a 12-stage shift register holds the most recent corrections determined as above. when the register fills up, it is sorted and the two highest and two lowest samples discarded. The remaining eight samples are averaged and become the update for the local clock. There are two reasons for this type of processing, one to improve accuracy and the other to provide a measure of dispersion, which is the difference between the highest and lowest values in the register. The dispersion is used by the clock selection algorithm to mitigate among possibly many sources of synchronization, only one of which might be the PPS signal.

The baseline delay between the on-time transition and the timestamp capture was measured at 400±10 μs on an otherwise idle SPARCstation IPC. As the UART delay at 38.4K bps is about 270 μs, the difference, 130 μs, must be due to the hardware interrupt latency plus the time to call the

kernel routine which actually reads the local clock. For these measurements the assembly-coded version of this routine included in the NTP Version 3 distribution was used, rather than the C-coded routine included in the native kernel. This routine reduces the time to read the local clock from 42-85 μs with the native routine to about 3 μs with the assembly-coded routine, so can be ignored. Thus, the 130 μs must be accounted for in interrupt service, register window, context switching, streams operations and measurement uncertainty, which is probably not unreasonable. The reason for the difference between this figure and the previously calculated value of 1.9 ms for the CLK facility and serial ASCII timecode is probably due to the fact that all streams modules other than the CLK module were removed, since the serial port is not used for ordinary ASCII data.

An interesting feature of this approach is that the PPS signal is not necessarily associated with any particular radio clock and, indeed, there may be no such clock at all. Some precision timekeeping equipment, such as cesium clocks, VLF receivers and LORAN-C receivers produce only a precision PPS signal and rely on other mechanisms to resolve the second of the day and day of the year. It is possible for an NTP-synchronized host to derive the latter information using other NTP peers, presumably properly synchronized within ±0.5 second, and to remove residual jitter using the PPS signal. This makes it quite practical to deliver precision time to local clients when the subnet paths to remote primary servers are heavily congested. In extreme cases like this, it has been found useful to increase the tracking aperture from ±128 ms to as high as ±512 ms. This scheme is now in use at the Norwegian Telecommunications Research Establishment in Oslo, Norway.

In the current implementation the ASCII timecode and PPS signal are separately processed. The timecode capture and CLK facility, if provided by the radio clock driver, operate the same way whether or not the PPSCLK facility is enabled. If the local clock is reliably synchronized within ±0.5 s and the PPS signal has been valid for some number of seconds, its offset rather than whatever synchronization source has been selected is used instead. However, while this procedure delivers a new offset estimate every 12 seconds, the local clock is updated only as each valid update is computed for the peer selected as the source of synchronization.

There is a hazard in the use of the PPS signal if the radio clock generating the signal misbehaves or loses synchronization with the transmitter. In such a case the radio clock might indicate the error in the timecode, but the system has no way to associate the error with the PPS signal. To deal with this problem, a special “prefer” parameter documented in the NTP Version 3 distribution can be used both to encourage the clock selection algorithm to choose a specified peer, all other things being equal, as well as associate the error indications in such a way that the PPS signal will be disregarded if the peer stops providing valid updates. The prefer parameter can be used in other situations as well when preference is to be given a particular source of synchronization.

3.1.3. The PPS Facility

For the ultimate accuracy and lowest jitter, it would be best to eliminate the UART entirely and capture the PPS on-time transition directly using an appropriate interface. This is in fact possible using a modified serial port driver and modem status lead in the serial port interface cable. In this scheme, described in detail in the NTP Version 3 distribution, the PPS source is connected via a gadget box to the carrier-detect lead of a serial port. Happily, this can be the same port used for a radio clock, for example, or another unrelated serial device. The scheme, referred to subsequently as the PPS facility, is specific to the SunOS 4.1.x kernel and requires a special streams module.

Except for special-purpose interface modules, such as the KSI/Odetics TPRO IRIG-B decoder and the modified audio driver for the IRIG-B signal to be described, the PPS facility provides the most accurate and precise timestamp available. There is essentially no latency and the timestamp is captured within 10-30 μs of the on-time epoch, depending on the system architecture.

The PPS facility requires a serial port to be selected as the PPS interface; however, the PPS signal path has nothing to do with the ordinary serial data path; the two signals are not related, other than by the need to activate the PPS facility and pass the file descriptor to a common processing routine. In case of multiple radio clocks on a single time server, the PPS configuration is necessary on only one of them; more than one PPS configuration would be an error.

The PPS facility works just like the CLK facility in the treatment of the prefer parameter and behavior with peer errors. As in the CLK facility, only the offset within the second is used and only when the offset is less than ±0.5 s. However, the precision of the clock adjustments is usually so fine that the error budget is dominated by the inherent stability of the local clock oscillator. Therefore, it is advisable to reduce the poll interval for the preferred peer from the default 64 s to something less, like 16 s. This is done using the minpoll and maxpoll parameters of the configuration command associated with the clock. These parameters take as arguments a power of 2, in seconds, which becomes the poll interval and, indirectly, affects the bandwidth of the PLL.

3.2. The IRIG Facility

The preceding discussion demonstrates the extraordinary intricacies necessary to achieve good timekeeping performance using ordinary serial ports and no special interface hardware. In point of fact, there are special-purpose peripherals designed just for this purpose, such as various types of counter/timer bus peripherals for a variety of workstations, but these are moderately to very expensive, making them incompatible with a ubiquitous, general-purpose timekeeping service.

Most of the discussion on serial ports was concerned with the PPS signal available on some radio clocks. This has the disadvantage that two interfaces are required, one for the PPS signal and the other for the ASCII timecode. It would be most convenient if these two signal could be combined. There is another signal produced by some radio clocks that can be used for this purpose, the Inter-Range Instrumentation Group (IRIG) signal, which was developed to synchronize instrumentation recorders in the early days of the U.S. space program. A modified form of this signal is also used in the radio broadcasts of the NIST time and frequency dissemination services. There are several timing receivers now on the market that can produce IRIG signals, including those made by Austron, TrueTime, Odetics and Spectracom, among others.

Among the several IRIG formats is one particularly suited for computer clock synchronization and designated IRIG-B. It consists of an amplitude-modulated 1000-Hz sinewave, where each symbol is encoded as ten full carrier cycles, or 10 ms in duration. The symbols are distinguished using a pulse-width code, where 2 ms corresponds to logic zero, 5 ms to logic one and 8 ms to a position identifier used for symbol synchronization. The complete IRIG-B message consists of a frame of ten fields, each field consisting of a nine information symbols followed by a position identifier for a total frame duration of one second. The first symbol in the frame is also a position identifier to facilitate frame synchronization. The 100-Hz modulation is synchronous with the 1000-Hz carrier and the zero crossing of the first carrier cycle of the frame coincides with the on-time epoch of the second as transmitted. The IRIG-B signal modulation encodes the day of year and time of day in binary-coded decimal (BCD) format, together with a set of control functions used for housekeeping.

Roy LeCates at the University of Delaware designed and implemented an IRIG driver with which the IRIG-B signal can be connected to the audio codec of some workstations, demodulated and used to synchronize the local clock. There are two components of the IRIG facility, a set of modifications to the BSD audio driver for the Sun SPARCstation, which was designed and implemented by Van Jacobson and collaborators at Lawrence Berkeley National Laboratory, and a companion clock driver for the NTP Version 3 daemon. This scheme does not require any external circuitry other than a resistor voltage divider, but can synchronize the local clock in principle to within a few tens of microseconds.

In operation, the 1000-Hz modulated IRIG signal is sampled at an 8-kHz rate by the audio codec and processed by the IRIG driver, which synchronizes to the carrier and frame phase. The driver then demodulates the symbols and develops a raw binary timecode and decoded ASCII timecode. A good deal of attention was paid in the software design to noise suppression and efficient demodulation technique. A matched filter is used to synchronize the frame and the zero crossing determined by interpolation to within a few microseconds. An automatic gain control function is implemented in order to cope with varying IRIG signal levels and codec sensitivities.

At the on-time epoch a timestamp is captured from the local clock and adjusted for the computed phase of the carrier signal relative to the 8-kHz codec sample clock. This is done as early in the code path as possible, in order to minimize residual errors. When the clock driver issues a read() system call, the IRIG driver returns the most recent timecode data, including a status byte and the corrected timestamp in a structure. Depending on the frequency with which the read() system call is called, this may result in old data or duplicate data or even invalid data, should the driver be called before it has computed its first timestamp.

In practice, the resulting ambiguity causes few problems. The clock driver converts the ASCII timecode returned by the read() system call to Unix *timeval* format and subtracts it from the kernel timestamp included in the structure. The result is an adjustment that can be subtracted from the kernel time, as returned in a gettimeofday() call, to correct for the deviation between IRIG time and kernel time. The result can always be relied on to within 128 μs, the audio codec sampling interval, and ordinarily to within a few microseconds, as determined by the interpolation algorithm.

Appendix A contains a description of the programming model, including data structures and calling sequences used by the IRIG driver and NTP clock driver. Either the ASCII timecode or the combined raw timecode and ASCII timecode can be returned in response to a read() system call. The ASCII timecode is in the handy format: “ddd hh:mm:ss*” for convenience in client programs. In this format the “*” status character is ASCII space “ ” when the driver is operating normally and “?” when errors may be present, as described in Appendix A.

In practice, the timekeeping accuracy of the IRIG facility is comparable to the accuracy achievable with a PPS signal and modified serial port driver (PPS facility) described previously. However, the IRIG facility requires only a simple resistor voltage divider, and not the more complex pulse generator and level converter required with the CLK and PPS facilities. One disadvantage of available IRIG signal generators is the absence of year, leap warnings and, for some generators, equipment health indicators. However, it is the practice in at least some IRIG generators to remove the modulation if synchronization with the primary source is lost. The IRIG driver detects this condition as a protocol error and lights a bit in the status byte.

Finally, while the NTP Version 3 daemon has been ported to a number of different architectures and operating systems, the modified BSD audio driver operates at present only with Sun SPARCstations running the SunOS 4.1.x kernel. A project is under way now to provide equivalent functionality in Digital Equipment architectures, either using the DECaudio TurboChannel peripheral or the native audio codec included in Alpha workstations.

