What every developer should know about time

Martin Thoma
E-Mail: info@martin-thoma.de

***Abstract*—This paper introduces basic concepts around time, including calendar systems, time zones, UTC and offsets. It gives a brief historic overview of systems that are applied to simplify the understanding.**

I. Introduction

Time is such a fundamental concept that we rarely think about it in detail. When one is forced to develop software or analyzes data generated by software, one needs to understand the edge cases. This paper is a short introduction to those concepts and edge cases. The paper is inspired by [Sus12a], [Sus12b] and John Skeet's talk at NDC London in January 2017.

The target audience of this paper are computer scientists and developers. There are many more details relevant to historians like other calendar types, units of time such as Danna [Eng88] or techniques like radiocarbon dating. While this paper touches many topics, I recommend [Sei05] for details.

II. A brief historic overview

The history of time and dates is driven by religious beliefs, economic interests, technological advances and navigation.

One of the oldest types of time measurement is measuring the apparent solar time by sundial. In *sundial time*, the time of the day when the sun reaches the local meridian is defined as 12:00 [Ste07].

1500 BC: Oldest known sundial was created [Bor11].

46 BC: The Julian Calendar was proposed. In the Julian Calendar, every 4^{th} year is a leap year. This means a Julian year is 365.25 days, which is also used as the basis for a light year [Wil89]. The Julian Calendar brought the 12 months in line with the seasons.

1582: The Gregorian calendar was introduced to counter seasonal drift [Wik18b]. It was desired that the Christian holiday of Easter is on spring equinox - the day, when there is an equal time of daytime and nighttime are of approximately equal duration all over the planet. To keep equinox around 20 March and 22–23 September, every 4^{th} year is a leap year except for every 100^{th} year. But every year that is divisible by 400 is again a leap year.

1712: Sweden wanted to gradually change from the Julian Calendar to the Gregorian calendar by skipping leap years for 40 years. During the Great Northern War, however, they didn't skip the leap years in 1704 and 1708. To restore the Julian Calendar, they had to add another day. As 1712 was already a leap year, they added February the 30^{th}, making it a double-leap year [dou].

1807: The Noon Gun starts firing a time signal in Cape Town, South Africa [Bis79]. This allows ships in the port to check the accuracy of their marine chronometers. Marine chronometers are used on ships to help calculate the longitude.

1825: The Stockton and Darlington Railway opened [Tom15]. This raised the need for synchronized times for train schedules started to rise. Often, the time of a big city like Berlin was chosen. This was then called *Berlin Standard Time*.

1838: Telegraphy made time synchronization possible [TM99].

1876: After missing a train, Sir Sandford Fleming proposes to use a 24-hour clock. So instead of distinguishing 6am and 6pm, he proposes to distinguish 6 o'clock and 18 o'clock.

1884: Sir Sandford Fleming proposed a worldwide standard time at the International Meridian Conference to which 24 time zones of $\frac{360^\circ}{24} = 15^\circ$ latitude are added as local offsets. This way, the local time at each place would be at most half an hour off from the standardized time and simplify the system (see Figure 5). That conference accepted a different version of Universal Time but refused to accept his zones, stating that they were a local issue outside its purpose. In this conference, **the prime meridian was defined to be the Royal Observatory, Greenwich, United Kingdom**.

1891: The Prussian railway replaced Berlin time with Central European Time (CET) as a common time [Bar07].

1893: The Imperial German government adopted CET for all state purposes [Bar07].

1916: The German Empire introduces Daylight Saving Time [vD16].

1924: The Greenwich Time Signal is introduced as a way to synchronize time [McI90].

1955: **The International Time Bureau (BIH) began a time scale using both local Caesium clocks and comparisons to distant clocks** using the phase of VLF radio signals [GA05].

1956: The United States Naval Observatory began the A.1 scale using a Caesium standard atomic clock [For85].

1959: DCF77 started service as a standard-frequency station.

1960: The Coordinated Universal Time (UTC) was introduced [McC09].

1967: **The SI second was defined in terms of the Caesium atom** (see Section III). Based on this exact definition, the *Temps Atomique International* (TAI) is calculated by the International Bureau of Weights and Measures (BIMP).

