Time is an illusion.

LUNCHTIME DOUBLY SO.

—Ford Prefect to Arthur Dent in
The Hitchhiker's Guide to the Galaxy, by Douglas Adams

GEORGE NEVILLE-NEIL

One of the more surprising things about digital systems—and, in particular, modern computers—is how poorly they keep time. When most programs ran on a single system this was not a significant issue for the majority of software developers, but once software moved into the distributed-systems realm this inaccuracy became a significant challenge. Few programmers have read the most important paper in this area, Leslie Lamport's "Time, Clocks, and the Ordering of Events in a Distributed System" [1978],² and only a few more have come to appreciate the problems they face once they move into the world of distributed systems.

Any discussion of time should center around two different measurements: synchronization and syntonization. *Synchronization*, loosely defined, is how close two different clocks are to each other at any particular instant. If two clocks, or computers, claim that it is 15:30 at exactly the same moment, then they are considered to be in sync. The definition of "exactly the same moment" is where the first difficulty arises. Do you care about the same minute (15:30), the same second (15:30:00), millisecond, microsecond, nanosecond, etc.? The level of accuracy you wish to achieve

defines, roughly, the level of difficulty in attaining it.

Syntonization is the quality of the timekeeping of an individual clock. One might assume that a second is always a second, but in reality there needs to be a physical system from which to work and from which to derive time. Computers use quartz crystals as the basis of all of their internal operations. The 3-GHz CPU in your laptop or server is able to run at that speed because somewhere on the motherboard is a bit of quartz that, when electricity is applied to it, vibrates at a known frequency. That frequency is then multiplied and divided to drive the various parts of the rest of the system. If crystals—and other pieces of hardware—were unaffected by their environment, then all computers would have perfect syntonization. Alas, this is one place where the real world intervenes. Crystals experience changes in heat, power, and age, and no two of them are exactly alike, but they vibrate within a tested range provided on the data sheet from the manufacturer. When a server heats up—for example, because it is running a complex workload—then the crystal will oscillate faster, and the computer's internal clock will run ahead, or skew, faster. The effect of skewing on a typical server is shown in figure 1. If the air gets colder, then the crystal will oscillate slower and seconds will last longer, skewing time on the server negatively.

The level of syntonization that can be achieved with a particular crystal is also an economic problem. Cheaper crystals are less stable (i.e., more prone to skew) than more expensive ones. All commodity hardware—from the most expensive server to the cheapest tablet—has a very cheap,

1

FIGURE 1: PTP LOG GRAPH

low-quality crystal, which costs about 10 cents. A typical laptop or server, left without any type of external time conditioning, will drift out of sync within minutes and after a few hours may already be several minutes away from good synchronization with other systems.

To correct for poor syntonization the operating system has a set of routines to steer the clock. If the system is running too fast, then the routines will tell it to slow down; if it is running too slow, then they will tell it to speed up. These steering inputs must be applied gently, over a period

of time, to avoid pushing the system into hysteresis, a condition whereby the clock oscillates wildly around proper syntonization but is never able to actually achieve stability.

To provide proper steering to the clock, the system needs some idea of what is too fast or too slow, and for this it must have an external source of correct, or at least better, time.

Most developers may logically think the solution is to buy better machines, but to this day you cannot buy a commercial server or laptop where the built-in crystal is any better than the cheapest ones on the market. Perhaps in time, as more companies build distributed systems, this might change, but in 2015 it is not a practical option. To achieve better syntonization you can purchase a high-quality, stable crystal on a PCI (peripheral component interconnect) add-on board. The system clock is then conditioned, across the PCI bus, from this card. At about \$1,000 per card, however, this solution is practical for only a small number of servers, or for those with unlimited budgets.

Another way to condition the clock on a server is to have an external crystal oscillator distributed over a serial bus. Injecting a high-quality 10-MHz signal into a server via a serial line is another way to give a set of machines good syntonization. Distributing such a signal is reasonable for a small number of machines (up to 48) but breaks down as the number of connections and length of the cables increase. Wiring an entire data center for such a signal is, again, prohibitively expensive and prone to its own set of problems.

We will return to the question of how to synchronize the system to an external time source later, but first we must

address the question of how a program finds out what time it is on the system on which it is currently executing.

WHAT TIME IS IT NOW?

For the majority of software developers the concept of “now” is represented by a single system call, **gettimeofday()**, which returns to the caller the system’s idea of what time it is at the moment of the call. Whether this is useful information or not depends on the quality of the time measurement required. If the system needs to be accurate only to within a second, then the **gettimeofday()** routine is adequate— and it is often good down to the millisecond, although this can vary with the quality of the timekeeping software in the operating system. The tradeoff that must be made when deciding how to get the time in a program is between speed and accuracy. Asking for the time requires quite a bit of work on the part of the operating system software to give an accurate representation back to the user program. While **gettimeofday()** will work on almost any system, the **clock_gettime()** and **clock_getres()** routines give the programmer more flexibility in working with time.