4. Unix Kernel Modifications for Precision Timekeeping

On the realization that modern workstations are not necessarily designed to be good timekeepers, it is not surprising that some are better than others. As shown in [MIL90] and later in this report, most modern Unix workstations and servers can keep reliable time to the low milliseconds, almost all of the time. However, there are certain well known exceptions, such as when the Unix kernel writes system error messages to the console printer, or when performing certain disk synchronization checks. During these procedures interrupts can be disabled, with result that hardware timer interrupts can be lost and the time-of-day clock disrupted. Experience has shown these disruptions are rare and usually due to something broken. In fact, on several occasions timing glitches have been the first indicator of a failing hardware component.

Notwithstanding these disruptions, there are two areas where system timekeeping quality can be significantly improved without major impact on existing Unix kernel software. One of these reduces the jitter inherent in some kernel implementations when reading the local clock. This can be done in certain workstation architectures where hardware support for a high resolution clock is available, but for some reason is not utilized by the kernel implementation. The other moves the clock adjustment functions normally associated with a disciplined oscillator from the synchronization daemon directly to the kernel. This reduces residual frequency errors and sawtooth errors, such as those shown in Figure 4, essentially to zero.

The following sections describe modifications to Unix kernel routines that manage the local clock and timer functions. They provide improved accuracy and stability through the use of a disciplined oscillator model and interface for use with the Network Time Protocol (NTP) or similar time-synchronization protocol. The NTP Version 3 daemon automatically detects the presence of the modifications and changes behavior accordingly. There are three versions of this software, one each for the Sun SPARCstation with the SunOS 4.1.x kernel, Digital Equipment DECstation 5000 with Ultrix 4.x kernel and Digital Equipment 3000 AXP Alpha with the OSF/1 V1.x kernel. The software involves minor changes to the local clock and interval timer routines and includes interfaces for application programs to learn the local clock status and certain statistics of the time-synchronization process.

The principal feature added to the kernels is to change the way the local clock is controlled, in order to provide precision time and frequency adjustments. Another feature of the Ultrix and OSF/1 kernel modifications improves the local clock resolution to 1 μs. This feature can in principle be used with any Ultrix-based or OSF/1-based machine that has the required hardware counters, although this has not been verified. Other than improving the local clock resolution, the addition of these features does not affect the operation of existing Unix system calls which affect the local clock, such as gettimeofday(), settimeofday() and adjtime().

The design principles of the new kernel software are outlined in following sections, including the design of the PLL code, precision time resolution code and extraordinarily idiosyncratic leap-second code. A detailed description of the programming model, including data structures and calling

sequences, is given in Appendix B, while Appendix C describes details of the PLL implementation using a purpose-built simulator to facilitate the exposition. Detailed installation instructions are given in the software distributions. However, the software distributions are provided only by special arrangement, since they involve changes to licensed code.

4.1. Design Principles

As described previously, in the original Unix design a hardware timer interrupts the kernel at a fixed rate: 100 Hz in the SunOS kernel, 256 Hz in the Ultrix kernel and 1024 Hz in the OSF/1 kernel. Since the Ultrix timer interval (reciprocal of the rate) does not evenly divide one second in microseconds, the kernel adds 64 microseconds once each second, so the timescale consists of 255 advances of 3906 μs plus one of 3970 μs. Similarly, the OSF/1 kernel adds 576 μs once each second, so its timescale consists of 1023 advances of 976 μs plus one of 1552 μs. In this design and with the original NTP daemon design, the adjustment interval σ is fixed at 1 s.

In the original NTP implementation, the NTP daemon provides offset adjustments to the kernel at periodic adjustment intervals using the adjtime() system call. However, this process is complicated by the need to parcel out large adjustments and to compensate for roundoff error. In the new software this scheme is replaced by another that represents the local clock as a multiple-word, precision time variable in order to provide very precise clock adjustments. At each timer interrupt a precisely calibrated quantity is added to the time variable and carries propagated as required. The quantity is computed as in the NTP local clock model, which operates as a type-II phase lock loop. This PLL can provide precision control of the local clock oscillator phase to within ±1 μs and frequency to within ±5 ns per day. With this scheme σ becomes the timer interrupt interval and the sawtooth error reduced as the reciprocal of the timer rate.

The type-II PLL model is identical to the one implemented in the NTP Version 3 daemon *xntpd*, except that the daemon needs to call the kernel only as each new update is received at update intervals μ, not at the much smaller adjustment intervals σ required by the original scheme. In addition, the need to parcel large updates, account for odd timer rates and compensate for roundoff error is completely avoided. The modified kernel routines do these things automatically and economically at the expense of only a small amount of new code in the kernel. The amount of code added to the kernel for the new scheme is about the same as removed if the original scheme is deleted. However, in the interest of flexibility and consistent semantics, both schemes are retained. In addition, doing the frequency correction in the kernel means that the system time runs true even if the daemon ceases operation or the network paths to the primary reference source fail.

In the new scheme, a system call ntp_adjtime() operates in a way similar to the original adjtime(), but does not affect the original system call, which continues to operate in its traditional fashion. It is the intent in the design that settimeofday() or adjtime() be used for changes in system time greater than ±128 ms. It has been the Internet experience that the need to change the system time in increments greater than ±128 ms is extremely rare and is usually associated with a hardware or software malfunction or system reboot. The easiest way to do this is with the settimeofday() system call; however, this can under some conditions cause the clock to jump backward. If this cannot be tolerated, adjtime() can be used to slew the clock to the new value without running backward or affecting the frequency discipline process.

Once the local clock has been set within ±128 ms, the ntp_adjtime() system call is used to provide periodic updates including the time offset, maximum error, estimated error and PLL time constant.

With NTP the update interval depends on the measured error and time constant; however, the scheme is quite forgiving and neither moderate loss of updates nor variations in the length of the polling interval are serious. The engineering architecture constants given in Appendix B have been optimized for update intervals in the order of 16 s. For other intervals the PLL time constant can be adjusted to optimize the dynamic response up to intervals of 1024 s. Normally, this is automatically done by NTP.

In order to preserve the correctness assertions in the protocol specification, the new kernel software adjusts the maximum error to grow by an amount equal to the specified oscillator frequency tolerance times the elapsed time since the last update. In any case, if updates are suspended, the PLL coasts at the frequency last determined, which usually results in the maximum error increasing only to a few tens of milliseconds over a day.

The new code needs to know the initial frequency offset and time constant for the PLL, and the daemon needs to know the current frequency offset computed by the kernel for monitoring purposes. These data are exchanged between the kernel and protocol daemon using ntp_adjtime() as documented in Appendix B. Provisions are made to exchange related timing information, such as the maximum error and estimated error, between the kernel and daemon and between the kernel and application programs.

While any protocol daemon can in principle be modified to use the new system calls, the most likely will be users of the NTP Version 3 daemon xntpd. The xntpd code determines whether the new system calls are implemented and automatically reconfigures as required. When implemented, the daemon reads the frequency offset from a file and provides it and the initial time constant via ntp_adjtime(). In subsequent calls to ntp_adjtime(), only the time adjustment and time constant are affected. The daemon reads the frequency from the kernel using ntp_adjtime() at intervals of about one hour and writes it to the system log file. This information is recovered when the daemon is restarted after reboot, for example, so the sometimes extensive training period to learn the frequency separately for each system can be avoided.

4.2. Precision Time Resolution

For the Digital Equipment DECstation 5000/240 and possibly other machines of that family, there is an undocumented IOASIC hardware register that counts system bus cycles at a rate of 25 MHz. The new microtime() routine for the Ultrix kernel uses this register to interpolate system time between hardware timer interrupts. This results in a precision of ±1 μs for all time values obtained via the gettimeofday() and ntp_gettime() system calls. For the Digital Equipment 3000 AXP Alpha, the architecture provides a hardware Process Cycle Counter and a machine instruction *rpcc* to read it. This counter operates at the fundamental frequency of the CPU clock or some submultiple of it, 133.333 MHz for the 3000/400 for example. The new microtime() routine for the OSF/1 kernel uses this counter in the same fashion as the Ultrix routine uses the IOASIC counter.

The SunOS kernel already includes a time-of-day clock with microsecond resolution; so, in principle, no microtime() routine is necessary. There is in fact an existing kernel routine *uniqtime()* which implements this function, but it is coded in the C language and is rather slow, as mentioned above. A replacement microtime() routine coded in assembler language is available and is much faster at about 2 μs per call. The replacement routine is included in the NTP Version 3 distribution.

In both the Ultrix and OSF/1 kernels the gettimeofday() and ntp_gettime() routines call the microtime() routine, which returns the actual interpolated value, but does not change the kernel time

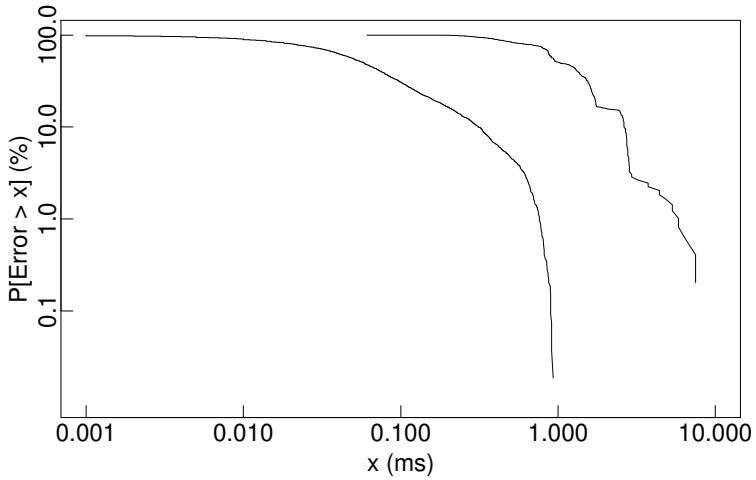

Figure 8. Probability of Error with Ultrix Kernel

variable. Therefore, other kernel routines that access this variable directly and do not call either gettimeofday(), ntp_gettime() or microtime() will continue their present behavior. The microtime() feature is independent of other features described here and is operative even if the kernel PLL or new system calls have not been implemented.

