Time on Unix

Patrick Louis

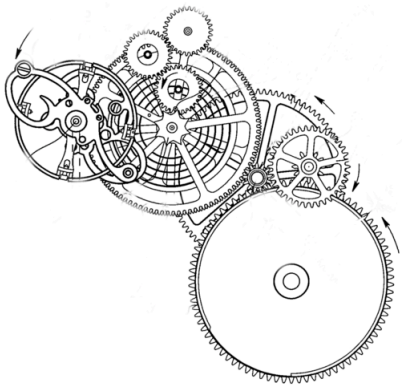
2020-05-02

Published online on venam.nixers.net

First published eBook format 2020

Contents

Introduction



Roseusa / CC BY-SA

- Time is relative
- Measuring time and standards
- Coordinating time
- Time zones
- DST

Time, a word that is entangled in everything in our lives, something we're intimately familiar with. Keeping track of it is important for many activities we do.

Over millennia, we've developed different ways to calculate it. Most prominently, we've relied on the position the sun appears to be at in the sky, what is called apparent solar time.

We've decided to split it as seasons pass, counting one full cycle of the 4 seasons as a year, a full rotation around the sun. We've also divided the passing of light to the lack thereof as days, a rotation of the earth on itself. Moving on to more precise clock divisions such as seconds, minutes, and hours, units that meant different things at different points in history. Ultimately, as travel got faster, the different ways of counting time that evolved in multiple places had to converge. People had to agree on what it all meant.

In physics, time is the progression of events, without events there's no time. It is defined by its measurement, what changes at a specific interval can be considered a unit, though still infinitely divisible.
In physics there are two ways to view time. In classical physics, time is absolute, independent of the perceiver, synchronized for everyone. While in modern

physics, we have Einstein's special and general relativity that applies, things depend on a frame of reference, time can dilate or contract with the effect of gravity, we talk of space-time.
In physics, equations work equally well in both ways, the math holds up, in the future and in the past. However, the arrow of time in our universe seems to go in a unique direction. Peculiarly, we'll see that time in computers, unlike in our universe, can actually go backward at specific events.

All of this to say that because of the importance of tracking time, we've created ultra-precise atomic clocks that have an error of 1 second every 30 million years. We can be categorically sure of the lapse that happens between two beats/oscillations, if there's an error then it's outside the human life-span, and we've got many of them to correct errors. Those clocks are our sources of truth, they give us the definition of the standard unit of second, SI second.

We have, on one hand, the atomic clocks counting time regardless of events happening around, and on the other hand, we have a moving planet in space that is subject to forces, where we've chosen the fact that one full orbit around the sun equals a year and that one full (approximate) rotation on itself is a solar day, the space between two transit of the sun (maximum height in the sky). Both of those values ought to diverge and differ eventually.
The earth, because of its unevenness and current position in its orbit, could rotate around the sun or itself faster or slower, its speed changing how long days and years are.

What we've done is used this standard definition of the SI second as our anchor. A day is now not defined by the apparent sun position but as the average number of standard unit seconds that make up an average stellar day, somewhat around 86400.002 seconds. This actual uniform clock time is called the mean time. Mean solar time is the exact average mean time for a day in a year. That is the sum of all solar days over n days.

Thus, clocks that have a uniform fixed value, mean-time, will differ with the apparent sun time. This difference is called the equation of time (EOT) and it can vary as much as 15min (ahead 14 minutes near February 6, behind 16 minutes near November 3) but rebalances itself as the earth finishes its orbit around the sun. There are many simulations you can find online to understand this concept.

As for years, our calendars can only hold entire days but the actual number of days it takes to finish an orbit is fractional. And so we accumulate this fraction over multiple years and add an extra day to the year that follows making it a leap year, 366 days instead of 365. On a Julian calendar, a year is 365.25, however this is not precise, it is higher than the actual number of days it takes to form a year: 365.242199. The Gregorian calendar, which is the most common today, defines it more precisely as 365.2425, adding a leap year 97 out of 400 years.

But we still use 86400 seconds to define a day in our current lives, in our software,

right? What about the rest of this complex system, how do we cope with these discrepancies, who chooses the mean time, how can we all sync on those values, who's in charge? Noon where you are might not be noon where I am.

The local time that shows on our clock is chosen by our local authorities, we call it civil time.
And because we all live on the same planet, instead of each syncing in our local community with what appears to us as the mean time, we can choose a fixed geographical spot, create a rigorous standard time there, and for the rest of the world, derive from it. Anything further away in longitude from this meridian can add a delta time difference. That way we can all sync and make less of a mess in this interconnected world.

The first major standard was set at the Royal Observatory in Greenwich, London. The mean time recorded there was used as the one to derive your local civil time as an offset from, called Greenwich Mean Time, or GMT for short. However, it was not as precise as it could be and thus got replaced in 1967 by another standard called Universal Coordinated Time, UTC.

UTC is a version of the Universal Time standard. In this standard we also find UT1, that keeps track of Earth rotation angle using GPS satellites, it is the mean solar time at 0 degree longitude, it's a better and more precise version of GMT.

UTC, instead of relying on the rotation of the earth, relies on the International Atomic Time (TAI), which is the time we talked about that defines precisely the SI second using 400 atomic clocks at multiple laboratories worldwide. Additionally, to keep count of the rotation of the Earth, and keeps in sync with UT1, the UTC authorities can add or remove a second in a day, a leap second. The difference between UTC and UT1 is DUT1, basically when DUT1 is one second we need a leap second.
So in UTC, a second is well known, but the number of seconds in a minute can vary between 59,60, or 61 if there was a leap second. Any unit bigger than the SI second reference can vary. Let's also note that UTC uses the Gregorian calendar as previously said.

As you could've guessed, introducing a leap second isn't a decision we take instantaneously, it's announced at least six months in advance in "Bulletin C" by the International Earth Rotation and Reference Systems Service (IERS) which is one of the authority. There's also involvement in the standard from the International Astronomical Union (IAU) and the International Telecommunication Union (ITU).

With this, we're set, we have a clean standard, now how should we divide the world such that civil local time keep in sync with the sun.

Time and longitude difference is what we need, we split the world into 24 meridians, each 15 degrees apart, each meridian zone represents one hour separation offset from UTC. Those are called time zones, they can go from UTC-12 to

UTC+14, and can sometimes be referred to by their name, for example Western European Time, Central European Time, etc… However, countries don't fall precisely on meridian, and thus local authorities can choose which section of the country follows which time zone as their civil local time, this difference doesn't even have to be an integer number of hours, it could be 15 or 30min for example.

Moreover, there's a practice called daylight saving time (DST), or summer time, which is used in civil time to advance forward the clock by one hour in spring and set it back one hour in autumn/fall. For example in winter the region could be on UTC+2 (EET) and in summer on UTC+3 (EEST). Creating a 23h day in late winter and a 25h day in autumn/fall. This practice is being reconsidered in the EU and planned to be removed by 2021.

So that's it we're all in sync!

Now on computers, how do we write time, how do we represent it textually.

Representing time

- locale
- tzdata

The easiest way I've found to test many formats is to use the `date(1)` command.

It can show the time both in human-readable format string and more machine-readable numeric formats.

Some formats include the timezone as numeric, others as the alphabetic time zone abbreviation. You can represent the date with or without the time zone, with it to make it more precise to the viewer.

Some formats prepend the time zone part with the 'Z' character, which has origins in the GMT standard but that now stands for zone description of zero hours, also sometimes called "Zulu time".

We can see that the date command automatically knows our time zone and how to display the time in a way we can read it. How do we set this format and where does it take it from. How to set the time zone.

Let's start with the formatting.

The date commands relies on `locale`, which is an internationalization mechanism used by Unix-like operating systems. Locale are configurations that can be used to set the language, money, and other representational values that can change by location. The libc on the system, and consequentially the coreutils, should be aware of those locale values.

The specific category of locale we are interested in is the `LC_TIME`, which is used for the formatting of time and date information, spelling of weekdays, months, format of display, time zone, 24 vs 12h format, etc.

To inspect specific values in `LC_TIME` you can do, see `man locale(7)` for info:

```
$ locale -ck date_fmt
LC_TIME
date_fmt="%a %b %e %H:%M:%S %Z %Y"
```

The available locale are usually found in:

```
/usr/share/locale
```

Locale can also be set on a user level in `~/.config/locale.conf`, or `$XDG_CONFIG_HOME/locale.conf`, or generally `$HOME/.config/locale.conf`.

All of this works because of the way profiles are loaded in the shell, you can take a look at `/etc/profile.d/locale.sh`.

Now regarding the time zone.

The time zone information database is distributed by the IANA, it's referred to as the tz database. The Unix distribution downloads when updated and installs it in `/usr/share/zoneinfo/` so that other libraries and programs can use it. In the tz data/zoneinfo db we can find all the information required to keep track of time in specific places. It's distributed in such a way to make it easier to choose time zone by choosing the city name or location instead of the exact drift/skew from UTC. That takes care of all the differences in civil time, all historic weirdness over time, daylight saving, and leap seconds.

To change the timezone globally we have to link `/etc/localtime` to an entry in `/usr/share/zoneinfo/`. For instance:

```
ln -s /usr/share/zoneinfo/America/New_York /etc/localtime
```

Many Unix-like system provide helpers to not have to manually link it. There's `timedatectl(1)` from systemd and `/etc/timezone` on Debian, for instance.

The `TZ` POSIX environment variable can also be used to specify the zone in human-readable format on the command line, and the `TZDIR` to specify the location of the tzdata. That means separate users on a single system can have different time zones.

Example:

```
TZ='America/Los_Angeles' date
```

The format of the tz database aka tzdata is explained in details in the `man tzfile(5)`. To create it you can use a textual format, and convert it using the command `zic(1)` (zone information compiler).

Example of creating your own tzdata:

```
> echo "Zone COOL +2:25 - COOL" > COOL.zone
> mkdir ./zoneinfo
> zic COOL.zone -d ./zoneinfo
> TZDIR=zoneinfo TZ=COOL date
# Should Output something similar to
Sun 12 Apr 2020 11:44:00 AM COOL
```

So now programs that rely on the standard `time.h` header can be aware of the time zone info. You can also load your own dynamically using `tzset(3)` from the `TZ` env.

Where do we usually find time on Unix

- POSIX time
- Uptime
- time(1)
- Programming language and timestamp
- File system atime, ctime, mtime, etc.
- Cron

Time is used in so many places in our operating system. We're going to list some common places where it is found and then build on them to approach more complex topics in the following sections.

Let's start with the infamous POSIX time. POSIX time, or Unix time, or Epoch time, is the number of seconds that have elapsed since Epoch time, which is the 1st of January 1970 00:00:00 UTC, minus the leap seconds difference, so Unix time is the same everywhere. That means that in Unix time, each day is considered to be exactly 86400 SI seconds, which would mean it should skew away from real UTC, and thus drifts away from UT1 (mean time).

To answer this, when there's a leap second introduced in UTC, POSIX time will repeat the same second or omit one because its minute cannot go over 60 seconds unlike real UTC.

Some rare systems can use TAI (International Atomic Time) as a source instead of UTC but will need a table on the system with the leap seconds in it to know how to calibrate itself to civil time.

Because Unix time start in 1970, dates that are before this need to be represented as negative numbers. However, prior to this date we have to keep in mind that there might not have been the UTC standard yet and thus it's better to rely on something else to represent the time and date accurately. Some real time operating systems (RTOS), that we'll see in a bit, like QNX choose to represent Unix time as an unsigned integer, which means it cannot represent time before 1970.

So is the Unix time a signed or an unsigned integer.

Historically, Unix time was a signed 32-bit integer that incremented at 60 Hz, the same rate as the system clock on the hardware of early Unix system, that is 60 times per second.
Epoch differed too, it was the 1st of January 1971 in the first edition of Unix Programmer's Manual. However, at 1/60th of a second precision, the 32-bit integer would have used all its range in only 2.5 years. Thus, it was changed again to the Epoch of 1 January 1970 at the rate of 1Hz, an increment of 1 bit every second. This gave about 68 years after 1970 and 68 years before 1970 using a 32-bit signed integer.

When the concept was made there was no consideration of all the issues with leap seconds, time zones, leap years, we've mentioned previously. It's only in the 2001 edition of POSIX.1 that there was some rectification about the faulty leap year rule in the definition of Unix time.

So what's this Unix time used for?

Unix time is the value that Unix system look at to keep track of their system time. Its value is kept in a `time_t` type format, defined vaguely in POSIX.1 as previously mentioned, and that can be included via the `time.h` header. POSIX only mandates that it should be an integer, and doesn't say anything about if it should be signed or not, explaining why QNX chose an unsigned integer.
For more precise time manipulation, the `time.h` and `times.h` header also includes other types such as `struct timeval`, `struct tms`, and `struct timespec`.

However, usually it is a single signed number which size is defined per system. For example on mine, `<time.h>` redirects to `<bits/timesize.h>` as a 64 bits signed integer.

One issue called the Year 2038 problem, Y2038, or Y2k38, is when system that chose to represent Unix time as a signed 32 bit value reach the maximum range of 32-bit integers and overflows, creating an undefined behavior.
This issue is completely avoided using a 64-bit signed integer.

Let's move to another topic, `uptime`.

The uptime of a machine is a measure of how long the machine has been running since the last reboot, the current time minus the time it was booted at.
Some system may require high availability due to service level agreement and this is one of the measure that can be looked at. However, high uptime can be a sign of negligence and rebooting after a long time may lead to unexpected consequences as some changes may only happen on reboot.

Most Unix-like OS come with the BSD `uptime(1)` command which shows the current time, how long the system has been running, how many users are currently logged in, and the system load average for the last 1, 5, and 15 minutes (Though those values aren't good metrics, see Brendan Gregg's blog). Load average being the average number of processing running at that time. It's the same information one can find in the first line of the `w(1)` command.