1968: The Universal Time No.1 (UT1) was introduced as a successor of Greenwich Mean Time (GMT). It is based on astronomical observations such as the mean solar time. In the same year, the definition of a second was changed from astronomical observations to atomic clocks.

1972: The first leap second was introduced [Leaa].

1986: The work on the IANA time zone database began.

III. Units

The SI base unit for time is a *second*. A second is historically defined as the fraction $\frac{1}{60 \cdot 60 \cdot 24}$ of the mean solar day. [Bro16] defines it more formally by a physical process:

> The second is the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium 133 atom.

Other time units are based on the second:

- All SI-prefixes, especially $1\,\mathrm{ns} = 10^{-9}\,\mathrm{s}$, $1\,\mu\mathrm{s} = 10^{-6}\,\mathrm{s}$, and $1\,\mathrm{ms} = 10^{-3}\,\mathrm{s}$. Applications in physics can go down to an attosecond ($10^{-18}\,\mathrm{s}$) [HKS+01].
- 60 seconds are one minute.
- 60 minutes are one hour.
- 24 hours are one standard day. The word "standard" is important here, for example when you think about Daylight Saving Time.

A $360°$ rotation of earth is called a sidereal day, while the noon to noon rotation is called a solar day. Due to the movement of Earth around the sun, this is not the same. Neither a sidereal day nor a solar day are of the same length as the standard day.

An *sidereal year* is the time it takes Earth to orbit around the sun with respect to fixed stars. A *solar year* is the time it takes the sun to return to the same position in the cycle of seasons as seen by Earth. As the axial precession has a cycle of 25 722 years [dIV18], the difference between a sidereal year and a solar year is about $\frac{365 \cdot 24 \cdot 60}{25722} \approx 20 \frac{\mathrm{min}}{\mathrm{year}}$).

A *calendar year* is an approximation of either a solar year, a sidereal year or different systems like lunar years. A *lunar month* is the time it takes the moon to orbit around the earth. Similar to the calendaric definitions based on Earths rotation around the sun, there are different lunar months depending on how a full rotation around the Earth is defined. A calendar month is an approximation to it. A day is the time it takes earth to rotate around it's axis.

And that is where we get problems: All of the calendar units have at least two definitions which don't match. Additionally, a year and a month does not consist of a round multiple of days. For this reason, leap days are introduced. Also, the speed of the rotation of the earth is not constant. To keep the exactly measured atomic time synchronized with Earths rotation, leap seconds are introduced.

To emphasize that there are two definitions which do not match, I want to explain two concepts of a day: One is 24 hours = $24 \cdot 60 \cdot 60$ SI seconds. The other one is a solar day. It is defined by a point on earth pointing exactly to the sun. The duration as measured in seconds of such a solar day differs from day to day. This can be seen with an "analemma", a diagram of the suns position against the local time.

IV. Measuring and Synchronizing Time

Within desktop computers are real time clocks (RTCs) which are powered by a small battery. RTCs are chips which contain a crystal oscillator. Often Quartz is used, because it is cheap, uses very little power and is accurate. It can gain or lose up to 15 seconds every 30 days. This phenomenon is called *clock drift*.

Today, many operating systems use the Network Time Protocol (NTP) to get the current time via internet. The used time servers can have more accurate sources like atomic clocks. The protocol also compensates network latency. Over the public internet it is accurate within a few tens of milliseconds. On local networks it can be reduced to less than $1\,\mathrm{ms}$. A set of atomic clocks throughout the world keeps time by consensus. This is defined to be the International Atomic Time (TAI).

For a security in NTP, see [DSZ16].

The Precision Time Protocol (PTP) focuses on higher accuracy than NTP within local networks. It is standardized in IEEE 1588. If PTP is implemented in hardware it can reach errors of less than $10\,\mathrm{ns}$ [Wei12] and implemented in software within milliseconds [GE17]. According to [Wei12], PTP is used to coordinate measurement instants, to measure time intervals, as a reference to determine the order of events and the age of data items.