For many applications it is important that the local clock never violate the monotonic requirement. With precisions in the microsecond regimes and frequency corrections that can reach many parts per million, it is inevitable that, unless special precautions are taken, this requirement will be violated, even if by a very small number of microseconds. In the case of all three kernels considered in this document, the microtime() routines have been carefully crafted to avoid this behavior. These routines remember the time returned at the last call and force the next value to be returned to be at least one greater than the last time, even if this is not the case for the actual new time. However, these routines are also careful not to do this if the new time is more than one second in the past on the assumption this cannot occur with a system synchronized with a protocol such as NTP, unless the clock is to purposely be stepped, instead of slewed to a new value.

In order to evaluate how well the kernel modifications work, it is useful to compare the operation over a typical day in the life of a DECstation 5000/240 both with and without the kernel PLL and microtime() routines. Figure 8 shows the cumulative probability distribution of time offsets between a primary server on the same Ethernet segment and the local clock. These curves show the probability that a randomly selected sample offset exceeds a particular value. The upper curve shows the clock behavior without the kernel modifications; the maximum sample offset is 7.95 ms in this case. The lower curve shows the clock behavior with the modifications; the maximum sample offset is 0.933 ms in this case. Clearly, the modifications do significantly improve timekeeping performance.

4.3. Daemon and Application Interfaces

Most Unix programs read the local clock using the gettimeofday() system call, which returns only the system time and timezone data. For some applications it is useful to know the maximum error of the reported time due to all causes, including clock reading errors, oscillator frequency errors and accumulated latencies on the path to a primary reference source. However, the new software can adjust the local clock to compensate for its intrinsic frequency error, so that the timing errors

expected in normal operation will usually be much less than the maximum errors. The user application interface includes a new system call ntp_gettime(), which returns the system time, as well as the maximum error and estimated error. This interface is intended to support applications that need such things, including distributed file systems, multimedia teleconferencing and other real-time applications. The protocol daemon application interface includes a new system call ntp_adjtime(), which can be used to read and write kernel variables used for precision timekeeping, including time and frequency adjustments, PLL time constant, leap-second warning and related data.

It does not seem generally useful in the user application interface to provide additional details private to the kernel and synchronization protocol, such as stratum, reference identifier, reference timestamp and so forth. It would in principle be possible for the application to independently evaluate the quality of time and project into the future how long this time might be "valid." However, to do that properly would duplicate the functionality of the synchronization protocol and require knowledge of many mundane details of the platform architecture, such as the subnet configuration, reachability status and related variables. However, for the curious, the ntp_adjtime() system call can be used to reveal some of these mysteries.

While the NTP Version 3 specification and daemon *xntpd* are used as an example application of the new system calls for use by a protocol daemon, the new system calls can be used by other protocols and daemon implementations as well. Even in cases where the local time is maintained by periodic exchanges of messages at relatively long intervals, such as using the NIST Automated Computer Time Service [NIS90], the ability to precisely adjust the local clock frequency simplifies the synchronization procedures and allows the call frequency to be considerably reduced.

4.4. Leap Seconds

There seems to be universal agreement in the standards community that some provision be made for leap seconds, but considerable disagreement on how they should be implemented. Leap seconds are inserted in the International Atomic Time (TAI) timescale in order that Universal Coordinated Time (UTC) agrees with conventional civil time, which is based on the rotation of the Earth. Corrections in the form of leap seconds are introduced as necessary at the end of the last day of June or December by international agreement. To date, there have been 19 occasions when seconds have been inserted, but none in which they have been deleted. Table 1 lists those occasions prior to the data of this report that leap seconds have been inserted in the UTC timescale along with the Modified Julian Day (MJD) number and NTP time of occurrence.

At issue is the manner in which the local clock responds to instances of leap-second insertion. According to international agreement, a leap-second insertion is implemented by inserting one second immediately following second 23:59:59 of the last minute of the day of insertion and becomes second 60 of that minute. Ideally, all synchronized clocks would behave in synchrony with that model; however, things are not quite that simple. The issues have to do with how intervals containing leap seconds are calculated. This may not be interesting when the intervals are relatively long, like years, but may be very important when the intervals are short, like a few seconds. They become even more interesting when the interval of interest lies wholly within a leap second, even though on a probabilistic basis such occurrences would be quite rare.

According to contemporary philosophy in the standards community, There are two approaches to these issues. In one the local clock runs at a constant rate (isochronous with TAI) and a special

UTC Date	MJD	NTP Time	Offset
01 Jan 72	41,317	2,272,060,800	10
30 Jun 72	41,498	2,287,785,600	11
31 Dec 72	41,682	2,303,683,200	12
31 Dec 73	42,047	2,335,219,200	13
31 Dec 74	42,412	2,366,755,200	14
31 Dec 75	42,777	2,398,291,200	15
31 Dec 76	43,143	2,429,913,600	16
31 Dec 77	43,508	2,461,449,600	17
31 Dec 78	43,873	2,492,985,600	18
31 Dec 79	44,238	2,524,521,600	19
30 Jun 81	44,785	2,571,782,400	20
30 Jun 82	45,150	2,603,318,400	21
30 Jun 83	45,515	2,634,854,400	22
30 Jun 85	46,246	2,698,012,800	23
31 Dec 87	47,160	2,776,982,400	24
31 Dec 89	47,891	2,840,140,800	25
31 Dec 90	48,256	2,871,590,400	26
31 Dec 91	48,621	2,903,212,800	27
30 June 93	49,168	2,950,473,600	28

Table 1. Table of Leap-Second Insertions

leap-second table is consulted when converting between kernel time and time provided to system and user processes. The table consists of the epoches of insertion with respect to TAI, with the initial insertion of 10 s at the initiation of UTC on 0^h 1 January 1972. When contriving a UTC timestamp, the kernel adds the offset computed from the table. Note that this model either requires the kernel to calculate all intervals spanning an epoch or requires system and user processes to understand these arcane details and access the table themselves. This raises atomicity issues where the kernel tables can be changed without notice to the user program.

In the alternative approach, the local clock also runs at a constant rate; however, the calculation of intervals assumes the local clock runs as UTC, so that no table is required. In this model the interval since the initiation of UTC, 1 January 1972, or since the Unix base epoch, 1 January 1970 for that matter, is assumed fixed as if no intervening leap seconds have occurred. This is the model adopted by virtually every existing Unix system. While this is simple and usually completely adequate, a problem exists just before, during and after the epoch of insertion, since it requires that the indicated time in effect stands still during the leap second itself.

In both approaches, a mechanism is required to find out in advance when the next epoch of insertion is to occur. In principle this is possible using any of several national means of time dissemination, including most U.S. time-dissemination services, although few radio clocks now include provision to decode this information. A user application may need to know whether a leap second is scheduled, since this might affect interval calculations spanning the event.

The NTP protocol design provides for the leap-warning information to be entered at the primary servers. A leap-warning condition is determined by the synchronization protocol (if remotely synchronized), by the radio clock (if implemented), or by the operator (if awake). This condition is set by the protocol daemon on the day the leap second is to occur (30 June or 31 December, as announced) by specifying in a ntp_adjtime() system call a clock status of either TIME_DEL, if a second is to be deleted, or TIME_INS, if a second is to be inserted. Note that, on all occasions since the inception of the leap-second scheme, there has never been a deletion occasion. If the value is TIME_DEL, the modified kernel adds one second to the system time immediately following second 23:59:58 and resets the clock status to TIME_OK. If the value is TIME_INS, the kernel subtracts one second from the system time immediately following second 23:59:59 and resets the clock status to TIME_OOP, in effect causing system time to repeat second 59. Immediately following the repeated second, the kernel resets the clock status to TIME_OK.

Depending upon the system-call implementation, the reported time during a leap second may repeat (with the TIME_OOP return code set to advertise that fact) or be monotonically adjusted until system time "catches up" to reported time. With the latter scheme the reported time will be correct before and shortly after the leap second (depending on the number of microtime() calls during the leap second itself), but freeze or slowly advance during the leap second. However, Most programs will probably use the ctime() library routine to convert from timeval (seconds, microseconds) format to tm format (seconds, minutes, ...). If this routine is modified to use the ntp_gettime() system call and inspect the return code, it could simply report the leap second as second 60.

To determine local midnight without fuss, the kernel simply finds the residue of the time.tv_sec value mod 86,400, but this requires a messy divide, which along with multiplies, is generally shunned be kernel implementers. Probably a better way to do this is to initialize an auxiliary counter in the settimeofday() routine using an ugly divide and increment the counter at the same time the time.tv_sec is incremented in the timer interrupt routine.

5. Timekeeping Errors in Time and Frequency

In preceding sections a number of improvements in driver software and hardware are described, along with modifications to the SunOS, Ultrix and OSF/1 Unix kernels. These provide increased accuracy and stability of the local clock without requiring an externally disciplined, precision oscillator. However, there is only so much improvement possible when the clock oscillator is not specifically engineered for good stability. The basic question to answer is: can the residual sources of error be systematically controlled so that the dominant factor remaining is the stability of the oscillator itself? The software and hardware previously described are designed to do just that. The following sections discuss these issues in detail and provide an assessment of the degree to which this goal has been achieved.

At the application level the timekeeping accuracy is limited by several causes, including the error in synchronizing the local clock to a national standard and the error in reading the clock by the application. In the case of a primary server connected directly to a radio clock, the accuracy of the radio time is limited by the propagation medium, its receiver clock oscillator stability and the time since it last synchronized to the broadcast signal. The accuracy at the synchronization daemon is limited by the receiver accuracy and the jitter accumulated in the input/output interface and operating system queues between the hardware and the daemon. The accuracy at the application level is determined by the daemon accuracy, the stability of the local clock oscillator and the time since it was last synchronized by the daemon.