On Linux the uptime can also be found in the `/proc` filesystem, as `/proc/uptime`, the file containing 2 values, the first one being the number of seconds elapsed since last reboot and the second how much time each core has spent idle in seconds, which are both indicator of system usage.

Another command that is common as a metric is `time(1)`. It is a simple command which reports the time a command has taken.
By default, it reports on real time, user time, and sys time. Real time means the wall-clock time, it's the total time for everything to execute from start to finish. User time is the amount of cpu time spent in user mode, while sys time

is the amount of time spent in kernel mode, system calls time and more. CPU time is calculated as how much each cpu uses, so if you're on a multicore system you may have two cores executing in 0.1s in parallel in user mode and have a total user cpu time of 0.2s. This shows that this has no direct relation with the actual time elapsed from start to finish without knowing which cores have executed, but this gives a small idea.
This article goes into the internals of `time(1)` GNU implementation.
Another related command is `times(1)`, which is part of POSIX and the equivalent of `times(2)` system call. It shows usage times of both the current process and the sum of its children in a two-line output.

In fact, there exists a panoply of commands that can be used for benchmarking processes and what they use, how much time they spend in which particular section of their code. We'll keep it at that.

That said, we often have to take snapshots of time in our programs, timestamps, saying this has happened at this specific time, attaching a metadata. These timestamps can be stored in Unix time, or UTC time, or in the specific time zone. However, time-stamping records with local time isn't advised because of issues that could arise with daylight saving, it's much easier to recalculate the offset from UTC. Though in certain cases, it would be valuable to know in which timezone the event happened, such as when timestamping pictures.

Examples of metadata timestamps are the Unix atime, ctime, mtime that are stored with files inodes on the file system.

Atime is the last access timestamp changed whenever the file is read or executed for instance, ctime is the last status change timestamp changed whenever the file is written to or its inode changes, mtime is the last modification timestamp changed whenever the file has data written to it, is created, truncated, etc.
An additional non-standard timestamp that we can find on some system is btime, the file creation/birth timestamp.

Additionally, some filesystems support flags related to those timestamps, usually for optimization to avoid disk load, such as ones that change the way atime is updated, remove access time from directories, etc.
This is the prevalent default on a lot of filesystems and thus gives a false sense of the definition of those timestamps.

To have more information about when those timestamps are actually updated have a look at `man 7 inode`, and check your file current values by calling the `stat` command or related system call on it. You can also use `touch(1)` to arbitrarily change the timestamps on files.

Let's end this section with timers and chronometers that trigger events at specified time, Unix alarm clocks if you want, clock daemons.

The de facto implementation of this is cron, which first appeared in Unix V6, a clock daemon tool written by Ken Thompson, in reference to the word Chronos which is the Greek word/prefix for time.

Cron specialized in scheduling the execution of programs periodically, at certain time events. It registers the events in a table.

It initializes its entries from directories containing the scripts: `/etc/crontab` or `/etc/cron.*/` or `/var/spool/cron`, you can take a look at `man 8 cron` for more info on that.

It can also be managed from the command line, for instance:

```
crontab -e
```

Cron will by default execute the entries using sh, that means those are simple shell scripts, letting you set environment variables, and more.
In the crontab you have to specify at which time to repeat the execution of events using a special syntax composed of every minute, every hour, every day, every month, every week day.

Despite cron being the go-to solution for repeated scheduled execution, others have created new solutions. Namely, init systems and service managers have tried to re-implement cron their own way and integrate timers as a type of service. Centralizing the management of timers along with services.

Prominently, systemd implements this function using `systemd.timers`, which are systemd unit files and inherit all the facilities and security that systemd provides for them.

You can list current running timers on systemd using:

```
sudo systemctl list-timers --all
```

`man 5 systemd.timer` provides all the info you need about systemd timer units.

System time, hardware time, internal timers

- Hardware time vs system/local time
- Clock hardware sources and configurations
- Tickers, timers, and their usages

We've said previously that POSIX time was used by Unix-like OS to keep track of system time, but we've cut it short at that. There's still a lot to add to this, like where does the system get its time to begin with, how does it store it when it's not running, what in the OS triggers the count of seconds, etc.

In this section I'll limit myself to Linux and FreeBSD as examples but the concepts should apply to any other Unix-like OS. I've chosen to do this because resources were scarce on this topic as most have chosen to not mention it and skip directly to the topic of NTP, Network Time Protocol, which we'll see in the next section.

There are two types of clocks on our machines, the first type goes by the name of RTC (Real Time clock)/CMOS Clock/BIOS Clock/Hardware clock, and the other by the name of system clock/kernel clock/software clock.
The system clock is the one we've mentioned before, that keeps track of time using a counter of seconds after the Epoch, and the hardware clock is one we haven't mentioned that resides physically on the system and runs without any interaction.
Their usages differ, the system clock runs only when the system is on, and so is not aware of time when its off, and the hardware clock has the purpose of keeping time when the system is not running.

As you would have guessed, any two clocks are bound to drift apart eventually, those clocks will differ from each other and from the real time. However, there are many methods to keep them in sync and accurate without using external sources.

Hardware clocks are usually found on the PC motherboard and interfaced with using an IO bus. Because some of those are on standard architecture such as the ISA (Industry Standard Architecture), it can be easy to know how to query and modify them. However, it's still hardware dependent and so can vary widely.

These clocks run independently of any control program and are even running when the machine is powered off, kept alive by their own power source, a small battery.
They are normally consulted only when the machine is turned on, to set the system time on boot. Unfortunately, they are known to be inaccurate, but weirdly inaccurate in a predictable way, gaining or losing the same amount of time each day, drifting monotonously, a systematic drift.

Hardware clocks don't have to store time as Unix time or UTC, and don't have to be limited in precision to seconds. It's up to the hardware implementation

to decide what can be done and on the user to decide what to do with it. In theory these clocks have a frequency that varies between 2Hz and 8192Hz, from 0.5s to 0.1ms precision.

Let's also note that there can be more than one hardware clock on a system.

Linux and FreeBSD come with drivers to interact with RTC.

On Linux for example, the RTC clocks are mapped to special device files `/dev/rtc*` backed by the driver. The star denoting the number of the clock if there are many, and `/dev/rtc` being the default RTC clock.
As with anything hardware, there could possibly be issues with the driver of the RTC and the clocks might not be mapped properly, especially if not following the industry standard. Linux has fallback mechanisms for other systems it wants to support.

On the other side, the only time that matters is the one you see when the system is running, and that is the system time.

As we said, system time is the number of seconds since the Epoch that is stored and kept track of by the kernel, however internally it might be more precise than seconds, it could go up to the precision offered by the architecture. We'll come back to this topic of high precision soon, just keep the simple concept in mind.

The system time, when displayed to us, refers to the timezone information and files we've previously mentioned. It's good to know that the Linux kernel also keeps its own timezone information for specific actions such as filesystem related activities, and this kernel timezone is updated at boot or via the utility `hwclock(8)` by issuing `hwclock --systz`.

When booting, the system clock is also initialized from the RTC that keeps running when the system is off. In some cases it can be initialized from external sources and not rely on the RTC.

Thus, when the system is running the hardware clock is pretty much useless, and we could do whatever we want with it. However, we have to beware of discrepancies on reboot.

The counter that the kernel uses to increment the system clock is usually based on a timer functionality offered by the Instruction Set Architecture of the CPU (ISA not to be confused with the other ISA we spoke about, the Industry Standard Architecture). In simple terms, that means that the CPU gets interrupted at known programmable intervals periodically, and when it's interrupted it executes a timer service/event routine. The routine is responsible for incrementing/ticking the system time and do some housekeeping. We're going to discuss this later.
Let's note that the frequency of the interrupt can be configured for better precision.

To set the system time and date, we can rely on the `date(1)` command, which takes many formats via its `--set` option.

For example:

```
date --set "5 Aug 2012 12:54 IST"
```

We could also initialize the system time from a remote server using `rdate(1)`:

```
rdate -s time.nist.gov
```

Or even, on some system, rely on the service manager. The infamous `timedatectl(1)` of systemd comes to mind, which can set and give information about pretty much everything we're mentioning in this section.

Example of output:

```
               Local time: Fri 2020-04-17 12:40:00 EEST
           Universal time: Fri 2020-04-17 09:40:00 UTC
                 RTC time: Fri 2020-04-17 09:40:01
                Time zone: Asia/Beirut (EEST, +0300)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: no
```

What's this line about "RTC in local TZ", can the time on the hardware clock be stored with timezone info, and why would we do this? What is stored on this clock, is it UTC or local time?

The answer to this, like most things, is that it depends. The time on the RTC can be configured to be set to whatever the system wants it to be. Yet, storing it in UTC is the best choice as UTC doesn't change regardless of time zones and daylight saving. Having the RTC storing the local civil time means that it would need to be aware of all the complication that it implies, which most RTC clock won't. If in local time, and the system has been down for a while, RTC might differ with actual local time. And even with clocks that have the ability to do the daylight saving themselves, the feature is mostly unused.

So it's preferable to store the time of the RTC in UTC but some systems still choose to not adhere to this. For instance, when dual-booting, the other operating system may expect the RTC to contain local time and update it accordingly. That creates a discrepancy, and the RTC has no way to indicate if it is storing local time or UTC, hence the OS has to keep track of this information internally. This is the kind of scenario that gives rise to the rule of not letting more than one program change the time of the RTC.

On FreeBSD, this information is given via the `/etc/wall_cmos_clock` file: if the file exists it means that the hardware clock keeps local time, otherwise the hardware clock is in UTC.
On Linux, this information is passed to the kernel at boot time via the `persistent_clock_is_local` kernel parameter/stanza (see notes `timekeeping.c`). The RTC can also be queried and set in localtime or UTC via the `hwclock(8)`

options `--localtime` or `--utc` which indicate which timescale the hardware clock is set to, hwclock will store this info in `/etc/adjtime`.

Hence, we have to keep those clocks in sync. The best way to do this is to rely on the predictable inaccuracy/systematic drift/instrument bias of the hardware clock. We can measure its drift rate, and apply a correction factor in software.

On Linux there are two tools to perform this, `hwclock(8)` and `adjtimex(8)`, while on FreeBSD there is `adjkerntz(8)`.

`hwclock(8)`, and its predecessor `clock(8)`, let you query, calculate drift, and adjust the hardware clock and kernel/system clock in both direction. While with `clock(8)` you had to calculate the drift manually, `hwclock(8)` does it automatically.
It does it by keeping track, in an ASCII file called `/etc/adjtime`, of the historical information, the records of how the clock drifts over time, and if the hardware clock is in UTC or local time, as we said before.

Here are some example runs:

```
# adjust drift of RTC
> hwclock --adjust

# set RTC to the same time given by --date
> hwclock --set --date='19:30'

# set the RTC from system clock
# and update the drift at the same time
> hwclock --systohc --update-drift
```

Thus, it would be a good idea to call `hwclock(8)` periodically in a cron job to keep the hardware time in sync and calibrate the drift.

On FreeBSD, the utility `adjkerntz(8)` is used similarly but only for local time RTC. It's called at system startup and shutdown with the `-i` option from the init system before any other daemon is started and sets the kernel clock from the RTC, managing the DST and timezone related configuration.

Taking a look at some `hwclock(8)` options gives us an idea about many RTC quirks.

We can select the clock device using these:

```
# if it's an ISA system
--directisa
# and maybe precise the device file explicitly
--rtc=filename
```

We can set the Epoch as something other than 1970:

```
# get it
--getepoch

# set it
--setepoch
```

However, this is only available on machines that support it.

We can specify if it's a clock that has issues with years above 1999 (which I can't find in the man page on my machine though):

```
# indicate the clock can't support years
# outside the 1994-1999 range
--badyear
```

As for the `adjtimex(8)` tool, on Linux, it doesn't actually change the hardware clock at all but specializes in the nitty-gritty details of the system/kernel clock and its relation with hardware.

It's especially useful for manually readjusting the system clock based on the drift of the RTC and raw access to kernel settings related to system time.

For instance, it can be used to change the speed of the system clock, telling it how much to add to time whenever it receives an interrupt. For example if the system clock ticks faster than it's supposed to be, it could be made to tick slower or it could be made such that each tick represents a smaller value to add to the time, both options are possible.
Those are done through the `--frequency` and `--tick` options respectively.

It can also be used to change the offset/drift, apply adjustment to the system time, and what affects the hardware clock.

Interesting options are the `-c` and `-a` which keep comparing the system time and hardware clock time every 10s and print the ticks and frequency offsets, which can be useful for estimating the systematic drift and then store it in `/etc/adjtime`, which `-a` actually does.

Example of a run:

```
                                    --- current ---    -- suggested --
cmos time     system-cmos  error_ppm   tick      freq    tick      freq
1587136817       0.277212
1587136827       0.278389      117.7  10000  -1701056
1587136837       0.279261       87.2  10000  -1701056    9998   5690519
1587136847       0.280304      104.3  10000  -1701056    9998   4571769
```

So my system considers 10000 ticks to be equal to 10s, basically having 1K ticks a second, but it's suggested that I use 9998 per 10s instead.

Note also the `error_ppm`, ppm (part per million), meaning that I've gotten a delta error of around 103 ticks per million that I need slew forward.