Systems with the **clock_*** routines expose several types of clocks to user programs. For example, on the FreeBSD operating system, the clocks are **CLOCK_REALTIME**, **CLOCK_MONOTONIC**, **CLOCK_UPTIME**, **CLOCK_VIRTUAL**, and **CLOCK_PROF**. **CLOCK_REALTIME**, which reports time the same way that a wall clock does, is the most common source of time for systems that generate log files, as these log files will be read by human users who want to know what time an event

occurred. The other clock routines are more specialized and are documented in the system manual pages.

Each clock has two variants: **PRECISE**, which gets the most precise time possible but takes the longest to return to the caller; and **FAST**, which is the least accurate but the quickest to give the answer. These two variants describe one of the tensions in computer-based timekeeping quite well: the more precise you want the time to be, the longer it will take to retrieve it. Asking low-level hardware for the current time will normally return the most precise answer, but it will also take time because a protection boundary must be crossed and the machinery required to return an accurate time has its own nonzero overhead.

The **clock_gettime** and associated routines also exist on Linux but do not appear in the latest Mac OS X release [Yosemite]. Microsoft's Windows operating system has similar functionality to **gettimeofday()** but under a completely different name, **GetLocalTime()**.

Measuring a brief interval—for example, the time required for a few lines of C or C++ code to execute—can often be done with less overhead by using on-chip instructions. Modern Intel CPUs have a single instruction, **rdtsc**, which returns the on-chip Time Stamp Counter. When a program wishes to measure the time between two local, non-network events, this is the fastest and cheapest way to work with local time.

It is important to choose the right clock option for a particular application. If an interval time needs to be measured, then **rdtsc**, or a library wrapped around it, is the best solution, whereas getting the system time for use

in log files probably ought to be carried out using `clock_gettime()` with a `FAST` option, or, if `clock_gettime()` is not available, then `gettimeofday()`.

FIND A BETTER CLOCK

Now that we know how to get programs to ask the system for the time properly, and to measure local intervals, we can return to the problem of synchronizing systems with some form of external reference. This challenge is most often addressed by finding a better clock, periodically asking it for the time, and adjusting the system's local clock to match the external reference. One solution to finding a better source of time is to use the network, since all systems in a distributed system are, by definition, connected to the network. In the 1980s a group of researchers defined the Network Time Protocol [NTP].³ First documented in 1985 and subsequently updated in 1988, 1992, and 2010, NTP is the most successful Internet-based network time protocol in use.

Using UDP (Unreliable Datagram Protocol), NTP is implemented as a distributed system that has higher- and lower-quality clocks forming a hierarchy in which those clocks with a lower *stratum* number are of higher quality. A stratum 0 clock is considered a *reference clock* and contains a highly stable crystal, such as one based on cesium or rubidium, or it may be synchronized with GPS (Global Positioning System), which transmits a stable and reliable time signal. Systems at stratum 1 are connected, often via a direct cable, to a stratum 0 clock; stratum 2 clocks are connected to stratum 1; and so on, with this chain of time

reference extending to stratum 15. There is no defined standard for the time quality of a clock at any particular stratum, only that a lower number is expected to be of a higher quality. Not having a hard definition of quality at each layer has meant that improvements in technology, whereby more accurate clocks could be deployed at the lower (better) strata, did not require the renumbering of current systems. As stratum 0 and 1 clocks improved, so did the time available at stratum 2 and above.

NTP uses a polling mechanism to query the time on a set of clocks that are used as input to an algorithm, which is then used to discipline the local clock. Most operating systems have some version of the **adjtime()** system call that allows an external program such as NTPd (Network Time Protocol daemon) either to speed up or to slow down the system clock as it deems necessary. A globally distributed protocol such as NTP must act as a good network citizen, avoiding flooding the network with requests. In part because NTP was specified early in the history of the Internet, when long, slow, and unreliable links were the norm, its default polling interval is once every 64 seconds. Because of this long default polling interval, a new clock can take tens of minutes or more to synchronize with external sources; and once some level of stabilization has been reached, it is still possible for the clock to wander as new inputs arrive only every minute or so.

Increasing the polling frequency to the maximum of once per 16 seconds can increase the accuracy of the clock and the expense of added network load, but it is still possible to have time skewing by many milliseconds, depending on the

environment and the quality of the crystal. As discussed in the first part of this article, the low polling frequency of NTP usually prevents systems from achieving submillisecond synchronization.