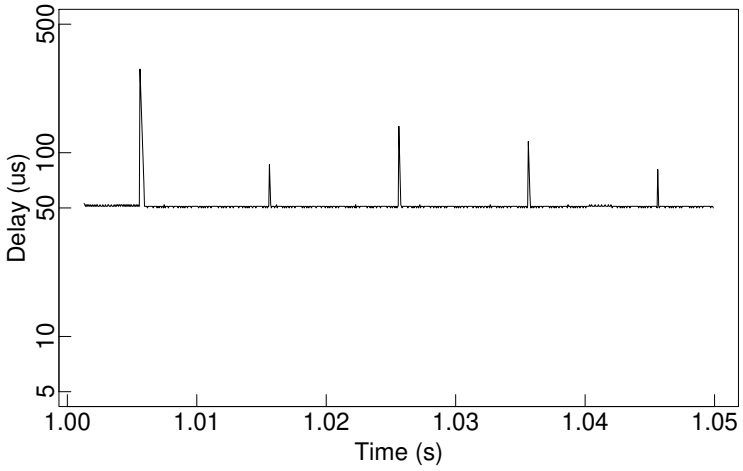

Figure 9. Kernel Latency for SPARCstation IPC - 1

There are two major components of error in the local clock itself, the timing accuracy, or precision, with which it can be set and read and the frequency accuracy, or stability, it can maintain after being set. A goal in the following discussion is to calibrate the overall precision and stability and to reveal the inherent limitations in timekeeping performance of typical computers on the market today. However, a tricky problem in precision timekeeping is sloppy system design in hardware and software components. For example, some systems rely on programmed input/output, rather than interrupts, when displaying kernel error messages. This can result in missed timer interrupts and timing glitches popularly known as *timewarps*. For instance, Unix workstations must be operated with windowing systems to avoid use of resident firmware-coded input/output routines, which do not use interrupts. All of the experiments described herein have been carefully scrutinized to insure that these particular hazards have been avoided.

5.1. Clock Reading Errors

In this section the primary emphasis is on the measurement and characterization of the errors in reading the local clock by an application. This component can be represented by the precision of the local clock and latency of the hardware and operating system to deliver a sample of the local clock, or *timestamp* to the application. The latency consists of three parts: a random quantity due to the clock precision, a fixed delay to cycle through the kernel code, and a variable scheduling delay to resume application processing upon return from the kernel.

In most Unix systems presently marketed the resolution of the local clock is determined by the hardware timer interrupt interval, which is generally in the 1-10 millisecond (ms) range and corresponds to an interrupt rate in the 100-1000 Hz range. For instance, the timer interrupt rate is 100 Hz for the SunOS system, 256 Hz for the Ultrix system and 1024 Hz for the OSF/1 system discussed earlier in this report. However, while all three systems considered here have hardware facilities to resolve time to the microsecond, only the stock SunOS kernel uses them. For the purposes of the measurements reported here, the Ultrix and OSF/1 kernels were modified to utilize these features and can indeed resolve time to the microsecond. The way in which these improvements were accomplished is discussed in a previous section.

In order to calibrate the error in reading the local clock from an application, the delay to cycle through the kernel and retrieve a timestamp using the gettimeofday() system call was measured on

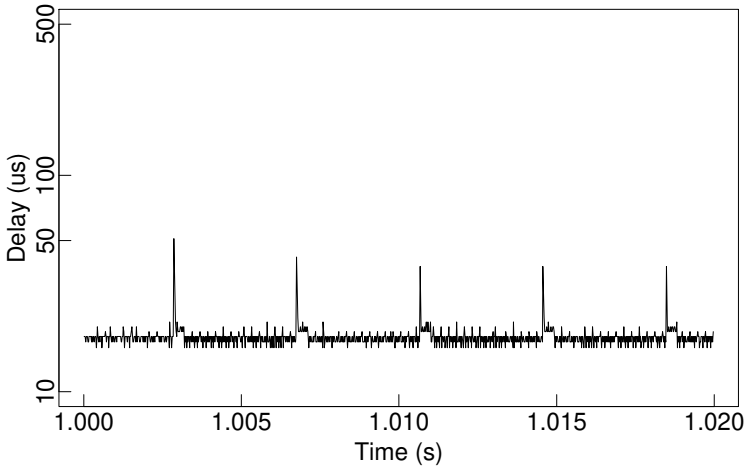

Figure 10. Kernel Latency for DECstation 5000/240

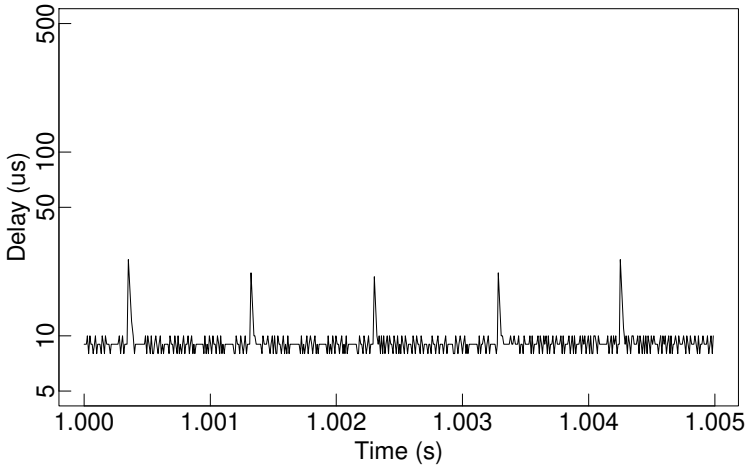

Figure 11. Kernel Latency for DEC 3000/400 AXP

each of the three Unix workstations: a SPARCstation 4/65 (IPC) SPARC processor running the SunOS 4.1.1 operating system, a Digital Equipment DECstation 5000/240 MIPS 3000 processor running Ultrix 4.2a and a Digital Equipment 3000/400 AXP Alpha 21064 processor running OSF/1. For the purposes of these measurements, the workstations were performing no tasks other than routine system maintenance and the application task making the measurements.

The experiment involves first touching up to 250,000 words (64 bits in the OSF/1 kernel, 32 bits in the others) of a main-memory array in turn. Since for this experiment the workstations were otherwise idle, this insures that the virtual memory pages are in main memory and that old data are flushed from the various caches and lookaside buffers. Following this, up to 250,000 calls on gettimeofday() are made and the timestamps returned are saved in the array for later processing. Finally, the array is saved in a file for later processing and display.

Figures 9, 10 and 11 show the latencies of the gettimeofday() system call on the SunOS, Ultrix and OSF/1 kernels, respectively. The figures are all basically similar and reflect the architecture of the processor and memory system and kernel code paths. Characteristic of these figures is a constant baseline, representing the minimum latency in microseconds of the code path through the kernel,

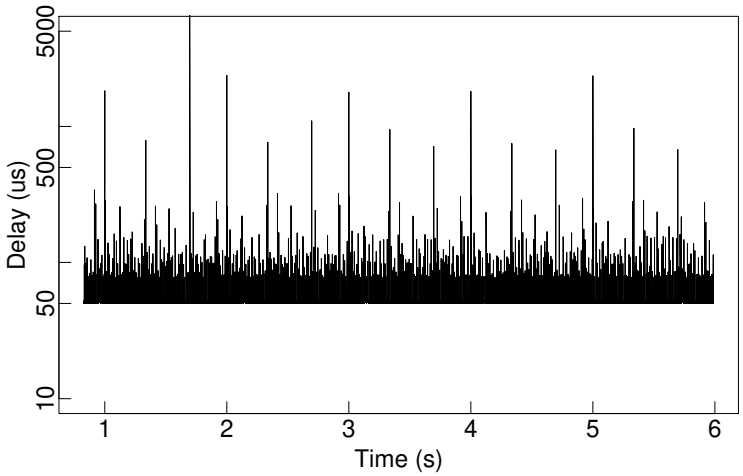

Figure 12. Kernel Latency for SPARCstation IPC - 2

interrupted at intervals with spikes up to several times the baseline. These spikes are due to the timer-interrupt routine which is called at each tick of the local clock and reflect the time to update the clock itself and perform certain scheduling and statistics tasks. As aside, there is an interesting feature of the Ultrix kernel which is manifested by a hump just following the timer interrupt. There is as yet no definitive explanation for this, other than a suspicion it may be due to cache or lookaside buffer turmoil.

The apparent regularity evident in these figures is belied by Figure 12, which shows the latencies of the SPARCstation IPC over a much longer interval than in the previous figures. The fine structure evident in this figure is due to the characteristics shown in Figure 9, but the much larger excursions up to a millisecond or more are most likely due to system daemon housekeeping functions. The figure suggests a basic heartbeat of three beats per second with somewhat longer latencies beating on the second. Obviously, the time intervals of the previous figures were selected to avoid these beats. The on-second beats are in part due to the NTP Version 3 daemon, which is activated once per second for housekeeping purposes. The cause(s) of the other beats are undetermined, but very likely due to housekeeping functions on the part of other system daemons.

Note that the characteristics shown in these figures are specific to an application process running at a relatively low system priority. It would ordinarily be expected that real-time processes are assigned a higher priority, so that latencies could be controlled with respect to other application and daemon processes. In fact, this is the case with the NTP daemon. In this way at least some the spikes evident in Figure 12 could probably be avoided. Nevertheless, the measurements reported previously in this report reveal delay excursions over 25 ms on rare occasions, even for the NTP daemon.

However, in many real-time applications it is more important to assign a precision timestamp to an event than it is to launch it at an exact prearranged epoch. In fact, in all three workstations considered here, internally timed events can be launched only as the result of a timer interrupt, and this limits the timing precision to that of the interval timer itself, which is in the range 1-10 ms. However, when it is necessary to derive a precision timestamp before launching an event, a simple trick can suffice to insure a precision approaching the minimum latency shown in the above figures. The algorithm consist of calling gettimeofday() repeatedly until the interval between two calls is less than a prescribed value. There is of course a tradeoff between the precision achievable in this way

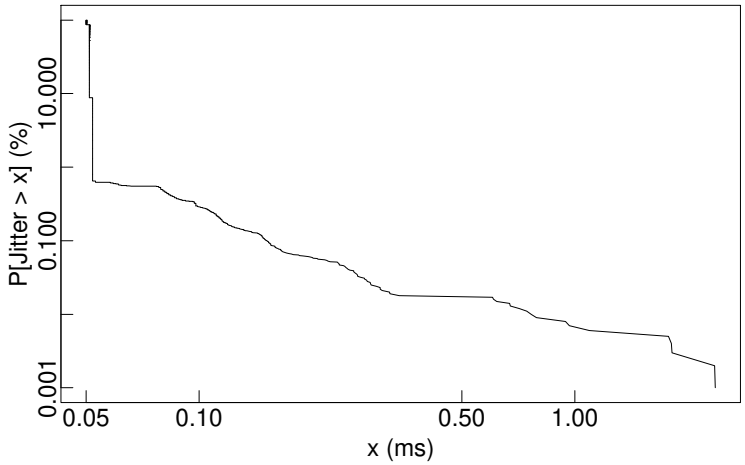

Figure 13. Probability of Error for SPARCstation IPC

and the overhead of repeated calls, since the smaller values will cause the calls to be repeated more times.