The `-p` option prints the internal kernel parameters related to time ticking.

```
         mode: 0
       offset: -852985
    frequency: -1701056
     maxerror: 483000
     esterror: 0
       status: 8193
time_constant: 7
```

```
    precision: 1
    tolerance: 32768000
         tick: 10000
     raw time:  1587136914s 254051439us = 1587136914.254051439
```

The status is a bit mask that represents the following:

```
    1   PLL updates enabled
    2   PPS freq discipline enabled
    4   PPS time discipline enabled
    8   frequency-lock mode enabled
   16   inserting leap second
   32   deleting leap second
   64   clock unsynchronized
  128   holding frequency
  256   PPS signal present
  512   PPS signal jitter exceeded
 1024   PPS signal wander exceeded
 2048   PPS signal calibration error
 4096   clock hardware fault
 8192   Nanosecond resolution (0 is microsecon)
16384   Mode is FLL instead of PLL
32768   Clock source is B instead of A
```

PPS standing for Pulse Per Second, PLL for Phase-lock loop, and FLL for Frequency-Locked Loop, which are different clock circuitries/feedback-loop, discipline, and slewing techniques, basically different methods of looping for time adjusting frequency and ticks to match real one, which are affected differently by the environment and have ups and downs.

What we can deduce is that my clock with status `8193` has PLL updates and a nanosecond precision.

That means that if 1000 ticks make up a second, a tick happens every millisecond. Can't we have more precise ticks, aren't I supposed to get nanosecond precision. But wouldn't that clog the CPU altogether, can we get multiple timers too. We'll see that with high resolution clocks later on, for now, again, let's keep those questions and concepts in mind.

An interesting line in the output of `adjtimex` catches our attention, bit 6, or 64 in decimal, "clock unsynchronized" what does it mean. It's similar to the "System clock synchronized: yes" line of systemd's `timedatectl` output.

An inspection of the Linux kernel source code let us know that there's a mechanism in the kernel to automatically synchronize the hardware clock with the system clock. It goes under the name of NTP "11 minute mode" because it adjusts it every 11 minutes.

Many other Unix-like operating systems choose to do this, have the kernel be the only program that syncs hardware time to system time. To not let other programs have to worry about all the drifting and calculation. In this case we don't need to create a cron job that adjusts the time, the kernel already does

it for us. Sometimes however, the kernel won't record the drift time anywhere while in this mode.

So it is synchronized by default on my system.

On Linux, the only way to turn it off is to stop the NTP daemon (network time protocol daemon, which we'll see in the next section), call any program that sets the system clock from the hardware clock such as `hwclock --hctosys` or actually recompile the kernel without the related option: `RTC_SYSTOHC`.

The `ntptime(1)` command also shows the Linux kernel `time_status` value:

```
  status 0x2001 (PLL,NANO),
```

Let's move to this high precision topic we've kept in mind.

So my system clock has nanosecond precision, are there ways to get higher precision from my system clock, what does that mean from the timer interrupt perspective. How much time should we spend on timer events handling instead of executing programs. How are they implemented in the instruction set, do I have a choice of clocks, are there other instructions to call. How do I check all that?

We said briefly that system time was kept track with using interrupts that were generated at predefined time or at specific periodic intervals. Whenever they happen, the kernel needs to handle time-based events such as the scheduling of processes, statistics calculation, time keeping (time of day), profiling, measurements, and more.
Different machines have different kinds of timer devices that can create this functionality. The job of the OS is to try to provide a system that unifies them abstractly to handle timer events for specific usages, using the best types of timer for the type of event its handling. It does it by programming them to act periodically or one-shot and keeping track of the event subscriptions that it'll need to handle.
Which hardware is available depends on many factors, but the most important one is related to the CPU and thus the architecture of the platform and its instruction set. Let's see what sort of timers we can find in our systems today that we could possibly choose from as clock event device.

- RTC - Real Time Clock

We could choose to actually rely on the RTC directly and not anything else. However, that comes with a cost because its quite slow, it ticks every 0.5s to 0.1ms, to stay energy efficient. So let's leave the RTC for boot time only and not timers.

- TSC - Time Stamp Counter

The Time Stamp Counter is a 64 bit register called TSC present on all x86 processors since the Pentium. It's connected via a CLK input pin that also drives the CPU clock and thus ticks as the same frequency. For example a

2GHz CPU will make this register tick every 0.5 nanosecond. The TSC register can be queried by the `rdtsc`, read tsc, instruction. It's very useful as it ticks along with the CPU, it can help us calculate the time, for example, if we can know the frequency of the CPU, giving us precise measurements. However, it's not so precise if the frequency can change over time.

- PIT - Programmable Interrupt Timer

The Programmable Interrupt Timer is also part of an instruction set. The way it works is that it can be programmed to send global interrupts after a certain time has elapsed, one-shot or periodically. It has a 16 bit precision and variable frequency rates that can be configured.

- APIC - Advanced Programmable Interrupt Controller

Similar to PIT in such that it can issue one-shot or periodic interrupts. Its size is 32 bits. And the interrupt it sent to the specific processor that requested it instead of globally (which PIT would do). Its frequency is based on the bus clock signal and can also be controlled but less flexible than PIT.

- ACPI_PM - ACPI Power Management Timer

ACPI Power Management Timer is part of the ACPI-based motherboards and have quite a low frequency of 3.58MHz, ticking every 279ns. It isn't nearly as accurate as other timers but has the advantage that it isn't affected by change based on power-management. It should be used as a last resort system clock.

- HPET - High Precision Event Timer

The High Precision Event Timer is a chip integrated into the Southbridge. It provides multiple hardware timers, up to eight 32 or 64 bit independent counters, each having their own clock signal and a frequency of at least 10MHz, 100ns. It is less precise than the TSC but has the advantage that it is separated and has multiple clocks.

It's always good to keep in mind that all those numbers about precision are for the best case scenario and that we may have overheads. We still have to remember that, for example, when querying the TSC we have to first issue the `rdtscp` command which has to be interpreted. It doesn't mean that we have a machine ticking at 0.5ns that we're going to be able to measure such intervals precisely.
Regarding TSC, we can only use it as a real time counter when it is stable. If it changes with CPU frequency we can't rely on it to calculate time properly as the distance between ticks will vary. TSC are categorized as "constant", "invariant", and "non-stop", or none. "Constant" meaning TSC stops during C state transition, C state referring to the low power mode of the CPU. "Invariant" meaning frequency isn't affected by the CPU state. "Non-stop" meaning it's both "invariant" and "constant".

On Linux you can check the features your CPU supports by consulting the flags in `/proc/cpuinfo`.

Example:

```
flags : tsc constant_tsc nonstop_tsc tsc_scale
```

NB: `tsc_scale` is used for virtualisation.

Before checking for the availability of the hardware timers and what is currently set for what on your system, let's take a moment to understand where we can use these timers.

There are in general 3 uses for the timers we've seen: clock source, clock event, and clock scheduling.

The clock source is the one that provides the basic timeline, it should be a continuous, non-stop (ideally), monotonic, uniform timer that tells you where you are in time. It's used to provide the system time, the POSIX time counter. When issuing `date` this is what is going to be consulted.
So the clock source should have a high resolution and the frequency should be as stable and correct as possible, otherwise it may require an external source to sync it properly.

Clock events are the reverse of clock source, they take snapshots on the timeline and interrupts at certain point, providing a higher resolution. They could in theory use the same hardware as the clock source but are not limited to it. They could use all the other hardware that are specialized in sending interrupts after a programmed time to trigger the events on the system timeline. It's also interesting to have the events triggered per CPU to have them handled independently, so APIC is especially useful here.

The clock scheduling is about how time affects the scheduling of processes on the system, what timeslice should be used to run processes and then switch to another one. This can possibly be the same counter as the clock source, however usually it needs smaller intervals as it has to be very fast but doesn't have to be accurate.

The clock source keeps time as a counter we refer to as jiffies. Jiffies are used to keep the number of ticks that happened since the system has booted, it is incremented by 1 at each timer interrupt. The number of ticks/interrupts in a second is denoted by a constant defined at compile time or in a kernel parameter called `HZ`, for Hertz, it's named this way in most Unix-like OS.
That means that there are `HZ` ticks in a second, thus there are `HZ` jiffies in a second. So that means `HZ` represents the precision of our clock source, and thus system time. For example, if `HZ=1000`, that means the system time has a resolution of 1ms (1K/HZ seconds).

On Linux you can check that value using:

```
getconf CLK_TCK
```

However, it is deprecated and will always return 100 (10ms), regardless of the precision. The actual value has to be set as a kernel parameter in `CONFIG_HZ`.

```
CONFIG_HZ_300=y
CONFIG_HZ=300
```

Nonetheless, it isn't such a good idea to go to higher precision `HZ` because if scheduling relies on jiffies it could affect performance.

Now let's check how we can see which devices we support and change the clocks.

On Linux, there's not many options regarding anything other than the clock source (system time). To check the ones available and the one currently in used you can rely on the `/sys/` filesystem.

```
> cat
    /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm

> cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

The clock source can be changed while the system is running by echoing the new clock to the same location:

```
> echo hpet >
    /sys/devices/system/clocksource/clocksource0/current_clocksource
```

For permanent changes you can recompile the kernel with different options or set the clock at boot by passing it as the `clocksource` option to the Linux kernel (kernel stanza) in grub or any other boot-manager.

```
linux    /boot/vmlinuz-linux
    root=UUID=12345678-1234-1234-1234-12345678 rw quiet
    clocksource=acpi_pm hpet=enable
```

Additionally, you can enable or disable `hpet` to be used as the base time for events clock.

As of today here are the relevant configurations and different clock sources for multiple CPU architectures:

```
clocksource=    Override the default clocksource
        Format: <string>
        Override the default clocksource and use the clocksource
        with the name specified.
        Some clocksource names to choose from, depending on
        the platform:
        [all] jiffies (this is the base, fallback clocksource)
        [ACPI] acpi_pm
        [ARM] imx_timer1,OSTS,netx_timer,mpu_timer2,
            pxa_timer,timer3,32k_counter,timer0_1
        [X86-32] pit,hpet,tsc;
            scx200_hrt on Geode; cyclone on IBM x440
        [MIPS] MIPS
        [PARISC] cr16
        [S390] tod
        [SH] SuperH
```

```
        [SPARC64] tick
        [X86-64] hpet,tsc

hpet=       [X86-32,HPET] option to control HPET usage
        Format: { enable (default) | disable | force |
            verbose }
        disable: disable HPET and use PIT instead
        force: allow force enabled of undocumented chips (ICH4,
            VIA, nVidia)
        verbose: show contents of HPET registers during setup
```

The process is quite similar on FreeBSD. By default, it's aware of the timer available on the system, it automatically ranks and chooses the best possible ones.

It has three timekeeping, one it calls `hardclock` running at 1000HZ (1ms), which is the same as the clock source, one it calls `statclock` used for statistics and scheduler events with a frequency of 128HZ, and a last one called `profclock` which is a bit higher in precision, 0.125ms. Those obviously can be tuned to preference.

To see the list them via you can use `sysctl`:

```
> sysctl kern.eventtimer
# or
> sysctl -a | grep kern.eventtimer
```

This should return the list of possible timers in the `kern.eventtimer.choice` entry.

Example output:

```
kern.eventtimer.choice:             HPET(550) LAPIC(400) i8254(100)
    RTC(0)
kern.eventtimer.et.LAPIC.flags:     15
kern.eventtimer.et.LAPIC.frequency: 0
kern.eventtimer.et.LAPIC.quality:   400
kern.eventtimer.et.i8254.flags:     1
kern.eventtimer.et.i8254.frequency: 1193182
kern.eventtimer.et.i8254.quality:   100
kern.eventtimer.et.RTC.flags:       17
kern.eventtimer.et.RTC.frequency:   32768
kern.eventtimer.et.RTC.quality:     0
kern.eventtimer.et.HPET.flags:      7
kern.eventtimer.et.HPET.frequency:  14318180
kern.eventtimer.et.HPET.quality:    550
```

The current time should be stored in the `kern.eventtimer.timer` entry.

The documentation about what the flag means can be found via the manpage `eventtimers(4)`, they are related to what the clock supports (periodic or not, per CPU or not). Those values can be changed in the `/etc/sysctl.conf` file or tunable via `sysctl` on the command line.

As with Linux, on FreeBSD hpet can be used for events if the driver is installed and enabled, it's part of the acpi. FreeBSD offers some beautiful documentation

about it in the `hpet(4)` manpage, discussing the configurations too. For instance if it can be used to support event timers functionality, and tune how many timers on the hpet per CPU can be used.

So now we should be all set, if we call POSIX functions part of `<time.h>` such as `gettimeofday` we can get the result in a structure that contains microseconds (0.001ms) if the precision allows it. And actually POSIX 1003.1b requires nanosecond precision.
There are also the POSIX `clock_gettime()` family of functions, which let you specify from which clock to get the time from, and `clock_getres()` which let you get the precision of the clocks available. The clock you can pass to those methods are predefined in the manpage and are useful for profiling. `CLOCK_MONOTONIC` being the best to calculate the time between two events in time.

There used to be a time when on Linux all the timers on the system were coupled to the jiffies, it isn't the case today. We have a decoupled clock event subsystem that gets delegated and manages interrupts, and the source device can be swapped without breaking everything. Linux also added a kernel configuration called `CONFIG_HIGH_RES_TIMERS` to allow high resolution time, which is enabled now everywhere.
This lead to the concept of dynamic ticks, having the clock for scheduling ticks at different speed while not affecting the clock source timeline, which can be used to save energy/power.
This furthermore lead to the idea of having tickless system, systems where the timeslice for scheduling is actually controlled by the scheduler instead of follow `HZ`. The `CONFIG_NO_HZ` option in the kernel can be set to allow this. It is also enabled on most desktops today.