V. Calendars

The Julian calendar only had one leap day every 4 years making the average year have 365.25 standard days. Comparing it to the 365.24219 standard days of a tropical year, the Julian calendar has an error of $-0.00781 \frac{\mathtt{standard\ day}}{\text{tropical year}}$ and the solar days shift from the standard days by one day every 128 years.

Today, we use the Gregorian calendar in most parts of the world. It is named after Pope Gregory XIII, who introduced it in October 1582. It uses leap years to make the average year $365 + \frac{1}{4} - \frac{1}{100} + \frac{1}{400} = 365.2425$ days long, by inserting a leap day at the end of February every fourth year except for every 100th year except for every 400th year. The Gregorian calendar has an error of $-0.00031 \frac{\mathtt{standard\ day}}{\text{tropical year}}$ and thus the shift is only one day every 3226 years. This is a major improvement compared the Julian calendar which was used before.

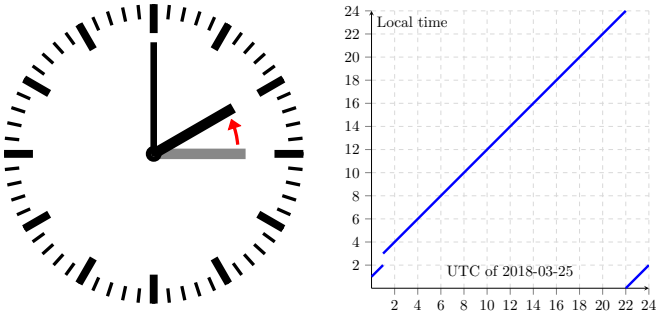
Internally, the computer can keep track of the year by storing the year as two characters. This is known as the Y2K or Millennium bug.

The majority of countries today use the Gregorian calendar, but some countries like India and Israel use other calendar systems alongside to the Gregorian calendar; mostly for holidays. Two exceptions are Iran and Afghanistan, which only use the Persian calendar (Solar Hijri Calendar).

VI. Time Zones

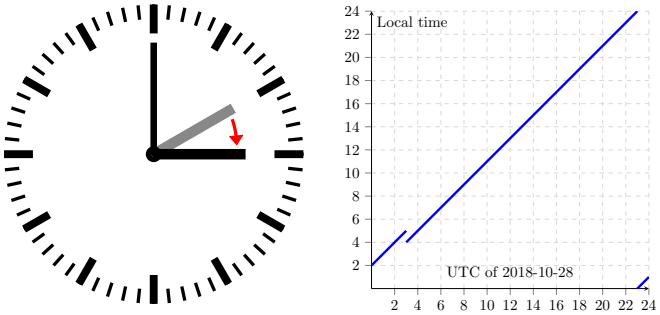
A time zone is a region where the same time is used. Time zones convert universal time into local time. The local time is defined by it's offset(s) compared to UTC - the Universal Time Coordinated. A single time zone often has multiple offsets depending on the date. UTC is not a time zone, but a standard. The offset of a time zone can change.

See Figure 4 and Figure 3 for an example of a timezone function which is neither injective nor surjective. This means one can always unambiguously convert UTC to a local time, but not always vice versa.



(a) Start of DST

(b) Start of DST



(c) End of DST

(d) End of DST

Figure 1: Visualization of the application of DST in Germany. The function plots show how UTC time is mapped to local time. While the mapping from UTC to local time is unambiguous, the function is clearly neither surjective nor injective: Some local times don't exist and for some local times two possible UTC times exist.

Having the UTC time with the offset, for example `2018-09-02T18:40:00+02:00`, we know the instant in time and the current local time (18:40). We don't know what local time it will a minute later.

Time zone abbreviations can be ambiguous. For example, IST can be Indian Standard Time or Irish Standard Time. CST can be Central Standard Time or China Standard Time.

A single city can also have two time zones: Nicosia, the capital of Cyprus, is one example [And16]. The north of the city has the Turkish time zone, the south has the Greek time zone.

Two places can have the same local time, but still not be in one time zone. For example, during DST, California and Arizona have the same time zone. They are, however, not in the same time zone as Arizona does not use DST but California does.