Figure 13 illustrates the results of this technique using the SPARCstation IPC. The graph shows the cumulative probability distribution for the workstation over one full day, from which a conclusion can be drawn that the probability of exceeding even a threshold as low as about 60 µs is about 0.5 percent, or about the probability of colliding with a timer interrupt on a random request. Note that the probability of an exceeding this value is roughly a straight line on log-log coordinates and that only a few of some 100,000 samples show latencies greater than 1 ms.

5.2. Clock Frequency Stability

The basic model for synchronizing computer network clocks is a set of disciplined oscillators, where each clock is periodically adjusted in time to agree with one or more peers. In the NTP local clock model, the discipline process is implemented as a type-II phase-lock loop. In principle, this is identical to the method used in many standard frequency sources, such as cesium oscillators and radio clocks. However, unless the oscillator is stable and has a low intrinsic frequency error with respect to an external standard, it will exhibit timing errors depending on the interval since last adjusted within nominal time tolerance. This section describes certain experiments designed to learn the oscillator stability typical of modern workstation hardware and software systems.

One of the problems in measurements of this type is a stable reference with which to compare the local clock against, as well as a low-jitter interface in the form of a PPS signal generated by a cesium clock or precision timing receiver. Ordinary radio clocks, such as those synchronized by WWVB, WWV, GOES or telephone modem, are generally unacceptable, since the short-term jitter and wander are generally much inferior to the precision sources. For the following experiments, the PPS signal from a GPS timing receiver was used as the disciplining source and all hardware and software improvements and kernel modifications described previously were in place. For instance, the residual latency to capture a timestamp from a PPS transition has been measured at less than 10 µs on a similarly equipped SPARCstation IPC [Craig Leres - personal communication].

Frequency errors can be classified by the time intervals over which they are measured. With respect to computer network clocks, such as those in this report, three components of frequency errors can be identified: noise, with timing intervals less than a minute or so, short-term stability (wander),

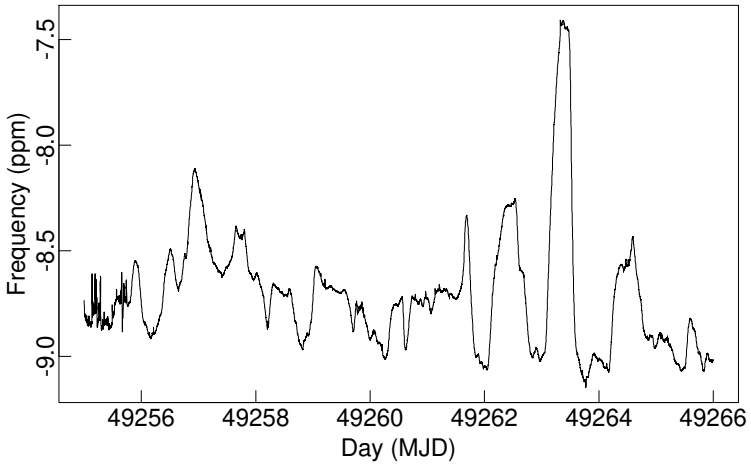

Figure 14. Typical Clock Oscillator Wander

with intervals from a minute to an hour or so, and long-term stability (mean frequency error), with intervals greater than an hour. The noise component depends on such things as power supply regulation and vibration, but is not ordinarily a problem. The wander component depends primarily on the ambient temperature and is the major source of timing errors in the quartz oscillators used in modern computers. In a type-II PLL the mean frequency error is minimized by the discipline imposed by the PLL and is normally not significant, except for an initial transient while the intrinsic frequency offset of the local clock oscillator is being learned.

Since the major contribution to frequency error is due to temperature fluctuations, it would make sense to stabilize the operating temperature of the circuitry. As will be demonstrated later, while the oscillator stability of modern workstations is typically within a couple of ppm in normal office environments, stabilities one or two orders of magnitude better than this are necessary to reliably reduce incidental timing errors to the order of a few tens of microseconds. However, a good temperature-compensated quartz oscillator can be a relatively expensive component not likely to be found in cost-competitive workstations. Therefore, it is assumed in this report that the time synchronization system must accept what is available, and this is what the following experiments are designed to evaluate.

For an example of the frequency wander in a typical workstation, consider Figure 14, which shows the frequency of the clock oscillator over a ten-day period for a DECstation 5000/240. The figure shows variations over a 2.5 ppm range, with marked diurnal variations on MJD days 49262 and 49263. These happened to be pleasant Fall days when the laboratory windows were open and the ambient temperature followed the climate. Nevertheless, the conclusion drawn from this figure is that frequency variations up to a couple of ppm must be expected as the norm for typical modern workstations.

5.2.1. Allan Variance of Typical Workstations

The stability of a free-running frequency source is commonly characterized by a statistic called *Allan variance* [ALL74], which is defined as follows. Consider a series of time offsets measured between a local clock oscillator and some external standard. Let $\theta(i)$ be the ith measurement and T be the interval between measurements. Define the *fractional frequency* $y(i) \equiv \frac{\theta(i) - \theta(i-1)}{T}$.

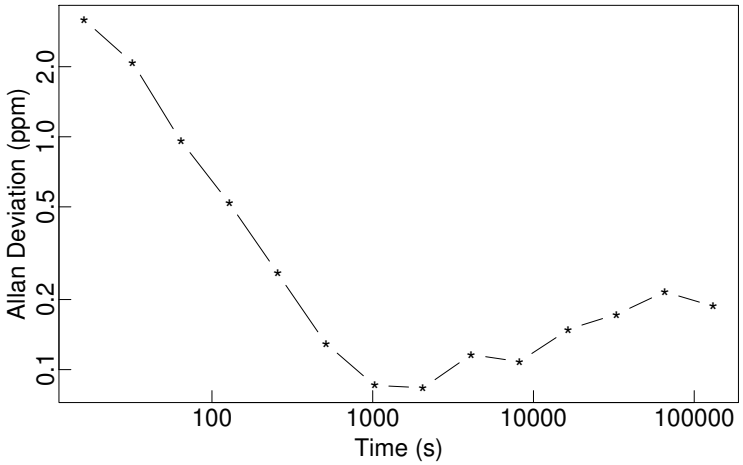

Figure 15. Allan Variance of Typical Local Oscillator

Consider a sequence of N independent fractional frequency samples $y(j)$ ($j = 1, 2, \ldots, N$). Let τ be the nominal integration interval over which these samples are averaged. Using the notation <> to indicate the average, the Allan variance is defined

$$<\sigma_y^2(N, T, \tau)> \equiv < \frac{1}{N-1} \left[\sum_{j=1}^{N} y(j)^2 - \frac{1}{N} \left(\sum_{j=1}^{N} y(j) \right)^2 \right] > ,$$

A particularly useful formulation is $N = 2$ and $T = \tau$, called the 2-sample Allan variance:

$$<\sigma_y^2(N = 2, T = \tau, \tau)> \equiv \sigma_y^2(\tau) = < \frac{[y(j+1) - y(j)]^2}{2} > .$$

The average over a sequence of n sample sequences as above is then

$$\sigma_y^2(\tau) = \frac{1}{2(n-1)} \sum_{j=1}^{n-1} [y(j+1) - y(j)]^2 .$$

The Allan variance $\sigma_y^2(\tau)$ or Allan deviation $\sigma_y(\tau)$ are particularly useful when comparing the intrinsic stability of the local clock oscillator used in typical workstations, as it can be used to refine the PLL time constants and update intervals. Figure 15 shows the results of an experiment designed to determine the Allan deviation of a DECstation 5000/240 under typical room-temperature conditions. For the experiment the oscillator was first synchronized to a primary server on the same LAN using NTP to allow the frequency to stabilize, then uncoupled from NTP and allowed to free-run for about seven days. The local clock offsets during this interval were measured using NTP and the same server, which revealed a mean frequency error less than 1 ppm. This model is designed to closely duplicate actual operating conditions, including the jitter of the LAN and operating systems involved.

It is important to note that both the x and y scales of Figure 15 are logarithmic. The characteristic falls rapidly from low values of τ until settling to about 0.5 ppm at high values. The conclusion to be drawn is that increasing the integration interval above about $\tau = 2000$ s does not substantially

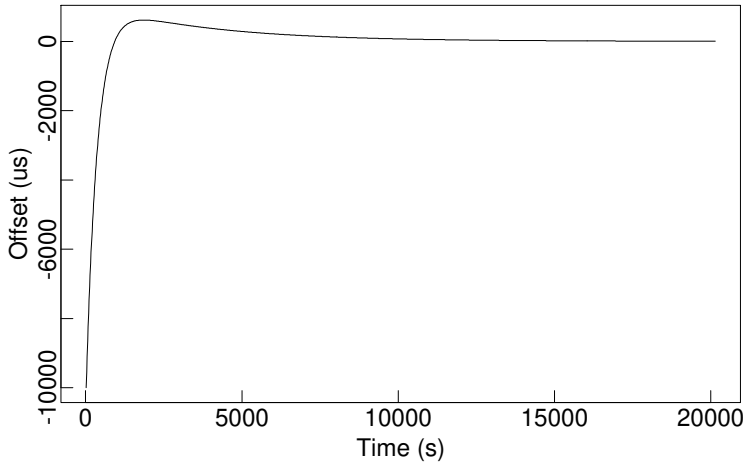

Figure 16. Transient Response of NTP PLL - Phase

improve the oscillator stability; however, increasing τ may be desirable for other reasons, as demonstrated in the next section. On the other hand, decreasing it much below about $\tau = 1000$ s can result in degraded stability. At $\tau = 16$ s, the lowest value used in the current NTP Version 3 implementation, the stability has already degraded by a factor of 37.