```
# CONFIG_NO_HZ_FULL is not set
CONFIG_NO_HZ=y
```

On Linux, all the information about timers and their statistics is propagated to user space in `/proc` for advanced debugging.

For instance, `/proc/timer_list`, gets us a list of the currently configured clocks and running timers. We can use it to check their precisions:

Example output:

```
HRTIMER_MAX_CLOCK_BASES: 2
now at 294115539550 nsecs

cpu: 0
 clock 0:
  .index:      0
  .resolution: 1 nsecs
  .get_time:   ktime_get_real
  .offset:     0 nsecs
active timers:
 clock 1:
  .index:      1
  .resolution: 1 nsecs
```

```
  .get_time:   ktime_get
  .offset:     0 nsecs
[...]
 event_handler:  hrtimer_interrupt
```

We can see that the `.resolution` is `1 nsecs` and that the `event_handler` is `hrtimer_interrupt` instead of `tick_handle_periodic`, which would be for lower resolution timers.

`/proc/timer_stats` is an advance debugging feature that can be enabled via the `CONFIG_TIMER_STATS` kernel configuration and that let us gather statistics about all the timers on the system, you turn it on and off whenever you want. It can tell us which routines in the kernel are using the timers and how frequently they are requesting them. The format is as follows:

```
<count>,  <pid> <command>   <start_func> (<expire_func>)
```

Now let's move to syncing the system time using an external source.

Syncing time with external sources

- Ways to update time
- Why use external sources
- A primer on precision calculation and terms
- Discipline/slewing and smearing
- Leap second propagation
- List of external sources
- NTP
  - Protocol
  - Tools/clients
  - Pool and organization
  - List of public servers
- Other implementations
- Security issues
- PTP

Before starting this section I'd like to point out three ways that time can be updated.
One is called stepping and it consists of making one discontinuous change to time, a sudden instant jump from one value to another that could be backward or forward in time. This happens when something triggers the system time to go to a specific value, external sources of time can possibly do this.
Another is called slewing or sometimes disciplining and it consists of making the clock frequency ticks faster or slower, or to change the value that ticks represent. It means that it's adjusting it gradually over time. This is what we've seen in the previous section with tools such as `adjtimex(8)` for system time.
The last way to change time is actually a category of slewing called smearing or fudging and it consists of changing time by gradually adding parts of a larger chunk of time over a period. This leads to less breaking changes if, for example, we add 10s to our system time but split it across a whole day, it could be 0.1ms every second or even less. However, in fact, it can fit in the slewing category, but we usually talk about smearing when we are forced to add an expected change, such as a leap second.

We also have to keep in mind the time it takes to fetch the value of time from a source: for it to be transmitted and interpreted by our system. It won't be an instant transfer and sometimes may take such a long time that we are forced to take it in consideration when doing our calculations of adjustment. We call this the delay, the time it takes to do the round trip. As we'll see there are many factors to take in consideration too.

So why should we rely on an external source of time, why should we care about having precise time.

At human scale, on my watch I can rotate the hands of the clock a little and

adjust it to whatever someone else has. Nothing horrible is going to happen because of it, right? The majority of people don't need precision higher than a couple of minutes.
However, computers are different, you're not the one monitoring time, and their errors accumulate pretty fast if you forget about them; clocks drift as we've repeatedly said.

So who needs millisecond accuracy or more? Who has a need for precise time? And precise time according to what. Haven't we said in the previous section that our system clocks already have a pretty good calibration mechanism in place.
However, even with all the accuracy we can get, we're still going to drift, no matter what, and we still need to be aware of UTC changes such as leap seconds.

So what does accurate mean, do you need accuracy across your internal network of machines, each not drifting too far away from the other even though not in sync with UTC. Or do you need them all to be in sync with UTC or your time zone.
How much does the software you are running tolerate change in time. Does it expect monotonic time, can it handle jumps, is it an issue if it differs from real UTC time. Do you have to keep in sync with UTC because of the software itself or because of compliance with standards or because otherwise the meaning is lost.

Here's a list of systems that actually require accurate synchronization with UTC:

- Security related software that verify certificates
- Security related software to match timestamp with real life events such as CCTV
- Similarly, intruder detection software
- Similarly, any type of audit logs or timestamping based on real world events
- Similarly, any network monitoring, measurement and control tool
- Radio, Telecommunication, and TV programs
- Any type of real time multimedia synchronization
- Many types of distributed systems
- Money related events, such as stock market
- Aviation traffic control software

We have to add to this list all the machines with nasty system clocks that drift in unexpected ways. Such machines are better off syncing with an external source.

So, in which case should we not synchronize time with an external source then.

- If the accuracy and adjustment that the system clock provides is enough
- If we don't want to worry about managing another daemon on our system
- If the system doesn't have access neither to the internet nor to any physical external source of time.

Now that we know if we need an external source of time or not, let's see how to calculate how precise and stable these sources are.

When using a clock as reference we want to make sure it's stable and precise.

A clock's frequency stability is rated and measured by its ppm or ppb error, part per million errors or part per billion errors. The "part" can be either the number of ticks or the number of seconds that are drifting from the actual value (both can be used interchangeably). The smaller this ppm value is the more stable the clock is.
The reasons why a clock drifts are environmental variations such as temperature, aging of the material, G-force, change in voltage, etc.

What does this mean.

Let's take as an example an HPET clock with a frequency of 10MHz (`10_000_000Hz`) in an environment between -40 to 80 degrees C. Add the fact that the clock manufacturer has specified that it has a stability that varies between -7.5 and +7.5 ppm.

For every 1 million ticks or 1 million seconds, there is a plus or minus 7.5 variation. Over a whole day the clock would drift by 0.648 seconds.

```
(7.5/1M) * 86400 = 0.648s
```

Atomic clocks have tremendously tiny error variations, those can be between 0.0001 and 0.000001 ppb (part per billion). They are drifting by a second every ~300k to 30M years, which confirms what we've explored in the first section of this article.

Temperature plays such a big role in the stability of our clock so to make them more stable we could either lock our machines in a controlled temperature environment or try to come up with a way to automatically compensate for how much the temperature is affecting the clock.
While the first option is possible in data centers, it's not something that is possible for most of us.

However, we could devise an experiment in which we find a formula that calculates how much the temperature affects the clock frequency and slew our clock appropriately.
We could monitor the temperature, feed it, and use it to train the correction mechanism in whatever software or mean we use to handle setting time. Which all comes up to gathering points that fit the data temperature and how much the clock drifts, plotting them on a graph and finding an equation that passes by those points. This is simple math using polynomial interpolation.

Unfortunately, no solution is perfect and this could be overly optimistic. Correlation doesn't equate causation. Still, such mechanism are great to keep the clock stable within a certain temperature range. Some experiments have found that temperature compensation reduces deviation by 3.5 times. Our earlier drift of 0.648s would be reduced to 0.185, or 2.14 ppm instead of 7.5 ppm.

Let's now define some important terms we need in our inventory to understand everything that is coming next about external clocks.

- "Reference clock" or "ordinary clock", this means any machine that can be used to retrieve accurate time, usually in UTC so that it can be used by anyone. Those could range from cesium clock, to GPS, to terrestrial broadcast like radio clocks.

The time from the reference clock will be forwarded from one server to another until it reaches you. Thus, the reliability of the network, and how far you are from it, play a big role in how accurate the value from the reference clock will be.

We're using server that are connected to external time source and external time source themselves interchangeably here unless explicitly mentioned.

- "Delay", a word we've seen before that means the time it takes to do a round trip. It's normally calculated by timestamping on both ends and doing an estimate of the difference in transport and processing.
- "Offset" or "Phase", is the time difference/deviation between the clock on one end and the clock on another end, usually your clock and a reference clock. Phase referring to the oscillation rhythm difference, as in "out of phase".
- "Jitter" or "Dispersion", is the successive time values difference after subsequent requests from remote servers or action. It's a great criterion to measure the stability of the network, how much delay changes, if it varies a lot the network isn't reliable. This term can be used as a measure of stability of any other repeatable action too.
- "clockhopping", the spreading of time, jumping from one server to another, which results in less and less accuracy.
- "Frequency error", this is how much the reference clock or our local clock drifts over time, measured in ppm and ppb, as we've seen before.
- "Stability", the generic term to refer to how much we can trust a clock. It's also a term used in control theory to refer to how far we are from a reaching a stable point (0).
- "Accuracy", also a generic term that means how far apart a machine's time is away from UTC. The typical accuracy on the internet ranges from about 5ms to 100ms, varying with network delays.
- "PPS", or "Pulse Per Second", a method of synchronizing two clocks based on a tick that happens every second.
- "Watchdog timer", is a timer that keeps the time since the last poll or update of time from the external source of time.

- "Fudge", I couldn't put my hand on the precise definition of this term other than that it refers to any special way in which you can configure an external clock.
- "Max Poll" and "Min Poll", throttling parameters which are the maximum and minimum number amount of time that should pass before the remote server allows you to query it again. This is usually expressed in powers of 2, for example 6 means `2^6` or 64 seconds.
- "Stiffness" or "Update Interval" or "Time Constant" (τ tau), how much the clock is allowed to change in a specified amount of time, and the time between the updates. A small time constant (update interval) means a clock that is less stiff and slews quickly. It's usually expressed like the max poll in powers of 2.

When all those values differ a lot, we can't allow an abrupt jump of time on our end, that would disrupt local processes. So what we do is slew time, but that would also mean having a slow calibration.
In an ideal world, all the reference clocks would be the same everywhere, however they aren't. So what should we do if there's a big offset.

First off, if the offset is too big we don't trust it until we have the same offset from multiple time sources.
If it's small enough we go on with our slewing.
If the offset is still big, we have to set the clock anew, step it.

However, on boot we have to sync from the hardware clock, like we've seen before, which might be off, so we have to either slew the system time which can take several hours or make the updates less stiff to quickly reduce the offset (in 5min usually with a less stiff PLL).

Moreover, we can't also believe any remote server or machine as a time source, so we ought to devise a mechanism, a sanity check, to filter which machines we trust and which ones we don't, maybe even combining multiple time sources in a quorum-like fashion.
We can evaluate remote machines for how stable they are by making them pass a statistical filter for their quality.

That also creates a trust issue on boot, so what can be done is to send multiple quick requests to multiple external time source servers to insure their reliability and get an estimate within 10 seconds at boot.

As time goes on, our system clock should become more stable, and we should be requesting the remote servers less frequently.

This is possible through different feedback mechanisms that learn to adjust the system time appropriately. In a way, this is similar to the mechanism that fixes the hardware clock drifting but for the system clock which we haven't tackled before.

Different Unix-like OS and software provide different means of adjusting the system clock according to external time. There are 4 mechanisms or system calls that can be used to implement the adjustment of the system clock.

The first method is through `settimeofday(2)`, which is used to jump to a fixed place in time, to step it. This could be used for very big offsets.

The second method is through `adjtime(2)` which is used to slew the time by changing the frequency of the clock. You pass a delta to the function and if this is positive the clock will be sped up until it gains that delta and if negative the clock will slow down until it has lost this delta. This is intended to be used to make small adjustments to the system time and thus there's a limit to how big the delta can be (plus or minus 2145 seconds).

The third method is through the `hardpps()` function that is internal to the kernel and handles an interrupt service that listens to a constant pulse that happens every second. The RFC 2783 defines how this API should behave, basically syncing the transition between pulses with the system clock.

The fourth, and last method, is through the `ntp_adjtime(2)` function, that is an advanced mechanism to discipline the system clock. It is defined in RFC 1589 called "A Kernel Model for Precision Timekeeping", also going under the name of "kernel clock discipline". Initially created as a better version of `adjtime(2)` that can be called from the software handling external precision time source as it accumulates successive precise corrections (could be in the microsecond range).
This method of adjusting time is based on an algorithm that depends on multiple environmental factors and that can be tweaked as needed. From correcting frequency and offset, to enabling or disabling PPS events processing, to synchronization status, handling leap second, estimating error and frequency change tolerance, and more.
At the core of this kernel clock discipline algorithm lies a concept from the domain of control theory, a closed loop that accumulates successive corrections, an adaptive feedback loop mechanism that tries to minimize network overhead. Today, the algorithm uses two kinds of loops, one is a phase/offset locked loop (PLL), and the other is a frequency locked loop (FLL). We've hinted at those previously when checking the status bit of the `adjtimex -p` and `ntptime` commands.

```
> adjtimex -p


        mode: 0
      offset: -7431812
   frequency: -1677305
    maxerror: 2000
    esterror: 0
      status: 8193
time_constant: 7
   precision: 1
   tolerance: 32768000
        tick: 10000
```