DATA-CENTER TIME

The proliferation of data-center-scale distributed systems has taken a seemingly obscure protocol, first defined by the IEEE in 2002, and brought it to the attention of a broader audience. Originally designed to synchronize systems in factory automation, electrical power generation, and cellular networks, the Precision Time Protocol (PTP)¹ is now used in financial services—in particular, HFT (high-frequency trading)—as well as other data-center systems where submicrosecond synchronization is necessary.

The need for submillisecond synchronization in factory automation, electrical generation, and cellular networks is apparent when looking at the time scales involved. A robot arm moving at 30 miles per hour moves 44 feet in one second, a speed at which a collision with a human operator or a neighboring robot would be catastrophic. Electrical power networks running at 60-Hz AC need to have all of their shared components synchronized; otherwise, handing off energy from one block to the next could result in a catastrophic, and fiery, failure. The alternations in the current of two adjacent power grids need to be synchronized to well under 10 microseconds, with a typical rating at 4.6 microseconds. In cellular networks the synchronization requirements were once only necessary to handle the

smooth handoff of a mobile device between towers, but now that high-speed data services are being provided via multiple base stations, the synchronization requirements among the base stations themselves are ± 1.5 microseconds. It was for these applications that IEEE originally designed PTP.

The first decade of the 21st century saw the rise of HFT, which brought together algorithms and high-performance computing in an effort to extract large amounts of value from millions of very small financial transactions. HFT companies operate racks of machines that all look at market data in realtime and then make trading decisions based on differences between the perceived value of stocks and other financial instruments. The HFT world is dominated by a need for speed, but it isn't possible to do all of the work on a single large computer. To achieve scale, racks of servers are reading the same data simultaneously and making trading decisions in realtime. These trading decisions require knowing the order of events—which stock moved which way before another stock. Did Mobil go up before Shell went down, or was it the other way around?

The time scales on which these financial transactions take place move first to milliseconds, then tens of microseconds, and then into the range of 1 microsecond—and there are applications where an even smaller range would be useful. The HFT market not only was able to apply PTP to its problems, but also its collective financial might meant that PTP features began to appear in expensive—though still commodity—hardware. Many network interface card and network switches now directly support the timestamping

of PTP packets or include a version of the PTP daemon as an option.

PTP differs from NTP in several key ways. Having been designed for a small set of hosts, such as a factory floor or a set of cell towers, PTP is meant for use on a single network—in fact, most documentation and papers about PTP assume that there are no switch or router hops between the various systems. PTP can be routed over a network, but the loss in time quality is sufficient to dissuade people from this use in most environments. Someone who wants a routable, Internet-based time protocol would be better off using NTP.

PTP is designed as a multicast protocol in which there is a *grandmaster*, PTP's name for the better clock, and a number of slaves that are following the grandmaster. The grandmaster sends out a **SYNC** packet once per second with the current time. The slaves periodically send a multicast **DELAY_REQUEST** packet, to which the grandmaster replies with a **DELAY_RESPONSE**. The **DELAY_REQUEST/DELAY_RESPONSE** pair allows the slave to determine the one-way delay between itself and the grandmaster. Without this measurement the slave has no way of knowing when the **SYNC** packet actually originated at the grandmaster. Taken together, these packets allow the slave to update its own clock based on the time of a better clock (the grandmaster) and to do so in the face of some amount of *jitter*, the small variations in the transit time of a packet between master and slave systems.

PTP assumes a symmetric network, where packets take the same amount of time to propagate from the grandmaster

Jitter is an important consideration when PTP is being used to achieve good intersystem synchronization.

to the slave as they do from the slave to the grandmaster. If the times are asymmetric, then the offset calculated by the client may be skewed by some number of microseconds. The open-source PTP daemon, PTPd, can be configured to take this asymmetry into account, but if network conditions change—for example, because one network switch was replaced with a new and different model—then the configuration will have to be changed as well. Most deployments of PTP do not use any fixed offset and simply assume that the network propagation times are symmetric.

Network time protocols are particularly sensitive to jitter. If packets always required a constant amount of time to get from the master to the slave, then a simple offset could be programmed into the slave and it could use this offset, along with the time in the **SYNC** packets, to steer its clock. A simple test with the **ping** program will show that the time for packets to transit a single network segment is subject to quite a bit of jitter that will prevent a clock from being synchronized to within 10 microseconds—and certainly to within 1 microsecond.