5.2.2. PLL Time Constant and Update Interval

In this report the symbol τ is used both to designate the integration interval of the Allan deviation and as the PLL time constant; however, the particular meaning in each case should be clear from context. A crucial determinant of stability is the time constant τ of the PLL, which in the NTP local clock model is dynamically determined as a function of dispersions measured between the local clock and its disciplining sources. If τ is too small, the oscillator frequency can deviate excessively due to noise transients and can result in large timing errors when contact with the disciplining sources is lost over extended periods. If τ is too large, the PLL can fail to track the oscillator wander due to ambient temperature variations.

A rule of thumb in the design of PLLs is that, in order to assure stability, the PLL loop delay must not be much more than 0.1τ. With the default PLL parameters specified in [MIL92c], the PLL has a time constant of about 1000 s. In order to assure stability, the loop delay due to the clock filter and update interval μ must not be more than about 100 s, which requires $\mu \le 16$ s. The calculated time response for the NTP PLL to a 10-ms step transient at time zero is shown in Figure 16. This figure was produced by the kernel PLL simulator described previously and in Appendix C. Note that the error reaches zero at about 900 s and that the overshoot is about 7 percent. The NTP PLL is designed so that the overshoot characteristics remain constant over the entire range $1 \le \tau \le 64$ s, which corresponds to $16 \le \mu \le 1024$ s.

While a type-II PLL can in principle eliminate residual timing errors due to a constant frequency offset, the PLL is quite sensitive to changes in frequency, such as might occur due to room temperature variations. A good working rule of thumb is that an uncompensated quartz oscillator varies about one ppm per degree Celsius. Room temperature variations in the order of one or two decrees can be expected in office and home environments. Figure 17 shows the timing errors induced by a 2-ppm step change in frequency. This figure was produced by the kernel PLL simulator described previously and in Appendix C. The error reaches a peak of 600 μs, which is large in

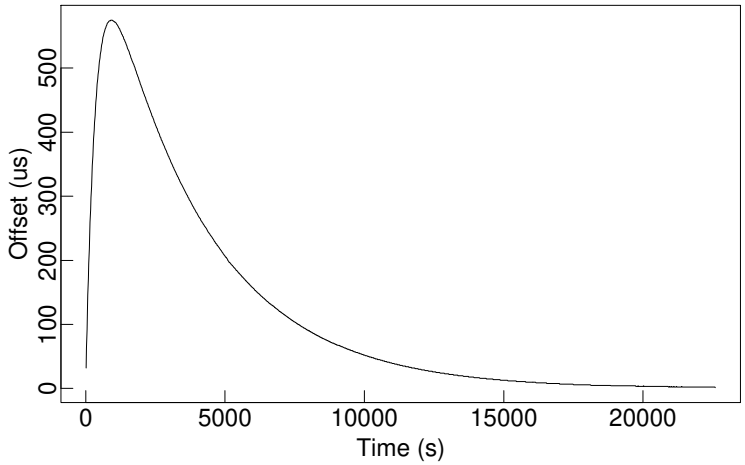

Figure 17. Transient Response of NTP PLL - Frequency

comparison with other sources of error considered in this report. The amplitude of this characteristic scales directly with the temperature change. Obviously, the timing error due to this cause dominates the error budget, not only for primary servers directly synchronized to precision PPS signals, but for secondary clients on a LAN as well.

The PLL characteristics shown in Figures 16 and 17 are calculated for the default time constant $\tau = 1$, which requires the update interval $\mu \le 16$ s. For subnet paths spanning a WAN, such frequent updates are impractical and much longer update intervals are appropriate. The design of the NTP PLL allows μ to be increased in direct proportion to τ while preserving the PLL characteristics. To do this, the optimum value of τ is determined on the basis of measured network delays and dispersions. For the longer network paths with higher delays and dispersions, this allows τ to be increased and with it μ. However, a large τ limits the PLL response to temperature-induced frequency changes. Analysis confirms the *x* and *y* axes of the characteristic shown in Figure 17 scale directly as τ, which means the timing errors will scale as well.

In principle, τ values near the upper limit of 64 could result in timing errors up to 64 times those shown in Figure 17, or 50 ms. For this reason τ is often clamped not to exceed 64 s or even 16 s for servers on a LAN. In spite of this hazard, and as shown in the next section of this report, even on WAN paths where τ operates near its upper limit, timing errors are usually less than a millisecond or two.

6. Timekeeping in the Global Internet

The preceding sections suggest that submillisecond timekeeping on a primary server connected directly to a precision source of time is possible most of the time, where the exceptions are almost always due either to system disruptions like reboot or such things as kernel error messages or large temperature surges. As a practical matter, it is useful to explore just how well the timekeeping function can be managed in an ordinary LAN workstation and in WAN paths of various kinds. In this section are presented the results from several experiments meant to calibrate the expectations in accuracy. As before, all hardware and software improvements and kernel modifications have been done on the LAN workstations, although this is not the case on systems outside the LAN.

It is important to note that, in all measurements reported in this section, timing offsets relative to the local clock are measured at the output of the clock filter on Figure 2, so include the smoothing

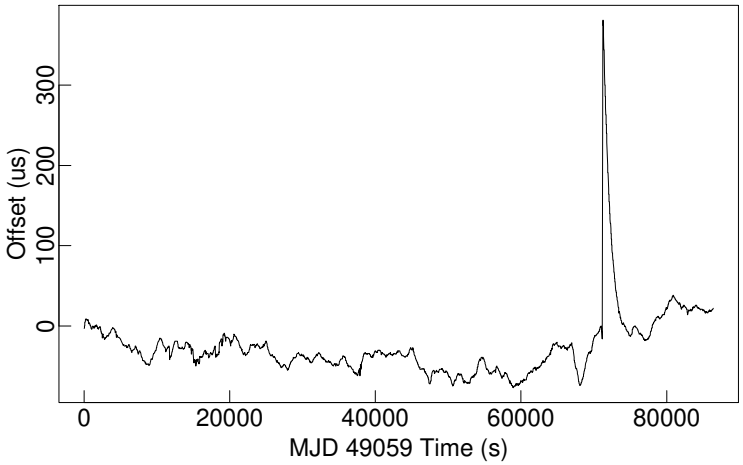

Figure 18. Timing Offsets of a Primary Server

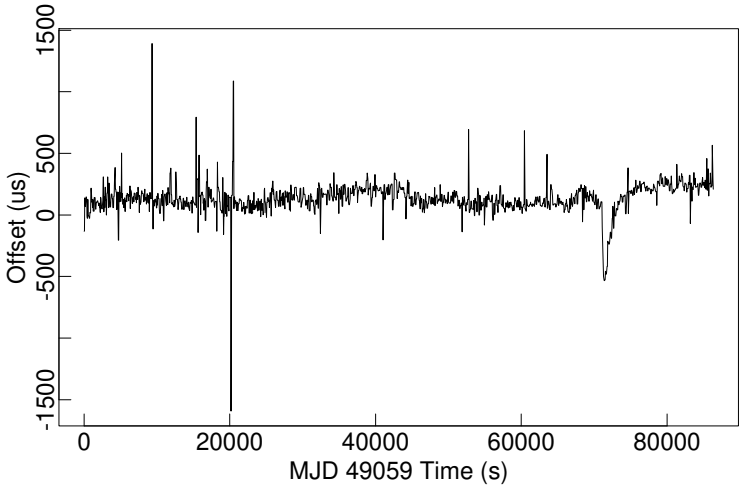

Figure 19. Timing Offsets Between Primary Servers

effect of that filter. However, the local clock itself is controlled by that output and others and processed further by the clock selection and combining algorithms before processing by the local clock algorithm. The local clock algorithm acts as a low-pass filter to suppress transients, so that solitary spikes shown in the data are almost always suppressed. Thus, while it is not possible to infer the exact local clock offset between two NTP time servers, it is certain that the actual offsets tend to the mean as shown in the figures.

6.1. Timekeeping in LANs and WANs

Figure 18 shows the timekeeping behavior of a primary server synchronized to the PPS signal of a GPS receiver over one full day. The data from which this figure was generated consist of measured offsets between the PPS signal and the local clock, where the measurements were taken every 16 s. This particular machine is a dedicated, primary server with both GPS and WWVB receivers and supporting over 250 clients, some of which use the computationally intense cryptographic authentication procedures outlined in the NTP Version 3 specification RFC-1305. Both noise and wander components are apparent from the figure, as well as a 400-μs glitch that may be due to something as arcane as a daemon restart or radio glitch, for example. The behavior shown in Figure 18 should

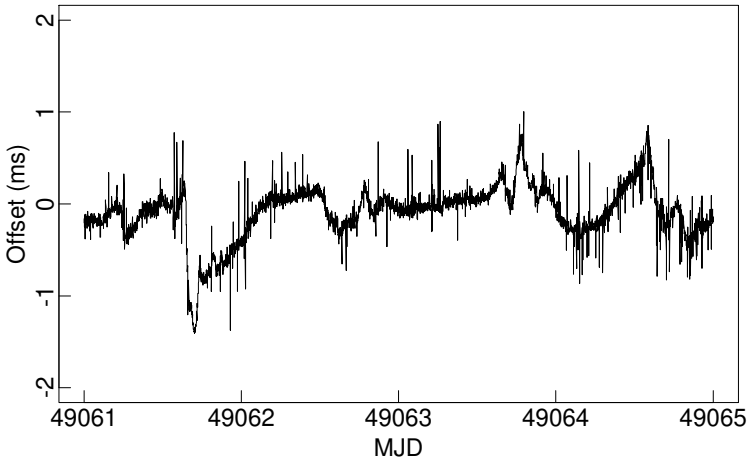

Figure 20. Timing Offsets of a LAN Secondary Server

be contrasted with the behavior shown in Figure 7, which is for a similarly configured primary server restricted to a lessor number of private clients. These data suggest a conclusion that, even with over 250 clients and two radio clocks, the local clock can be stabilized to well within the millisecond.