```
     raw time:  1588007066s 608698606us = 1588007066.608698606
```

```
> ntptime


ntp_gettime() returns code 0 (OK)
  time e2518f79.db34a44c  Mon, Apr 27 2020 20:06:01.856, (.856272195),
  maximum error 49500 us, estimated error 0 us, TAI offset 0
ntp_adjtime() returns code 0 (OK)
  modes 0x0 (),
  offset -6172.114 us, frequency -25.594 ppm, interval 1 s,
  maximum error 49500 us, estimated error 0 us,
  status 0x2001 (PLL,NANO),
  time constant 7, precision 0.001 us, tolerance 500 ppm,
```

Phase locked loop and frequency locked loops main difference is their predictor part that will output the value of their feedback loop. They both take as input the timestamp and compare it with local time but what happens afterwards, how they change either the phase/offset or the frequency depends on which one is chosen.

PLL is an offset discipline mode, its predictor is an integral of the offset over past updates and outputs the offset amortized over time in order to avoid setting the clock backward. It adjusts it gradually by small increments or decrements until the offset is gone. The time constant aka update interval is the rate at which it executes this update. The smaller the time constant, the less stiff it is, and the faster it'll converge to an offset of 0 (stability in control theory).

FLL is a frequency based discipline mode, its predictor takes the offset and divides it by the time since the last update and adjusts the clock frequency such that at the next update the offset will be as small as possible.

In the most recent software, the two modes are used together and mixed. They are weighted according to the polling intervals, when it is bellow Allan intercepts, which is 2048s (this can be changed), then a phased-lock loop is used with more weight, and whenever the polling interval is higher, then a frequency-locked loop is heavier.

When not fetching the time from another machine that is connected to a reference clock but having it connected directly to us, we'll require some hardware driver to interface with it.
These physical source of time can implement their own clock discipline algorithm and synchronization protocol and thus we have to adapt appropriately. If they do provide such mechanism via their drivers, we let the external clock be in control and determine which discipline should be used, normally they will themselves call `ntp_adjtime(2)` with the parameters they know about. If it fails, we can fallback to the previous way of adjusting time. Keep in mind that when there is an external clock taking care of system time adjustment, no other software can be aware of the error statistics and parameters it maintains.

Before moving on to what those devices could possibly be, let's have a small note on leap second smearing.

There are two main ways to handle a leap second, we could either step, that is stopping the clock completely for a second (repeating a second) or skipping an entire second, or we could either slew the clock using the smooth kernel discipline we've seen.
This is what leap smearing is about, it's a standard pushed by Google to "smear", that is slew that second by running the clock 0.0014% slower over 24h, from noon before the leap second to noon after the leap second. This is such that the slewing happens linearly around the leap second.
The change for a small smear is about 11.6 ppm.

However, keep in mind that such standard only has weight if everyone adhere to it, otherwise during the leap second event multiple servers will have different time and the dispersion will be bigger. That is why we shouldn't mix smearing with non-smearing time servers.

We've said we could connect to physical precise time source to become a reference clock, so what can those be? We'll give as example the two most popular ones.

In the first category we have terrestrial broadcasts, radio stations that broadcast time.
The most well known are the CHU in Canada, WWV in Colorado, USA, and WWVH in Hawai, USA.

CHU broadcasts at 3.33MHz, 7.85MHz, 14.67MHz since 1923, and WWV broadcasts on 2.5MHz, 5MHz, 10MHz, 15MHz, and 20MHz since 1945. They both get their time from other reliable sources such as atomic clocks and GPS.

As they are radio broadcasts, you need a radio receiver and a way to analyze the audio to be able to synchronize with them.

What they broadcast is some repetitive beep to sync on pulse per second and minute, some binary coded decimal, and literally someone talking from time to time to say in English, or French for the Canadian version, what time it currently is in UTC hour and minute. So they alternate beeps, ticks, and voice announcements.
You can give those a listen by searching their names on Wikipedia, Youtube, or actually turning on your radio on the right frequency.

Furthermore, there are also telephone numbers that you can call to get the time, similarly to the radio. One of them is provided by the same organization as the CHU, the NRC, National Research Council of Canada.

In the second category we have GPS, the Global Positioning System.
And let's be more explicit here, we're talking about the American NAVSTAR GPS that is composed of multiple satellites at 20K km orbit, always having 4 satellites visible from any point on Earth.

To sync time with a GPS you need a GPS receiver, some of those also come with a pulse per second feature for accuracy's sake. The receiver catches the civilian band frequency that the GPS continuously broadcasts and decodes the signal to get the messages in it. This message contains a multitude of information, but we're only interested in what is time related.

The GPS satellites include atomic clocks that are accurate to the nanosecond. However, we're loosing a bit of accuracy because of the delay between us and the satellite. So you'd think that since they have atomic clocks they would follow TAI (International Atomic Time), however they follow their own special time format called GPST, the Global Positioning System Time.
The GPST is similar to TAI, as in it is constant and unaffected by the rotation of the Earth, but the difference is that it's epoch, it's 0h, was set on 6 January 1980. Consequentially, it includes all the leap seconds before that date but none of the ones afterwards so currently it differs from TAI by 19 seconds, and from UTC even more. That is why newer GPS units include in their messages a field of 8 bits that contain the time difference between GPST and UTC, the number of leap seconds it missed, so that you can easily get back UTC.

The format time that GPS store and broadcast doesn't use the year/month/day Gregorian calendar but express it as a week number and a seconds-into-week number. The week is a 10-bit field and thus rotates every 1024 weeks, which is approximately 19.6 years. The first rollover happened on August 21, 1999 and the second one on April 6, 2019.
So to determine the Gregorian date you need to know in which GPS epoch you are. Future GPS will update the standard to use a 13-bit field instead, rolling over every 157 years.

This phenomenon of week rollover has been deemed the "Y2K" of GPS because many device drivers didn't anticipate it and had hard-coded the GPS epoch.
A solution to this would be to derive the GPS epoch from the leap second data broadcasted by the GPS and a leap second table. Weirdly, a GPS vendor, has a patent on a similar technique so you can't use it exactly the same way. Sometimes shipped software is shipped software and nobody is going to touch nor update it, beware.

Apart from NAVSTAR, there are plenty of other space agencies that have launched GPS technology.

- Beidou, People's Republic of China's
- Galileo, European Union and other partner countries
- GLONASS, Russia, peculiarly, its UTC is in Russia (BIH timezone)
- NavIC, Indian Space Research Organisation
- Michibiki, regional navigation system receivable in the Asia-Oceania regions

A last nota-bene, your position is derived from how far you are, the delay, from 4 satellites and calculating the intersection.

To link this to the previous ideas, if you have a driver that supports those external clocks hardware device receiver, it should implement the `ntp_adjtime(2)` or a custom discipline to take care of adjusting time itself. Be sure to check the list of drivers available for your solution.

Let's proceed from the abstract talk to the concrete: which protocols and standards can be used to implement time synchronization with external sources of time.

The most trivial protocol is the Time Protocol defined in rfc 868. It's a simple client-server protocol where the server when receiving a request directly replies the time in seconds since midnight 1st 1900 GMT as a 32 bit binary number. The protocol runs on UDP and TCP on port 37, as `/etc/services` shows:

```
time                37/tcp
time                37/udp
```

Because it's based on a 32 bit value, it's going to rollover at some point in 2036 which will deprecate it easily unless the value is upgraded to 64 bits.

While it's simple, it doesn't take into consideration leap seconds, delays, is only precise to the second, and disregards all the stuffs about time we've previously mentioned.

You can give the time protocol a try by testing is using the `rdate` utility.

```
rdate - get the time via the network

rdate connects to an RFC 868 time server over a TCP/IP network,
    printing
the returned time and/or setting the system clock.
```

The evolution of the time protocol is the Network Time Protocol, or NTP. It takes into consideration multiple things the other did not including: leap seconds, broadcasting mechanism, active/passive modes, security and digest, a hierarchical level of accuracy, polling mechanisms, more precision, versioning, considerations of delays, categorizing known clocks by reference identification, and much more.

The current protocol stands at version 4, NTPv4, which is documented in rfc 5905 but has additional addendum for extensions. It is backward compatible with its previous version, NTPv3, in rfc 1305.

NTP runs on UDP and TCP on port 123, as `/etc/services` shows:

```
ntp                123/tcp
ntp                123/udp
```

The timestamps that NTP sends and receives rely on UTC time, the timezone information is kept for local machines to decide. Additionally, NTP warns of any impending leap second adjustment.

Thus, in theory, all NTP servers should store the same UTC time up to a certain precision.

When an NTP client is running we have to choose what to do with the hardware clock, do we sync it with system time. Many implementations either save the drift to a file so that it can be used on the next boot and/or rely on the kernel "11 minute mode" we talked about earlier. Moreover, if a network connection is available at boot time there's the possibility of using NTP right away. Like this we remove the burden of relying on RTC when the machine is offline.

NTP uses a hierarchy, a semi-layered division, to classify clocks that are available. It calls them strata.

The stratum, singular, is a measure of the synchronization distance to a reference clock. Remember a reference clock is an actual hardware that can be used to get precise time, like a GPS. The stratum is the number of servers we need to pass through to reach such reference clock. Unlike jitter (dispersion) and delay, the stratum is a static measure, you don't get further away from a reference clock.
So it's preferable to use the closest (network distance) and lowest stratum possible NTP server.

The reference clock itself, the timekeeping device, is considered stratum 0 and the closest servers connected to it are at stratum 1. Thus, a server synchronized to a stratum `n` server will itself be considered stratum `n+1`.
The upper limit for stratum is 15, in theory, above this the dispersion may grow too much for it to be reliable, though, in practice it doesn't go above 5.

The stratum hierarchy helps in spreading the load and avoid cyclical clock dependencies as it's now in the shape of a tree. That means a small number of servers give time to a large number of clients, which in turn could be servers to others. That implies low stratum servers, such as stratum 1 servers, should be highly available and well maintained to support the rest of the hierarchy.

In addition, the NTP contains in its message a reference identifier, `refId`, which denotes which reference clock is used at stratum 0 on this path. So you can know you're getting your time from which source.

Let's also mention that NTP can be deployed locally, in a LAN. It's possible to create your own hierarchy by acquiring a timekeeping device, such as a GPS, to avoid network delays and get a better precision.

NTP is not limited to the usual client/server architecture, it includes horizontal peering mode and a broadcasting mechanism.

Horizontal peering is when multiple servers are coupled together in a group to synchronize time more accurately.

The broadcasting mode works by having a server sends the time to a broadcast address and have clients listen for NTP packets sent to that address. This

mode is useful for leap second propagation instead of having it sent only when the client connects.

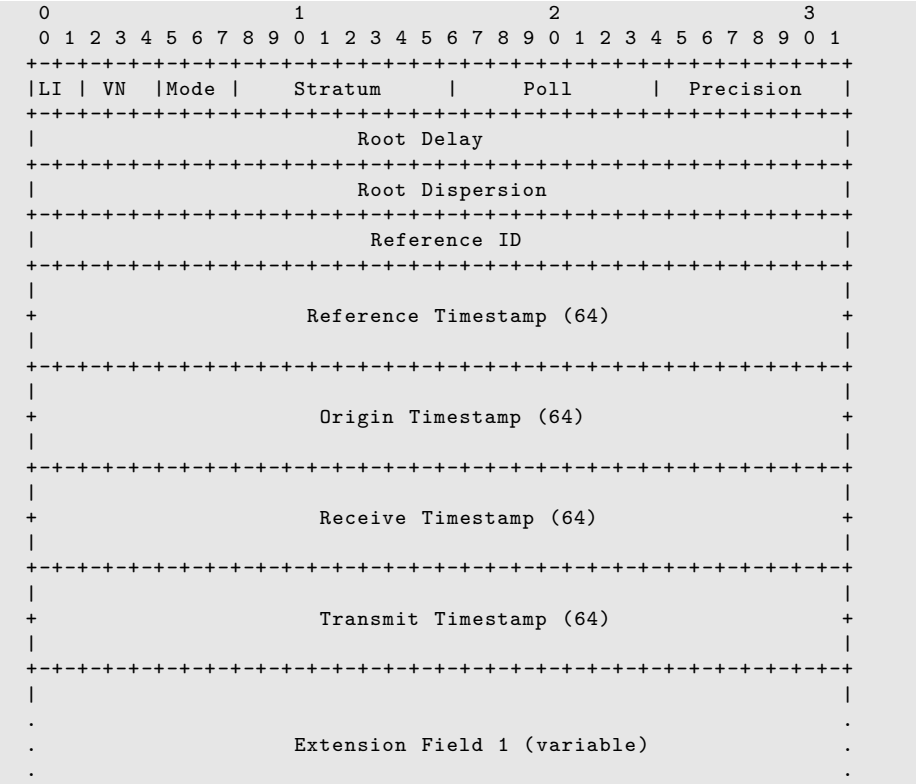
On that note, on the day of a leap second event, the leap second could be propagated either from a configuration file, a reference clock, or another NTP server. What then happens, how the leap second is applied, depends on the implementation. It could be a stop or skip mechanism or a leap second smearing. It is applied at the level of the server.

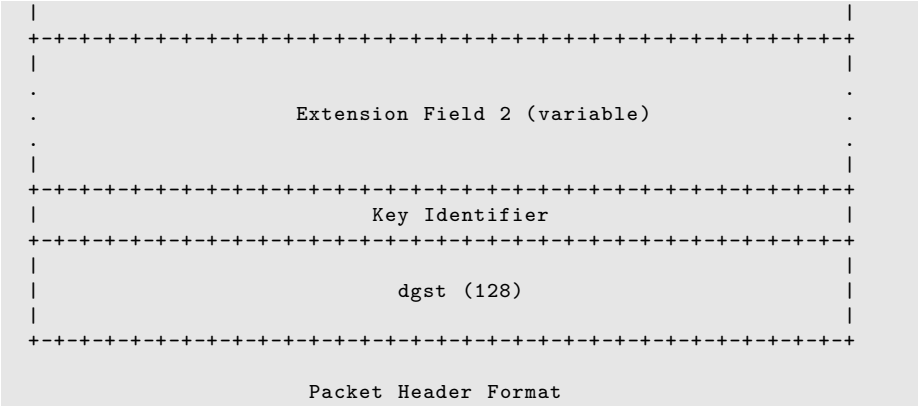
What does an NTP message look like.

In the early days of the NTP, the timestamp in the message used to have the same issue as the Time Protocol, a single 32 bit value, thus having rollover issue.

NTPv4 now uses a 128-bit date format that is split into 2 main parts, one is 64 bits for the seconds and the other 64 bits for fractional seconds.
The seconds part is again split into two others, the most significant 32 bits is the current era number (number of rollovers), and the least significant bits the number of seconds in this era. That removes all ambiguity and the 64 bit value for the fraction is enough for the time it takes a photon to pass an electron at the speed of light, so very precise.

An NTP message looks like this:





Packet Header Format

As you can see NTP is much more advanced than the time protocol. For example, a minimal request would need the version and mode filled, client mode being 3, or 011 in binary. The usual unencrypted messages are 90 bytes long (76 bytes at the IP layer). A broadcast happens every 64 seconds while a client/server architecture requires 2 packets per transaction, initially once per minute and gets reduced to once every 17 minutes in normal conditions.

With such protocol, though requiring minimal bandwidth, but with an insane amount of clients, there needs to be throttling system. The polling interval of a client depends on many factors, including its current precision and the maximum and minimum polling interval allowed by the servers.

NTP servers are viewed as a public utility of sorts and thus need help from the public, especially the people that are knowledgeable and have access to static public IP addresses. The pool of public NTP servers needs to keep growing to serve the increase in clients.

You can view a list of public NTP servers here:

- https://gist.github.com/mutin-sa/eea1c396b1e610a2da1e5550d94b0453
- http://support.ntp.org/bin/view/Servers/StratumTwoTimeServers

But why rely on publicly available NTP servers instead of building our own NTP server hierarchy on LAN, haven't we said this would offer more precision.

Not only would it offer more precision because of the stability of the bandwidth and its network distance, it would also mean it's under our control and thus not throttled, thus more available. That would also mean more security and trust as you could put the NTP server in a local demilitarized zone (DMZ) which is often required to pass security accreditations.
However, to do all this requires cost: the cost to acquire and maintain a time-keeping device such as a GPS, the cost of the setup fees, the cost of additional equipments, the cost of training the team. It's all a question of money.

But let's say you want to deploy your own NTP server, what's available out

there for you to use, what are the implementations.

The NTP reference implementation of the protocol, the canonical open source implementation, is called ntpd. It has continuously been developed and maintained for over 25 years.

It comes with sensible default configuration to fetch time from a pool of NTP servers on the internet. Most Unix-like distros have packages that are easy to set up.

What you can configure range from the location of the drift file to control local clock, to the location of the leap second and how it's applied, to the clock discipline related configurations like jitter rate, to security options, to log locations, and to hardware driver related configurations like when you are setting up a stratum 1 server.

The `ntpq` utility allows to manage the NTP server, be it local or remote, and query its status and configurations. Similar to openssl, it has an interactive mode and a command line arguments mode.
For instance, the `ntpq -p` output is quite interesting.

Example output:

```
     remote           refid  st t when poll reach  delay   offset
         jitter
==============================================================================
*time-A.timefreq .ACTS.          1 u  152 1024  377  43.527 -11.093
    3.982
+clock.isc.org   127.0.0.1       2 u  230 1024  377  67.958  -7.729
    0.071
 time-a.nist.gov .ACTS.          1 u  323 1024  377  58.705 994.866
     999.084
```

It displays the server name in the first column along with its state, a `+` meaning it's a candidate server and `*` meaning it's a peer. The `refid` column is the reference identifier we've mentioned. The `st` column is the stratum level of the server. The `when` column shows the number of seconds since we've last polled that server. The `poll` column is the number of seconds we have to wait between polls. The `reach` column is an octal bitmap of the result of the last 8 polls (377 means success for the last 8 polls). The `delay` column shows the number of milliseconds for the round trip, which we've said varies according to network stability and distance. The `offset` column is another term we've seen, it is the difference in milliseconds between your clock and the host. Finally, the `jitter` or `disp` column is the dispersion in the milliseconds, the difference between different queries to the same server, a measure of stability.

Another tool to test remote NTP servers is `ntpdate`. It can be used to initiate syncing local clock but when used with the `-d` option it can query an NTP server without changing system time.

Here's a trace:

```
> ntpdate -d time-b.nist.gov
30 Apr 19:04:27 ntpdate[72016]: ntpdate 4.2.8p14@1.3728-o Wed Mar 18
    13:44:46 UTC 2020 (1)
Looking for host time-b.nist.gov and service ntp
129.6.15.29 reversed to time-b-g.nist.gov
host found : time-b-g.nist.gov
transmit(129.6.15.29)
receive(129.6.15.29)
transmit(129.6.15.29)
receive(129.6.15.29)
transmit(129.6.15.29)
receive(129.6.15.29)
transmit(129.6.15.29)
receive(129.6.15.29)

server 129.6.15.29, port 123
stratum 1, precision -29, leap 00, trust 000
refid [NIST], root delay 0.000244, root dispersion 0.000488
reference time:      e2557598.00000000  Thu, Apr 30 2020 19:04:40.000
originate timestamp: e255759a.e0bb5ccd  Thu, Apr 30 2020 19:04:42.877
transmit timestamp:  e255759a.c79bb883  Thu, Apr 30 2020 19:04:42.779
filter delay:  0.20827    0.23691    0.20261    0.21184
               ----       ----       ----       ----
filter offset: +0.006270  +0.003106  +0.010331  +0.005027
               ----       ----       ----       ----
delay 0.20261, dispersion 0.00426, offset +0.010331

30 Apr 19:04:42 ntpdate[72016]: adjust time server 129.6.15.29 offset
    +0.010331 sec
```

By now you should be able to read the values and understand what they mean.

There are many other implementations of NTP servers other than the canonical ntpd. You can find multiple comparison tables online that show the differences between them. Here's a few things that often get mentioned:

- The type of license
- The programming language used
- The size of the program
- How well the codebase is cleaned up and maintained
- The time sources supported and their numbers
- What reference clocks drivers are supported
- The NTP modes it supports
- Which protocol versions it has implemented
- If you can create clusters/pools
- The way the clock discipline works and can be configured
- If it supports temperature compensation
- How it handles leap seconds correction and if it can be configurable
- Security and authentication mechanisms
- If it has rate limiting functionalities
- The way it timestamps, is it kernel based or hardware based
- If it has a way to manage the RTC or not
- Monitoring related functionality

The canonical implementation, ntpd, fully supports the specs as it's the reference implementation, has been ported to the biggest number of operating systems, has the largest number of drivers, and is probably the most stable.

Chrony is another software that implements NTPv4. It has been written from scratch and is known to be well maintained and secure.
Chrony's biggest selling point is that it works remarkably well in environments where the external time source isn't regularly available, a machine that is frequently disconnected from the internet. Though this begs the question of why use Chrony instead of relying on the built-in OS mechanisms we've seen in the previous section. You can even explicitly tell the daemon that you are about to go offline. The other biggest advantage of Chrony is that after an audit of multiple NTP implementation, it came out as the most secure between them. Chrony is also thought to be easier to configure than ntpd.
Unfortunately, Chrony lacks in the driver, OS, and specifications support department.

systemd's timesyncd is a network time protocol daemon that implements an SNTP client, Simple Network Time Protocol, defined in RFC 4330. SNTP is a simplified version of NTP that uses the same network packet format but with a different way to deal with synchronization. It doesn't bother with the full NTP complexity and only focuses on querying time and synchronizing the system clock.
Thus, there's no hardware driver support for systemd's timesyncd, but it's very simple.

The advantage of having the time synced using a service manager is that it can be hooked to automatically start whenever the network is operational, whenever there's connectivity.

The status of the clock can be requested using `timedatectl status`.

Example of output:

```
> timedatectl status
               Local time: Thu 2020-04-30 19:26:56 EEST
           Universal time: Thu 2020-04-30 16:26:56 UTC
                 RTC time: Thu 2020-04-30 16:26:56
                Time zone: Asia/Beirut (EEST, +0300)
System clock synchronized: no
              NTP service: inactive
          RTC in local TZ: no
```

The NTP synchronization status can also be checked using `timedatectl timesync-status`.

Example output:

```
> timedatectl timesync-status
       Server: 162.159.200.1 (1.arch.pool.ntp.org)
Poll interval: 34min 8s (min: 32s; max 34min 8s)
         Leap: normal
```

```
      Version: 4
      Stratum: 3
    Reference: A960804
    Precision: 1us (-25)
Root distance: 32.142ms (max: 5s)
       Offset: -9.034ms
        Delay: 49.905ms
       Jitter: 21.188ms
 Packet count: 418
    Frequency: -27.105ppm
```

The client configuration can be found in `/etc/systemd/timesyncd.conf` with a format defined in the `timesyncd.conf(5)` manpage and `systemd-timesyncd.service`.