Jitter is an important consideration when PTP is being used to achieve good intersystem synchronization. Sources of jitter include network equipment, including interfaces, cards, and switches, as well as software, such as the device driver and operating system. Network switches induce jitter when competing users of the network get in the way of network timing packets. All of the literature about PTP and all of the benchmarks from hardware providers describe conditions on otherwise unused networks. Running

the timing protocol on its own network is preferable, as it reduces the jitter induced by the network itself.

The current largest contributor to jitter in many deployments of PTP is in the software. When we took over the PTPd project (<http://ptpd.sf.net>), an open-source implementation of PTP, all timestamps were recorded in user space using `gettimeofday()` after a packet had gone through several layers of software, including the device driver, and the operating system's network stack. Each layer of software induces some level of jitter, as these layers are driven by timers, and packets show up asynchronously. Our first improvement was to move the timestamping into the kernel, recording timestamps using the `SO_TIMESTAMP` socket option. While this improved synchronization quality, it was not enough, and therefore timestamping was moved to just above the network driver code in the BPF (Berkeley Packet Filter). This resulted in achieving synchronization within 10 microseconds on a system that was not otherwise busy. The proliferation of PTP within the HFT world, where 10-Gb Ethernet was also prevalent, led several network card providers to add stable oscillators and timestamping hardware to their devices. At least one such vendor uses PTPd, adapted to the vendor's hardware, to achieve synchronization to within 500 nanoseconds of a grandmaster on an otherwise quiet network.

A lesser-known contributor to noise in the measurements comes from the way in which modern NICs (network interface controllers) are built. To achieve high bandwidth in a network interface it is important to batch many packets

together before they are all handed, in a group, to the operating system for processing. If each packet on a 10-Gb Ethernet network generated an interrupt when it arrived, that would result in 14.8 million interrupts per second, which is wasteful when moving data in bulk. Any NIC that supports speeds of 1 Gbps or higher is going to have some form of AIM (adaptive interrupt moderation), in which interrupts are gathered up without the device taking any action until a certain number have arrived. Having an option such as AIM left on introduces a sinusoidal wave, whose frequency is the interrupt moderation rate (in one instance this was 8 KHz), into the measurements, preventing the system from achieving a good level of synchronization.

One solution to delivering accurate time to servers, while also allowing them access to high-bandwidth connections, is to use the server's on-board 1-Gbps network port for the time service, and have a separate 1- or 10-Gbps network port for data. Providing two network connections for each host is typical in many data centers, as there is often an administrative network as well as a data network leading to each host. If the time data is piggybacked onto the administrative network, then the PTP packets should be placed in their own virtual LAN with a higher quality of service than the administrative data. Care also needs to be taken to avoid large data copies, such as those used to retrieve server log files, because this will introduce jitter to the time traffic.

A common practice for synchronizing time in a data center is to make the grandmaster a GPS-based clock with

a stable oscillator. While such a clock is expensive, on the order of a few thousand dollars, each data center may need only one, or perhaps two for failover, which is far cheaper than placing a stable oscillator card into each server. A good grandmaster will do all of its timestamping of packets just above the physical layer, much as expensive NICs do, in order to prevent any software-based jitter from entering the measurement.

MARCHING ON

Submicrosecond synchronization among a large group of distributed hosts is a significant achievement, but even higher levels of synchronization are going to be necessary for future applications. The proliferation of 10-Gbps Ethernet in data centers and the prospect of 25-, 40-, and 100-Gbps networking will require synchronization levels in the nanoseconds in order to determine whether the arrival of a packet at host A occurred before the arrival of another packet at host B. On a 10-Gbps network, packets can arrive only 67 nanoseconds apart; at higher speeds this number gets considerably smaller.

While the first data centers to apply PTP broadly are in the financial sector, many other applications are now realizing the need for higher qualities of synchronization than can be achieved with NTP. Any endeavor with a system that asks the question, “Who did what to whom and when?” and needs an answer that is correct to within less than a millisecond, will want to find a way to integrate PTP. In accordance with the IEEE requirement that all protocols

be reviewed and renewed every five years, the IEEE-1588 working group is currently discussing new features that may be present in version 3 of the protocol. Once the new standard has been approved, it will remain to be seen whether the hardware and software vendors follow.

1. IEEE Standards Association. 1588-2008—IEEE standard for a precision clock synchronization protocol for networked measurement and control systems; <https://standards.ieee.org/findstds/standard/1588-2008.html>.
2. Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21 (7): 558-565.
3. Mills, D. L. 1985. Network Time Protocol. RFC 958. Internet Engineering Task Force; <https://tools.ietf.org/html/rfc958>.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

George V. Neville-Neil works on networking and operating-system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

Copyright © 2015 held by owner/author. Publication rights licensed to ACM.