Figure 19 shows the timing offsets between two primary servers, each synchronized to the same PPS signal and connected by a moderately loaded Ethernet. One of these servers is the dedicated primary server mentioned above, while the other is both a primary time server and a file server for a network of about two dozen client workstations, so the experiment is typical of working systems. The jitter apparent in the figure is due to queueing delays, Ethernet collisions and all the ordinary timing noise expected in a working environment. There are occasional spikes of 1 ms or more due to various causes, but only a couple greater than that, which should be suppressed by the local clock algorithm. Note the 400-μs spike near second 72,000, which matches the spike of Figure 18 taken on the same day, and the occurrence of an apparent systematic offset of 50-100 μs, which is believed due to the difference in cable lengths connecting each server to the GPS receiver.

Figure 20 shows the timing offsets between a primary server and a secondary (stratum 2) client on the same LAN over five full days. The PLL time constant $\tau = 4$ and the update interval $\mu = 64$ s. Clearly, the wander due to ambient temperature variations has increased; there is a hint of diurnal variation as well. This particular machine is located in a room with a window air conditioner, so is subject to relatively large and sudden temperature changes. Similar graphs were obtained using several LAN workstations of various manufacture and comparable speeds using both Ethernet and FDDI transmission media.

Figure 21 shows the result of a similar experiment between a primary server and secondary client over a 1.544 Mbps circuit and several routers. The primary server is one of the DCnet public servers mentioned previously, while the secondary client is an IBM RS6000 which is a component of the NSFnet backbone node at College Park, MD. There are three routers on two Ethernets, the T1 circuit and a token ring on the path between the two machines, but the T1 circuit is loaded to only a fraction of its capacity. Again, note that, while the jitter evident in the figure ranges over ±2 ms, the PLL at the client is effectively a low-pass filter and removes much of the apparent jitter. It would not be adventurous to suggest the actual discrepancies between the two clocks are not much worse than shown in Figure 20.

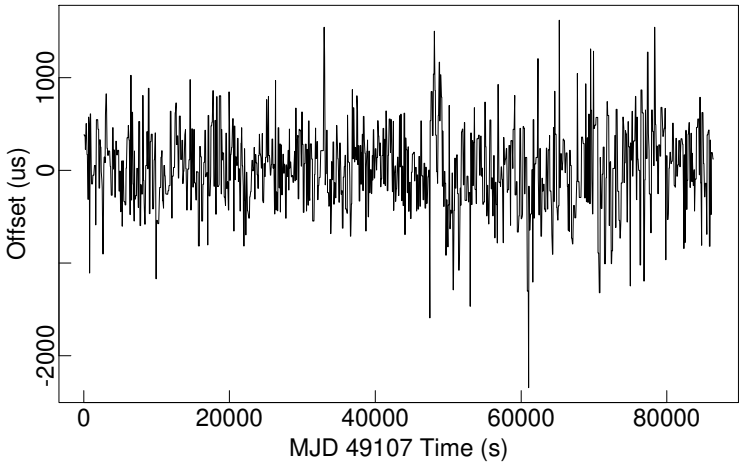

Figure 21. Timing Offsets of a NSFnet Secondary Server

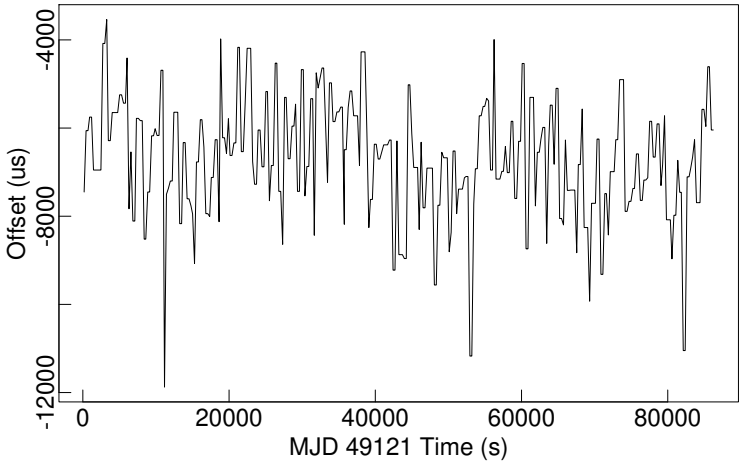

Figure 22. Timing Offsets of a NIST Primary Server

However, timekeeping accuracy using much longer paths spanning the globe can be uneven. Figure 22 illustrates a path between a DCnet primary server and a client at the National Institute of Science and Technology (NIST) at Boulder, CO, which is directly synchronized to the U.S. national standard cesium clock ensemble. Note the apparent bias of about -5.5 ms, which is due to the somewhat roundabout network paths on the outbound and return legs of the network path. The outbound path enters the NIST agency network at Gaithersburg, MD, while the return path enters the NSFnet backbone at National Center for Atmospheric Research (NCAR) at Boulder, CO. These two legs have different transmission delays, undoubtedly due to different network speeds. This illustrates the not surprising policy on the part of each network administration to offload Internet traffic from the local system at the first opportunity.

Finally, in a search to determine from among about 100 NTP primary time servers the one that was (a) independently synchronized directly to national standard time and (b) as far away as possible in the globe from the DCnet machines, a primary server in Sydney, Australia, was found. This is a truly heroic test, since the transmission facilities are partially by satellite, partially by undersea cable and the intervening networks sometimes slow and seriously overloaded. Figure 23 shows an

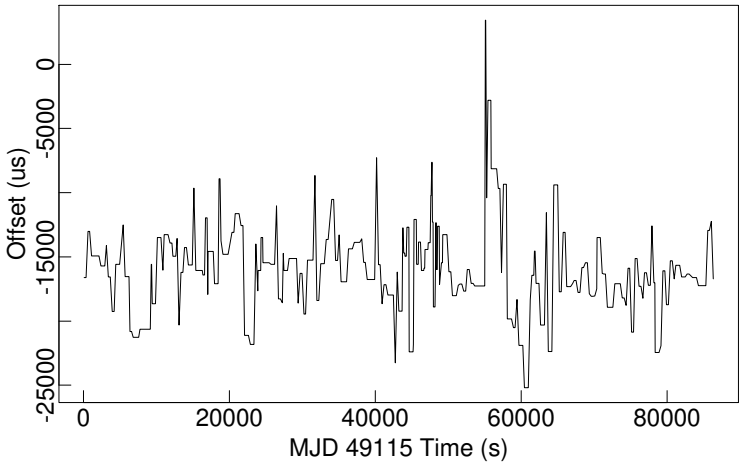

Figure 23. Timing Offsets for an Australian Primary Server

apparent -15-ms bias due to differential delays on the outbound and return legs, as well as spikes up to a few milliseconds due to ordinary network queueing, plus a few larger and longer spikes probably due to a circuit outage and network reroute.

6.2. NTP System Performance

The above experiments have used data collected over only one day or a few days. In order to gain some insight in the behavior over longer periods, a number of experiments were conducted over periods spanning up to several months. Table 2 shows a selection of client-server paths typical of the DCnet primary servers described in Appendix D. During the lifetime of these experiments various software and hardware were in repair, machines were rebooted, the network was reconfigured and different versions of the NTP protocol daemon and Unix kernel were in development. In the table, the first column identifies the peer or radio clock and the second the number of days in which data were collected (data were not collected on days when the local clock offset of the monitoring machine was greater than 1 ms relative to its radio clock). The next two columns give the mean and RMS error over all days of collection, while the next gives the maximum absolute error relative to the mean on the day of collection. Finally, the last four columns give the number of days on which the maximum absolute error exceeds 1 ms, 5 ms, 10 ms and 50 ms, respectively.

The results of these experiments are a mixed bag. The performance of the GPS receiver was excellent, as expected, but that of the WWVB receiver is disappointing. There is some evidence that Spectracom WWVB receivers elsewhere in the country do not experience relatively large errors of this kind; however, the performance of both Spectracom WWVB receivers, one on the DCnet campus subnet, the other on the backroom subnet, show the same problems. The performance of the clients on the DCnet Ethernet (subnet 1) and FDDI ring (subnet 5) confirm the claim that they can maintain submillisecond accuracy relative to the server, but all of them show errors greater than a millisecond on at least some occasions. This is a strong claim, since all it takes is one delay spike over 1 ms and the day is marked accordingly.

The performance of the global servers was somewhat better than expected. These primary servers are near Washington, D.C. (umd1.umd.edu), San Diego (fuzz.sdsc.edu), NIST Boulder (time_a.timefreq.bldrdoc.gov), Norway (fuzzy.nta.no), Switzerland (swisstime.ethz.ch), Germany (ntps1-0.uni.erlangen.de and Australia (swifty.dap.csiro.au). All of these servers are independently

NTP Server	Days	Mean	RMS Error	Max Error	>1	>5	>10	>50
Radio Clocks								
Spectracom WWVB	71	-0.974	2.179	57.600	18	4	1	1
Austron GPS	91	0.000	0.012	1.000	0	0	0	0
DCnet Servers								
rackety.udel.edu	95	-0.066	0.053	2.054	11	0	0	0
mizbeaver.udel.edu	17	-0.150	0.171	1.141	2	0	0	0
churchy.udel.edu	42	-0.185	0.227	3.150	15	0	0	0
pogo.udel.edu	88	0.091	0.057	1.588	8	0	0	0
beauregard.udel.edu	187	0.016	0.108	2.688	30	0	0	0
pogo-fddi.udel.edu	113	0.001	0.059	1.643	1	0	0	0
cowbird.udel.edu	63	-0.098	0.238	2.071	13	0	0	0
Global Servers								
umd1.umd.edu	78	-4.266	2.669	35.893	29	29	28	0
fuzzy.nta.no	22	0.015	5.328	70.470	2	2	2	1
swisstime.ethz.ch	37	3.102	4.533	97.291	14	14	13	4
swifty.dap.csiro.au	84	2.364	56.700	3944.471	27	27	27	13
ntps1-0.uni.erlangen.de	70	0.810	10.861	490.931	12	12	12	6
time_a.timefreq.bldrdoc.gov	85	-1.511	1.686	80.567	28	19	11	2
fuzz.sdsc.edu	77	-3.889	2.632	47.597	27	27	23	0
DARTnet Routers								
la.dart.net	83	-0.650	0.771	17.849	28	8	3	0
lbl.dart.net	72	0.103	0.214	15.729	20	8	1	0
isi.dart.net	79	-0.819	0.740	8.564	21	9	0	0
NSFnet Routers								
enss136.t3.ans.net	88	-0.657	1.203	32.659	38	23	10	0
enss141.t3.ans.net	87	-6.285	1.846	20.174	37	29	15	0

Table 2. Characteristics of Typical NTP Peers

synchronized to a local source of standard time, either by a radio clock or cesium oscillator. Most of these servers are many Internet hops distant, where the networks involved are not particularly fast and are overloaded. For example, the Australian server is 20 Internet hops distant from the DCnet monitoring machine and the Switzerland server is 17 hops distant. The mean offsets shown are undoubtedly due to differential path delays; however, the rather large maxima are probably due to network congestion.