```
> timedatectl show-timesync --all
LinkNTPServers=
SystemNTPServers=
FallbackNTPServers=0.arch.pool.ntp.org 1.arch.pool.ntp.org
    2.arch.pool.ntp.org 3.arch.pool.ntp.org
ServerName=1.arch.pool.ntp.org
ServerAddress=162.159.200.1
RootDistanceMaxUSec=5s
PollIntervalMinUSec=32s
PollIntervalMaxUSec=34min 8s
PollIntervalUSec=34min 8s
NTPMessage={ Leap=0, Version=4, Mode=4, Stratum=3, Precision=-25,
    RootDelay=62.911ms, RootDispersion=366us, Reference=A960804,
    OriginateTimestamp=Mon 2020-04-20 17:42:43 EEST,
    ReceiveTimestamp=Mon 2020-04-20 17:42:43 EEST,
    TransmitTimestamp=Mon 2020-04-20 17:42:43 EEST,
    DestinationTimestamp=Mon 2020-04-20 17:42:43 EEST, Ignored=no
    PacketCount=372, Jitter=15.678ms }
Frequency=-1003040
```

BusyBox also offers a compact built-in SNTP implementation. But beware that no driftfile is used.

clockspeed by D. J. Bernstein is an even simpler approach to SNTP that uses the TSC register to adjust the ticking speed.

Another protocol called Berkeley Algorithm works by polling the time from all the machines on a network and taking the average, all the other machines then syncs with this time.

Yet another interesting implementation is HTP, the HTTP Time protocol. HTP relies on the Date header of HTTP, defined in the HTTP/1.1 RFC 2616. It uses statistical analysis to arrive at the most accurate time possible. So if you can access a webpage then you can sync time. Though this protocol isn't the most accurate nor the most secure.

Hence, let's see what's up with security and NTP. We know that anything that is on the network should be secure and trusted.

If, for example, an attacker effectuate a man-in-the-middle attack they would be to be able to change the time source for your machine. The security implications

would be that certificates and signatures that shouldn't be trusted because they expired would be. It would tamper and mess logs too.

That's one reason why browsers today show errors whenever the system clock is out of sync.

The NTP specifications have been there for so long that we've had plenty of years to find security issues and fix them, the reference implementation being the testing ground for them. The codebase is constantly audited.
For example, the first version of NTP were all clear text and thus had no protection against MITM attacks. The specifications added the need for authentication, checksums, and even encryptions via symmetric and public/private keys in the latest addendum.

There has been a move by the IETF to create this encryption overlay called Network Time Security (NTS). CloudFlare currently implements it but not many others.

Another project called NTPSec forks the ntpd source, tries to remove complexity and clean the code. Finding vulnerabilities in it.

Audits are important, as we've said, Chrony came out as the most secure between multiple NTP implementations.

I quote:

> A 2017 security audit of three NTP implementations, conducted on behalf of the Linux Foundation's Core Infrastructure Initiative, by the security firm Cure53 suggested that both NTP reference implementation and NTPsec were more problematic than Chrony from a security standpoint.

On the other hand, there are other types of things we need to care about. We've discussed the polling issues and the lack of public NTP servers in the pool, and so it's important for them to be able to withstand heavy loads. One attack relies on computationally expensive operations to take down a public NTP server, a denial of service (DDoS) called the Kiss O'Death Packet. In this attack the client sends a very small query that gets amplified to a huge output content and overloads the server. This is similar to DNS amplification attacks.

Another security issue related to how much we rely on NTP as a public service, is about how some IoT devices have been found to hard-code the address of NTP servers. These kinds of assumptions are dangerous.

The last thing I want to address in this section is another protocol for syncing system time with an external time source called PTP, the Precision Time Protocol.

PTP is used for when NTP doesn't provide enough precision, for critical measurement and control systems such as financial transactions or mobile phone

tower transmissions. It is specially crafted for the local network scenario where you have a machine that has a reference clock device connected to it.

PTP was originally defined by the IEEE in 2002 in the IEEE 1588-2002 standard, officially entitled "Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", then reviewed in IEEE 1588-2008, PTPv2, and again reviewed in 2019 in IEEE 1588-2019.

PTP is similar to NTP in the way that it synchronizes time between machines but the difference is that it adds accurate network latency information using hardware timestamping.
Hardware assisted timestamp is done at the very lowest levels of the network stack such as the MAC layer, the ethernet transceiver, right before sending the packet. The clock can also be associated with network equipments. When the message passes through devices, as it traverses them, the timestamp is updated by them accurately. It is all helped with kernel features for PTP such as the socket option SO_TIMESTAMPING for packet timestamping.
So the timestamp offset is accurate, the delay precise and predictable, and with low latency, that's why PTP can achieve less than microsecond accuracy.

PTP runs on port 319 and 320 using both TCP and UDP for different scenarios.

It uses the same epoch as the Unix time however, while Unix time is based on UTC and is subject to leap seconds, PTP is based on the International Atomic Time (TAI).

An ordinary clock, in NTP parlance a reference clock, is the source of time in PTP that distributes it to a grandmaster, which can then again relay it to boundary clocks that others sync with. It is an architecture that is integrated with the network segmentation.

Within a group the system is automatically able to decide on who to elect as a master clock, which is the clock which is deemed the most accurate.

Let's end by saying that when push comes to shove, you can always buy a machine that comes with everything integrated instead of setting it up yourself.

In the next section we'll see systems that rely on precise time in special ways.

What depends on time

- Consequences of bad system clock
- Distributed systems
- RTOS and scheduling

We've hinted and mentioned some examples of what could happen if there is misbehavior in system time. What else could happen.

We rely on system time for anything related to communication with other humans, it creates the context of what is happening. It is also useful for sampling, data collecting for statistical analysis.

With an out-of-whack system time, logs will be out of synchronization and events will be hard to debug. It will be hard to correlate things with real life.
Database queries that rely on the `now()` to get the current time and date will also write the wrong values.
Many backup scripts will be messed up because they don't expect time to move backward or weirdly.
Similarly, cron jobs that you would have expected to start at a specific time may start at another that isn't appropriate.
The make utility also relies on the timestamp to know which files need to be recompiled. Mishaps in system time may lead it to always recompile the same files.
Many concepts in security such as certificate verification, one time code, and protocols for authentication rely on system time to be synchronized.

Apart from the human perception issues there are the countless overflows and rollovers we've mentioned. All of them being issues related to the size of the data structure that isn't enough to store what is needed, leading to either a rollover or unpredictable behavior by overflowing.

There are the problems related to time zones and daylight saving events. That can happen during the transition when computers repeat an event for one more hour or one hour less than the expected duration. This is tricky when those computers handle machineries and medical devices, it could harm lives or drive businesses to the ground.
Those can also lead to miscommunication between places that have different time zones or don't apply the same daylight saving time.
Programs that are not anticipating changes may need upgrade, things like email programs and calendars.

To resolve most of those it's better not to use time zones and DST in these cases but to rely on UTC and leave civil time for display only. But as you know, there's still the issue of leap seconds.

Some domains need more synchronization than others, let's discuss distributed systems and real-time operating systems as examples.

A distributed computer system consists of multiple software pieces running on different computers but that still try to act as a single system.

In a distributed system, it is important to keep clock synchronization, to ensure all the computers have the same notion of time so that the execution runs on the same timeline.
What we truly care about is event ordering, to know what happens before what, cause and consequences.

In general there are two ways to keep things in order, you either use wall clocks, which are all the timekeeping devices we've been mentioning thus far, or you use logical clocks which are simple monotonous counters all going in a single direction.
Example of logical clocks include Lamport clocks and vector clocks.

The complication in distributed systems is that system clocks on different machines will eventually drift and so it's hard to keep a strict ordering. That's not what's expected of a consistent monotonic clock.
That is why logical clocks are favored as they resolve the chain of events and conflicts.

However, Logical clocks are not always an option. Another one could be to have a strong coordinator for timestamp in the middle where messages always pass through it. But that solution adds a bottleneck to the architecture and limits the availability of the system.
Yet again you could go back to wall clocks but rely on atomic time along with an NTP server, it is not a perfect solution but it avoids time zone, leap seconds, and daylight saving time.

In distributed systems, virtual machines are often used, so a way to keep them in sync is to sync them with their host.

The other domain to explore is real-time operating systems, systems that are mission critical and require accuracy.

Real-time OS, aka RTOS, are similar to general-purpose OS (GPOS) in that they are responsible to managing computer resources and hosting applications, but they differ in the way they are made to have precise timing and high reliability. They are especially useful in machinery environment and embedded systems.

RTOSs are deterministic by design, they are made to meet deadlines associated with external events, they are made to be responsive.
Its jitters, the measure of error in the timing of subsequent tasks, should be extremely low, tasks should execute with almost no delay or a predictable one.
If a system doesn't meet the maximum time allocated to perform a critical operation, if it doesn't fulfill the timing constraint to act on an event, it isn't considered real-time.

Depending on the type of events it can guarantee, real-time OS are separated

into two categories. If it is mandatory that it should be truly deterministic and that a not adhering means a system failure, we categorize it as a hard real-time OS. If the system can only guarantee a certain set of events to happen in real-time and that not adhering won't lead to catastrophic events, we categorize it as a soft real-time OS.

So programs running on real-time OS should run with consistent timing, but we don't stop at this, programmers should have full control on how these tasks are prioritized and be able to check for deadlines. They should be able to dictate how the scheduling takes place and expect it to be directly reflected.

The priority given to task are a parameter that is part of the scheduling algorithm an OS uses to choose which task should run next. It schedules tasks based on interrupts. The time to handle this interrupt routine is variable in general-purpose OS but in real-time OS its latency is bounded.
This is the same time interrupt handler we've seen before, the one that ticks at specific intervals, allows the OS to schedule tasks, increment system time, call timer routines, do periodic tasks, etc. This is where the constraint is applied.

The scheduling is strict, no low priority task can run before a high priority one. Real-time OSs use rate monotonic, earlier deadline first, preemptive scheduling algorithms while general-purpose OSs use completely fair scheduling, round robin, rotary inventor fair scheduling, etc.
Pre-emptive scheduling differs from co-operative scheduling algorithm in that with co-operative scheduling we trust the task to explicitly relinquish control once it's done, while with pre-emptive scheduling we can force to suspend its execution and replace it by another. This all allows a RTOS to respond quickly to real-time events.

There are many examples of real-time OS such as VxWorks, QNX, RtLinux, FreeRTOS, RiotOS, and more. Some are open source, some are POSIX compliants, some are hard real-time, some are soft real-time.

On Linux specifically, to enable real-time you have to enable the `PREEMPT_RT` scheduling patch. Though, it's arguably soft real-time as it's not mathematically provable real-time.

Another real-time Linux related project is the ELISA project, the Enabling Linux in Safety Application project, which aims at creating a standard to build mission critical Linux systems.

I quote:

> Founding members of ELISA include Arm, BMW Car IT GmbH, KUKA, Linutronix, and Toyota. To be trusted, safety-critical systems must meet functional safety objectives for the overall safety of the system, including how it responds to actions such as user errors, hardware failures, and environmental changes. Companies must demonstrate that their software meets strict demands for reliability, quality assurance, risk management, development process,

> and documentation. Because there is no clear method for certifying Linux, it can be difficult for a company to demonstrate that their Linux-based system meets these safety objectives.

That just shows how much time in computers affects real humans lives.

Conclusion

For machines time may need accuracy but for humans it is definitely subjective. We feel time as our emotions are swayed by life. This certainly matters.

We possess clocks within us, biological clocks in which sun time and memories play an integral part in.

For example in emergencies and dangerous situations, time seems to slow down. And as we grow older we perceive time to move faster. It's interesting to see time from a human perspective.

We live in such an interconnected world that greeting people in the morning on the internet has lead to the Universal Greeting Time. Time in this interconnected world has also led to marketing campaigns and funny new standards such as the Beats.

I've written in depth about "internet time perception" in this other article, https://venam.nixers.net/blog/psychology/2019/03/01/internet-time.html.

I hope this colossal article has cleared the topic of time, and time on Unix. It may not be as concise and scientifically true in some places but still convey an approximate overview to readers. Please reach out to me for corrections or comments.

Cheers!

Bibliography

Helmenstine, Anne Marie. "What Is Time? Here's a Simple Explanation." *ThoughtCo*, https://www.thoughtco.com/what-is-time-4156799.

"Physics of Time." *Exactly What Is Time?*, http://www.exactlywhatistime.com/physics-of-time/.

Falk, Dan. *Arrows of Time*, Quanta Magazine, https://www.quantamagazine.org/what-is-time-a-history-of-physics-biology-clocks-and-culture-20200504/.

"Year." *Wikipedia*, Wikimedia Foundation, 7 May 2021, https://en.wikipedia.org/wiki/Year.

"Second." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Second.

"Leap Year." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Leap_year.

"Solar Time." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Solar_time.

"Clock Drift." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Clock_drift.

"Civil Time." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Civil_time.

"Time Zone." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Time_zone.

"Prime Meridian." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Prime_meridian.

"Standard Time." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Standard_time.

"Greenwich Mean Time." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Greenwich_Mean_Time.

"Universal Time." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Universal_Time.

"Coordinated Universal Time." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Coordinated_Universal_Time.

"Daylight Saving Time." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Daylight_saving_time.

"Time." *Equation of Time Calculations*, https://www.space.fm/astronomy/earthmoonsun/eotcalculations.html.

"Time." *Apparent and Mean Sun*, https://www.space.fm/astronomy/earthmoonsun/apparentmeansun.html.

"Time." *Time Zones*, https://www.space.fm/astronomy/earthmoonsun/timezones.html.

"International Atomic Time." *Timeanddate.com*, https://www.timeanddate.com/time/international-atomic-time.html.

"What Is Local Mean Time?" *Timeanddate.com*, https://www.timeanddate.com/time/local-mean-time.html.

"European Union Ready to Scrap DST." *Timeanddate.com*, https://www.timeanddate.com/news/time/eu-scraps-dst.html.

Eggert. "Eggert/Tz." *GitHub*, 9 Jan. 2021, https://github.com/eggert/tz/blob/master/leap-seconds.list.

"Normal Time and Daylight Saving Time." *Blog*, https://www.uninformativ.de/blog/postings/2018-09-15/0/POSTING-en.html.

"Tony Finch's Blog." *Fanf*, https://fanf.dreamwidth.org/133823.html.

"UTC, TAI, and UNIX Time." *Cr.yp.to*, https://cr.yp.to/proto/utctai.html.

"Linux Date Command Help and Examples." *Help and Examples*, 4 May 2019, https://www.computerhope.com/unix/udate.htm#01.

"Locale." *Locale - ArchWiki*, https://wiki.archlinux.org/index.php/locale.

LC_TIME Category for the Locale Definition Source File Format, https://www.ibm.com/support/knowledgecenter/ssw_aix_71/filesreference/LC_TIME.html.

"Zoneinfo." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, https://en.wikipedia.org/wiki/Zoneinfo.

Dhobi, Rahul. "Timezone Setting in Linux." *Unix & Linux Stack Exchange*, https://unix.stackexchange.com/questions/110522/timezone-setting-in-linux#110529.

Feel. "TimeZone Explained." *Feel*, 19 Nov. 2019, https://www.jusfeel.com/2018/10/28/TimeZone-Explained/.

Howto: Create Your Own Time Zone, https://joeyh.name/blog/entry/howto_create_your_own_time_zone/.

"How Often Is `/Usr/Share/Zoneinfo` Updated?" *Unix & Linux Stack Exchange*, https://unix.stackexchange.com/questions/314424/how-often-is-usr-share-zoneinfo-updated.

"Unix Time." *Wikipedia*, Wikimedia Foundation, 2 May 2021, https://en.wikipedia.org/wiki/Unix_time.

"C Date and Time Functions." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/C_date_and_time_functions.

"time_t." *Cppreference.com*, https://en.cppreference.com/w/c/chrono/time_t.

"Falsehoods Programmers Believe about Unix Time." *Falsehoods Programmers Believe about Unix Time*, https://alexwlchan.net/2019/05/falsehoods-programmers-believe-about-unix-time/.

"Year 2038 Problem." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/Year_2038_problem.

"Uptime." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/Uptime.

"Brendan Gregg's Blog." *Linux Load Averages: Solving the Mystery*, https://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html.

"E.2.28. /Proc/Uptime Red Hat Enterprise Linux 6." *Red Hat Customer Portal*, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/s2-proc-uptime.

"Time (Unix)." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/Time_%28Unix%29.

Time(1) - *Linux Man Page*, https://linux.die.net/man/1/time.

"Timestamp." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/Timestamp.

Lokhorst, Tom. "Blog Post: Why 'Always Use UTC' Is Bad Advice." *Q42 Engineering*, Q42 Engineering, 16 Feb. 2021, https://engineering.q42.nl/why-always-use-utc-is-bad-advice/.

"The Worst Server Setup Mistake You Can Make." *Yeller*, https://yellerapp.com/posts/2015-01-12-the-worst-server-setup-you-can-make.html.

"Atime, Ctime and Mtime in Unix Filesystems." *Unix Tutorial*, 11 Apr. 2008, https://www.unixtutorial.org/atime-ctime-mtime-in-unix-filesystems/.

"Mtime Comparison Considered Harmful." *Apenwarr*, https://apenwarr.ca/log/20181113.

"Crontab Guru." *Crontab.guru - The Cron Schedule Expression Editor*, https://crontab.guru/.

Orth, Alan. "Replacing Cron Jobs With Systemd Timers." *Mjanja Tech*, 10 July 2016, https://mjanja.ch/2015/06/replacing-cron-jobs-with-systemd-timers/.

"Cron vs Systemd Timers." *Unix & Linux Stack Exchange*, https://unix.stackexchange.com/questions/278564/cron-vs-systemd-timers.

The Clock Mini-HOWTO, https://www.tldp.org/HOWTO/Clock.html#toc1.

Rean, Rob. *The Clock Mini-HOWTO*, Nov. 2000, https://ftp://ftp.wayne.edu/ldp/en/Clock/Clock.pdf.

"Industry Standard Architecture." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/Industry_Standard_Architecture.

"Hwclock(8) - Linux Man Page." [Hwclock(8): Query/Set Hardware Clock - Linux Man Page_, https://linux.die.net/man/8/hwclock.

"Adjtimex(8) - Linux Man Page." *Adjtimex(8): Display/Set Kernel Time Variables - Linux Man Page*, https://linux.die.net/man/8/adjtimex.

"System Time." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/System_time.

"Timekeeping.c - Kernel/Time/Timekeeping.c - Linux Source Code (v5.0)." *Bootlin*, https://elixir.bootlin.com/linux/v5.0/source/kernel/time/timekeeping.c#L1302.

"How To Sync Your System Time to Hardware Clock Consistently." *Howtos: Hardware: syncing_hardware clock and system local time - SlackDocs*, https://docs.slackware.com/howtos:hardware:syncing_hardware_clock_and_system_local_time.

"Switch off 11-Minute Kernel Mode." *Unix & Linux Stack Exchange*, https://unix.stackexchange.com/questions/285129/switch-off-11-minute-kernel-mode.

Ma, Eric. "How to Set Date, Time and Timezone in Linux." *SysTutorials*, 5 Sept. 2020, https://www.systutorials.com/linux-setting-date-time-and-timezone/.

"IRQ0 - SYSTEM TIMER." *Int 08*, https://www.ctyme.com/intr/rb-0043.htm.

"Timex.h - Include/Uapi/Linux/Timex.h - Linux Source Code (v5.0)." *Bootlin*, https://elixir.bootlin.com/linux/v5.0/source/include/uapi/linux/timex.h#L128.

"High Precision Event Timer." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/High_Precision_Event_Timer.

"Time Stamp Counter." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2021, https://en.wikipedia.org/wiki/Time_Stamp_Counter.

Torpey, Bill. "Confessions of a Wall Street Programmer." *Measuring Latency in Linux - Confessions of a Wall Street Programmer*, https://btorpey.github.io/blog/2014/02/18/clock-sources-in-linux/.

"What Does the Change of the Clocksource Influence?" *Unix & Linux Stack Exchange*, https://unix.stackexchange.com/questions/164512/what-does-the-change-of-the-clocksource-influence.

"TSC Resynchronization." *The Chromium Projects*, https://www.chromium.org/chromium-os/how-tos-and-troubleshooting/tsc-resynchronization.

clock_getres(2) - Linux Manual Page, https://man7.org/linux/man-pages/man2/clock_gettime.2.html.

"Difference between CLOCK_REALTIME and CLOCK_MONOTONIC?" *Stack Overflow*, https://stackoverflow.com/questions/3523442/difference-between-clock-realtime-and-clock-monotonic.

The Linux Kernel Archives, https://www.kernel.org/doc/Documentation/timers/timekeeping.txt.

"Kernel Timer Systems." *Kernel Timer Systems - ELinux.org*, https://elinux.org/Kernel_Timer_Systems.

Cyberglory. "Jiffies in Linux Kernel." *Cyber Glory*, 21 Aug. 2011, https://cyberglory.wordpress.com/2011/08/21/jiffies-in-linux-kernel/.

"High Resolution Timers." *High Resolution Timers - ELinux.org*, https://elinux.org/High_Resolution_Timers.

The Linux Kernel Archives, https://www.kernel.org/doc/Documentation/timers/highres.txt.

"Clockevents and Dyntick." *LWN.net*, https://lwn.net/Articles/223185/.

Molnar, Ingo. "Kernel/Timer.c Design (Was: Re: Ktimers Subsystem)." *Lkml.org*, 19 Oct. 2005, https://lkml.org/lkml/2005/10/19/46.

"FreeBSD Manual Pages." *Eventtimers(4)*, https://www.freebsd.org/cgi/man.cgi?query=eventtimers&sektion=4&manpath=freebsd-release-ports.

"FreeBSD Manual Pages." *Hpet(4)*, https://www.freebsd.org/cgi/man.cgi?query=hpet&sektion=4&manpath=freebsd-release-ports.

"How to Change the Clock Source in the System." *Red Hat Customer Portal*, 12 Mar. 2021, https://access.redhat.com/solutions/18627.

Torvalds. "Torvalds/Linux." GitHub, 8 May 2021, https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/kernel-parameters.txt.

"Tinkering with Time." *Circuit Cellar*, 31 Jan. 2019, https://circuitcellar.com/cc-blog/tinkering-with-time/.

"CHU (Radio Station)." *Wikipedia*, Wikimedia Foundation, 24 Jan. 2021, https://en.wikipedia.org/wiki/CHU_(radio_station).

"WWV (Radio Station)." *Wikipedia*, Wikimedia Foundation, 14 Apr. 2021, https://en.wikipedia.org/wiki/WWV_(radio_station).

"Global Positioning System." *Wikipedia*, Wikimedia Foundation, 6 May 2021, https://en.wikipedia.org/wiki/Global_Positioning_System#Timekeeping.

GPS, UTC, and TAI Clocks, http://www.leapsecond.com/java/gpsclock.htm.

"GPS, Relativity, Andpop-Science Mythology." *GPS, Relativity, and Pop-Science Mythology*, http://alternativephysics.org/book/GPSmythology.htm.

"Time Dilation." *Wikipedia*, Wikimedia Foundation, 14 May 2021, https://en.wikipedia.org/wiki/Time_dilation.

"Real-World Relativity: The GPS Navigation System." *GPS and Relativity*, http://www.astronomy.ohio-state.edu/~pogge/Ast162/Unit5/gps.html.

GLONASS Time and Ephemeris, https://www.e-education.psu.edu/geog862/node/1875.

Esr. "The Rollover of Doom: a Trap for Good Programmers." *Armed and Dangerous*, 15 Jan. 2011, http://esr.ibiblio.org/?p=2869.

Esr. "Embracing the Suck." *Armed and Dangerous*, 22 Jan. 2011, http://esr.ibiblio.org/?p=2882.

Reference Clock Drivers, http://doc.ntp.org/4.1.2/refclock.htm.

"Understanding and Using the Network Time Protocol." *How Does It Work?*, https://www.eecis.udel.edu/~ntp/ntpfaq/NTP-s-algo.htm.

External Clock Discipline and the Local Clock Driver, http://doc.ntp.org/4.1.2/extern.htm.

rfc868, https://tools.ietf.org/html/rfc868.

rfc5905, https://tools.ietf.org/html/rfc5905.

"Network Time Protocol." *Wikipedia*, Wikimedia Foundation, 9 May 2021, https://en.wikipedia.org/wiki/Network_Time_Protocol.

Weber, Johannes. "Packet Capture: Network Time Protocol (NTP)." *Weberblog.net*, 6 May 2021, https://weberblog.net/packet-capture-network-time-protocol-ntp/.

Network Time Protocol, https://networking.ringofsaturn.com/Protocols/ntp.php.

"The NTP Pool Needs More Servers." *The NTP Pool Needs More Servers | NTP Pool News*, https://news.ntppool.org/2012/06/more-servers-please/.

Masterclock. "Internet vs GPS NTP Servers." *Masterclock, Inc.*, Masterclock, Inc., 10 May 2021, https://www.masterclock.com/company/masterclock-inc-blog/network-synchronization-internet-ntp-servers-vs-gps-ntp-servers.

NIST Internet Time Service, https://tf.nist.gov/tf-cgi/servers.cgi.

Ntpd - Network Time Protocol (NTP) Daemon, https://www.eecis.udel.edu/~mills/ntp/html/ntpd.html.

Miscellaneous Commands and Options, https://doc.ntp.org/4.2.8/miscopt.html.

"Does Ntpd Have a Default Driftfile?" *Unix & Linux Stack Exchange*, https://unix.stackexchange.com/questions/232767/does-ntpd-have-a-default-driftfile.

"Ntpd.c Networking - Busybox - BusyBox: The Swiss Army Knife of Embedded Linux." *Busybox*, https://git.busybox.net/busybox/tree/networking/ntpd.c.

"Rdate(1) - Linux Man Page." *Rdate(1): Time via Network - Linux Man Page*, https://linux.die.net/man/1/rdate.

Seriot, Nicolas. *A Tiny NTP Client*, http://seriot.ch/ntp.php.

"c h r o n y." *Chrony*, https://chrony.tuxfamily.org/comparison.html.

"PPM Clock Accuracy Examples." *Best Microcontroller Projects*, https://www.best-microcontroller-projects.com/ppm.html.

Lockhart, Roger. "What's All This PPM Stuff?" *Data Acquisition and Data Loggers Whats All This PPM Stuff*, 10 Aug. 2011, <www.dataq.com/blog/data-logger/whats-all-this-ppm-stuff/>.

Chris's Wiki :: Blog/Sysadmin/NtpdToChrony, https://utcc.utoronto.ca/~cks/space/blog/sysadmin/NtpdToChrony.

"Clockspeed." *Cr.yp.to*, http://cr.yp.to/clockspeed.html.

"Systemd-Timesyncd." *Systemd-Timesyncd - ArchWiki*, https://wiki.archlinux.org/index.php/Systemd-timesyncd.

Chris's Wiki :: Blog/Linux/SwitchingToTimesyncd, https://utcc.utoronto.ca/~cks/space/blog/linux/SwitchingToTimesyncd.

Keene Enterprises :: Open Source :: HTP, http://rkeene.org/oss/htp/.

Techopedia. "What Is Simple Network Time Protocol (SNTP)? - *Definition from Techopedia."* *Techopedia.com*, Techopedia, 13 Sept. 2011, https://www.techopedia.com/definition/4539/simple-network-time-protocol-sntp.

External Clock Discipline and the Local Clock Driver, https://www.eecis.udel.edu/~mills/ntp/html/extern.html.

Clock Discipline Algorithm, https://www.eecis.udel.edu/~mills/ntp/html/discipline.html.

"Leap Smear." *Google*, Google, https://developers.google.com/time/smear.

"Making Every (Leap) Second Count with Our New Public NTP Servers." *Google*, Google, https://cloud.google.com/blog/products/gcp/making-every-leap-second-count-with-our-new-public-ntp-servers.

Kamp, Poul-Henning. "The One-second War (What Time Will You Die?) As more and more systems care about time at the second and sub-second level, finding a lasting solution to the leap seconds problem is becoming increasingly urgent." *Queue 9.4* (2011): 10-16.

"Don't Trust Time." *YouTube*, YouTube, 25 Aug. 2017, https://www.youtube.com/watch?v=ylfyezRhA5s.

"Securing Network Time." *Core Infrastructure Initiative*, 26 Apr. 2018, https://www.coreinfrastructure.org/news/blogs/2017/09/securing-network-time.

Welcome to NTPsec, https://www.ntpsec.org/.

Encryption, http://www.ntp.org/ntpfaq/NTP-s-algo-crypt.htm.

Schick, Shane. "How the Kiss O'Death Packet and Other NTP Vulnerabilities Could Turn Back the Internet's Clocks." *Security Intelligence*, https://securityintelligence.com/news/how-the-kiss-odeath-packet-and-other-ntp-vulnerabilities-could-turn-back-the-internets-clocks/.

Andy. *The Ongoing Struggle*, 25 Dec. 2018, http://strugglers.net/~andy/blog/2018/12/24/the-internet-of-unprofitable-things/.

Martinec, Mark. *NTP Temperature Compensation*, 3 Jan. 2001, https://www.ijs.si/time/temp-compensation/.

"NTP vs. PTP: What's The Difference?" *TimeMachines Inc.*, 11 Sept. 2019, https://timemachinescorp.com/2018/11/06/ntp-vs-ptp-whats-the-difference/.

"Precision Time Protocol." *Wikipedia*, Wikimedia Foundation, 21 Jan. 2021, https://en.wikipedia.org/wiki/Precision_Time_Protocol.

Precision Time Protocol, https://www.cs.rutgers.edu/~pxk/417/notes/ptp.html.

Arnold, Douglas. "What Are All Of These IEEE 1588 Clock Types?" *Time Synchronization*, 21 Oct. 2013, https://blog.meinbergglobal.com/2013/10/21/ieee-1588-clock-types/.

Delalandre, Mathieu. *Distributed Computing "Time Synchronization,"* University of Tours, https://mathieu.delalandre.free.fr/teachings/dcomputing/part2.pdf.

ICHIKAWA, Ken. "Precision Time Protocol on Linux." *Precision Time Protocol on Linux ~ Introduction to Linuxptp*, FUJITSU LIMITED., 2014, https://events.static.linuxfound.org/sites/events/files/slides/lcjp14_ichikawa_0.pdf.

TimeMachines Time Server Accuracy, TimeMachines, https://www.timemachinescorp.com/wp-content/uploads/TMTimeServerAccuracyRevB.pdf.

"Signals and Threads Podcast." *Signals and Threads - Clock Synchronization*, https://signalsandthreads.com/clock-synchronization/.

"What Consequences / Implications Can Arise from a Wrong System Clock?" *Server Fault*, https://serverfault.com/questions/284638/what-consequences-implications-can-arise-from-a-wrong-system-clock.

"Time Formatting and Storage Bugs." *Wikipedia*, Wikimedia Foundation, 13 May 2021, https://en.wikipedia.org/wiki/Time_formatting_and_storage_bugs.

The Trouble with Timestamps, https://aphyr.com/posts/299-the-trouble-with-timestamps.

Time Disorder. https://caolan.uk/articles/time-disorder/.

"What Is Distributed Computing." *IBM*, https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.2.0/com.ibm.cics.tx.doc/concepts/c_wht_is_distd_comptg.html.

Luckham, David C., and Brian Frasca. "Complex event processing in distributed systems." *Computer Systems Laboratory Technical Report CSL-TR-98-754*. Stanford University, Stanford 28 (1998): 16.

"Clock Synchronization." *Wikipedia*, Wikimedia Foundation, 22 Feb. 2021, https://en.wikipedia.org/wiki/Clock_synchronization.

"What Is a Real-Time Operating System (RTOS)?" *NI*, https://www.ni.com/white-paper/3938/en/.

Cedeño, Walter, and Phillip A. Laplante. "An overview of real-time operating systems." *JALA: Journal of the Association for Laboratory Automation* 12.1 (2007): 40-45.

RTOS, http://www.inf.ed.ac.uk/teaching/courses/es/PDFs/RTOS.pdf.

"The Many Challenges of Real-Time Linux." *IntervalZero*, 26 Sept. 2018, https://www.intervalzero.com/real-time-2/the-many-challenges-of-real-time-linux/.

Foundation, The Linux. "Intro to Real-Time Linux for Embedded Developers." *Linux Foundation*, The Linux Foundation, 22 Aug. 2017, www.linuxfoundation.org/blog/2013/03/intro-to-real-time-linux-for-embedded-developers/.

"Open Source RTOS Kernel for Small Embedded Systems - What Is FreeRTOS FAQ?" *FreeRTOS*, 13 Nov. 2019, https://www.freertos.org/FAQWhat.html.

"The Friendly Operating System for the Internet of Things." *RIOT*, https://riot-os.org/.

S, Dave. "What Are 'Co-Operative' and 'Pre-Emptive' Scheduling Algorithms?" *Rapita Systems*, 22 Oct. 2013, https://www.rapitasystems.com/blog/cooperative-and-preemptive-scheduling-algorithms.

BlackBerry QNX, https://blackberry.qnx.com/en.

"Advancing Open Source Safety-Critical Systems." *ELISA*, 6 May 2021, https://elisa.tech/.

UGT, https://www.total-knowledge.com/~ilya/mips/ugt.html.

"Swatch Internet Time." *Wikipedia*, Wikimedia Foundation, 2 Apr. 2021, https://en.m.wikipedia.org/wiki/Swatch_Internet_Time.

Louis, Patrick. "Time On The Internet." *Venam's Blog*, Patrick Louis Aka Venam, 28 Feb. 2019, https://venam.nixers.net/blog/psychology/2019/03/01/internet-time.html.