The data for the DARTnet routers suggest a claim of submillisecond accuracy on a network composed of 1.544 Mbps T1 circuits may be adventurous, since there were at least some days when the offsets exceeded 10 ms. However, some experiments involving DARTnet are designed to stress the network to extremes and likely lead to large variations in delay; it is likely that at least some data were recorded during these experiments and account for some of the spikes. Note that lbl.dart.net is independently synchronized to a GPS receiver and that there is only one path between

any two DARTnet routers, so the mean offset shown represents true measurement error and should be compared with the mean offsets shown for DCnet servers rackety.udel.edu and pogo.udel.edu, which are also synchronized directly to a GPS receiver.

Two of about two dozen NSFnet backbone routers are shown in the table. The College Park router enss136.t3.ans.net is the same one shown on a smaller interval in Figure 21. The path between this router and the monitoring machine is the main connecting link between the DCnet and other national and regional backbones. Since it carries one of the "multicast tunnels" involved in the MBONE multimedia conferencing network, it is subject to relatively heavy loads on occasion, which explains at least a few of the spikes evident in the data. This site and the other shown operate as secondary servers (stratum 2), with each server configured to use different primary servers. Since these primary servers are located in regional networks some number of hops distant from the NSFnet point of presence, there are differential path delays which account for the mean time offsets shown.

7. Summary and Conclusions

In the several years over which the NTP versions evolved, the accuracy, stability and reliability expectations have increasingly become more ambitious. As each new version was developed, a particular crop of error sources was found and remedial algorithms devised. This work led to the clock filter algorithm, intersection algorithm, clustering algorithm and combining algorithms of the NTP Version 3 specification and implementation. In parallel, the NTP local clock model was refined and tuned for best performance on existing Internet paths, some of which have outrageous delay variations due to gross overload. Previous experience has suggested that timekeeping accuracies in most portions of the Internet can be achieved to some tens of milliseconds.

This report discusses issues in precision time synchronization of computer network clocks. The primary emphasis in the discussion is on achieving accuracies better than a millisecond on a network with a primary server and a number of modern workstation clients. Networks with which this goal has been demonstrated include Ethernet, FDDI and light to moderately loaded 1.544-Mbps T1 circuits. As evident from measurements reported herein, accuracies in the order of 10 ms can be achieved on heroic paths of the global Internet, including paths to Australia and Europe. However, the eventual success of this goal may require prior knowledge of differential delays that are unfortunately common in some portions of the Internet.

Much of the discussion in this report is on methods to improve the accuracy of primary servers and their clients using engineered hardware and/or kernel software modifications. These include mechanisms to capture a precision timestamp from the PPS or IRIG signals generated by some timing receivers. It is apparent, however, that accumulated latencies over 8 ms accrue in some Unix kernels, unless means are taken to capture timestamps early in the code path between the interrupt and the synchronization daemon. Most of the latency burden can be avoided without kernel modifications, but some workstations will require additional hardware or kernel software to achieve submillisecond accuracy. The report describes hardware and Unix kernel software that can improve the kernel time resolution to a microsecond for the DEC Alpha and some models of the DECstation 5000.

The report describes a new local clock model for the Unix kernel in which the NTP local clock algorithm is moved from the daemon to the kernel. This is an implementation of a type-II phase-lock loop, which adjusts both the time and frequency in response to updates computed by NTP or another synchronization protocol. It allows dramatic reduction in sawtooth error and jitter, yet does not

require frequent calls on kernel routines, as does the unmodified NTP version 3 implementation. A thorough engineering analysis is given in the text and in two appendices of this report in the hope that this technology can find its way into future kernel implementations.

There is considerable discussion in this report on the results of a series of experiments intended to calibrate the residuals in time and frequency resulting from the improved hardware and kernel software. The results show that the minimum latencies to read the kernel clock range from about 9 μs for the DEC 3000/400 AXP Alpha to 50 μs for the SPARCstation IPC. The latency distribution has a knee at about three times the minimum as the result of timer interrupts and has a long tail up to some tens of milliseconds on the IPC, apparently due to daemon housekeeping functions. While the tail on the latency distribution is rather large, techniques are described which can avoid most of it at the cost of a slight increase in processing time.

The results show, as expected, that timekeeping accuracy depends on the network path between server and client; however, considering the complexity of some paths in the global Internet, the results are better than expected. The report demonstrates that the residual offsets due to differential network delays and other causes can be bounded on an absolute basis and on a statistical basis; However, there is no way other than using outside references to determine the asymmetries in practice. Measurements between servers independently synchronized to national time standards with expected accuracy better than 1 ms were used to calibrate the errors. This reveals the dominant contribution to the error budget is differential network delays. It is in principal possible to compensate for these delays using information broadcast from designated time servers, for example.

One striking fact emerging from the experiment program is the observation that the limiting factor to further accuracy improvements is the stability of the local clock, which is usually implemented by an uncompensated quartz oscillator. The stability of such oscillators varies in the order of 1 ppm per degree Celsius. With normal room-temperature variations, the timing error can reach a large fraction of a millisecond. While it is possible to reduce these errors by more frequent updates, this is practical only in primary servers where the radio clock can be read more frequently without imposing additional traffic on the network.

8. Acknowledgements

This research was made possible with equipment grants and loans from Sun Microsystems, Digital Equipment, Cisco Systems, Spectracom, Austron and Bancomm Divisions of Datum and the U.S. Coast Guard Engineering Center. Thanks are due especially to David Katz (Cisco Systems), James Kermitz (U.S. Coast Guard), Judah Levine (NIST) and Jeffery Mogul (Digital Equipment). Acknowledgement is also due to the over two dozen contributors to the NTP Version 3 implementation, especially Dennis Ferguson (Advanced Network Systems), and Lars Mathiesen (University of Copenhagen).

9. References

[ALL74] Allan, D.W., J.H. Shoaf and D. Halford. Statistics of time and frequency data analysis. In: Blair, B.E. (Ed.). *Time and Frequency Theory and Fundamentals*. National Bureau of Standards Monograph 140, U.S. Department of Commerce, 1974, 151-204.

[DAR81a] Defense Advanced Research Projects Agency. Internet Protocol. DARPA Network Working Group Report RFC-791, USC Information Sciences Institute, September 1981.

[DAR81b] Defense Advanced Research Projects Agency. Internet Control Message Protocol. DARPA Network Working Group Report RFC-792, USC Information Sciences Institute, September 1981.

[DEC89] Digital Time Service Functional Specification Version T.1.0.5. Digital Equipment Corporation, 1989.

[LIN80] Lindsay, W.C., and A.V. Kantak. Network synchronization of random signals. *IEEE Trans. Communications COM-28, 8* (August 1980), 1260-1266.

[MAR85] Marzullo, K., and S. Owicki. Maintaining the time in a distributed system. *ACM Operating Systems Review 19, 3* (July 1985), 44-54.

[MIL88] Mills, D.L. The Fuzzball. *Proc. ACM SIGCOMM 88 Symposium* (Palo Alto, CA, August 1988), 115-122.

[MIL89] Mills, D.L. Network Time Protocol (version 2) - specification and implementation. DARPA Network Working Group Report RFC-1119, University of Delaware, September 1989.

[MIL90] Mills, D.L. Measured performance of the Network Time Protocol in the Internet system. *ACM Computer Communication Review 20, 1* (January 1990), 65-75.

[MIL91a] Mills, D.L. Internet time synchronization: the Network Time Protocol. *IEEE Trans. Communications 39, 10* (October 1991), 1482-1493.

[MIL91b] Mills, D.L. On the chronology and metrology of computer network timescales and their application to the Network Time Protocol. *ACM Computer Communications Review 21, 5* (October 1991), 8-17.

[MIL92a] Mills, D.L. Network Time Protocol (Version 3) specification, implementation and analysis. DARPA Network Working Group Report RFC-1305, University of Delaware, March 1992, 113 pp.

[MIL92b]Mills, D.L. A computer-controlled LORAN-C receiver for precision timekeeping. Electrical Engineering Department Report 92-3-1, University of Delaware, March 1992, 63 pp.

[MIL92c]Mills, D.L. Modelling and analysis of computer network clocks. Electrical Engineering Department Report 92-5-2, University of Delaware, May 1992, 29 pp.

[MIL92d] Mills, D.L. Simple Network Time Protocol (SNTP). DARPA Network Working Group Report RFC-1361, University of Delaware, August 1992, 10 pp.

[NIS90] *NIST Time and Frequency Dissemination Services*. NBS Special Publication 432 (Revised 1990), National Institute of Science and Technology, U.S. Department of Commerce, 1990.

[POS80] Postel, J. User Datagram Protocol. DARPA Network Working Group Report RFC-768, USC Information Sciences Institute, August 1980.

[POS83] Postel, J. Time protocol. DARPA Network Working Group Report RFC-868, USC Information Sciences Institute, May 1983.

[RAM90] Ramanathan, P., K.G. Shin and R.W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer 23, 10* (October 1990), 33-42.

[SHI87] Shin, K.G., and P. Ramanathan. Clock synchronization of a large multiprocessor system in the presence of malicious faults. *IEEE Trans. Computers C-36, 1* (January 1987), 2-12.

[USNO] Daily time differences - series 4. U.S. Naval Observatory, Washington, D.C., (advisory published weekly).

[VAS88] Vasanthavada, N., and P.N. Marinos. Synchronization of fault-tolerant clocks in the presence of malicious failures. *IEEE Trans. Computers C-37, 4* (April 1988), 440-448.

[WIL90] Wilcox, D.R. Backplane bus distributed realtime clock synchronization. Technical Report 1400, Naval Ocean Systems Center, December 1990, 52 pp.