Greenwich Mean Time (GMT) was the same time as Universal time (UT) until 1972. Since then, GMT is no longer a standard but a time zone.

Typically, the offset is by complete hours. But Nepal, for example, has an offset of 5.25 hours.

The offset can also change: For example, the time zone of Samoa changed from having UTC-11:00 to UTC+13:00 on December 29th, 2011. The reason for this is trade with Australia and New Zealand, which were almost one calendar day ahead. So on Fridays in Samoa it was already Saturday in Australia and on Sunday in Samoa it was already Monday in Australia. This switch also is a reason for another oddity: Samoa skipped Friday. The time went from 2011-12-30 10:00-11:00 to 2011-12-30 10:00+13:00

It is also possible to have multiple time zones within one region due to political disputes [BA13].

A commonly used database for time zones is the *IANA time zone database* [Ols09]. It has the mailing list https://mm.icann.org/mailman/listinfo/tz through which errors are reported.

VII. Daylight Saving Time

According to [Wik18a], 72 out of 193 UN member countries currently use DST. 29 of them start DST on the last Sunday in March at 01:00 UTC, 12 more countries also use the last Sunday in March but the article didn't specify a time. Mexico uses the first Sunday April, Israel the Friday before last Sunday March, Jordan and Syria the last Friday of March.

When DST is switched on, the name of the time zone changes. For example, the time zone used in New York is Eastern Standard Time (EST) during winter and Eastern Daylight Time (EDT) during summer.

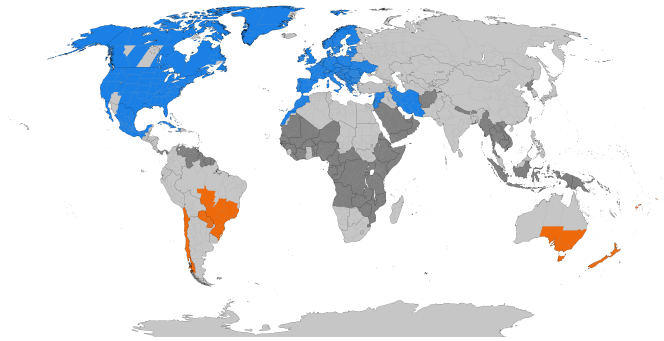


Figure 2: Blue countries apply northern hemisphere summer time, orange ones apply southern hemisphere summer time, bright gray ones formerly used DST and dark gray ones never applied DST. Author: TimeZonesBoy

VIII. Time Keeping

The history of time keeping devices dates back as early as 2000 BC, but the devices, protocols and infrastructure relevant for today's computer system was invented after 1950.

The devices which are most accurate in measuring durations are Caesium clocks. The first accurate atomic clock was built in 1955 [EP55].

The first version of NTP was published in 1985 [M$^+$85]. An updated version [MMBK10] is still in use today. Computers that synchronize time via NTP do so every 2^τ minutes, with values from $\tau = 2$ (4 seconds) to $\tau = 17$ (36h 24min 32s).

The source of time is called a primary server or stratum-1. A primary server is synchronized to a reference clock (stratum-0) directly traceable to UTC. Such a reference clock could either be a Caesium clock, a GPS receiver or receivers of terrestrial broadcasts like DCF77 or MSF. Radio clocks and watches use long-wave time signals like DCF77 in Germany, MSF in England and France Inter in France to synchronize themselves automatically. DCF77 has an atomic clock that is also synchronized with PTB in Braunschweig - the legal time in Germany.

Note that the UTC time is synchronized via NTP, but the time zone is applied only locally. This means the local time zone library has to be updated once in a while.

NTP prevents time drift because the systems continuously synchronize time.

When the client polls the time from the server, the difference between server time and client time is called Δ. If $\Delta > 1000\,\mathrm{s}$ (16 min 40s), the program exits with a diagnostic message. If $1000\,\mathrm{s} \geq \Delta > 125\,\mathrm{ms}$, then the clock is stepped to the correct time.

In this process, errors are introduced in various levels:

- **GNSS**: Global navigation satellite systems like GPS have multiple satellites of which each contains an atomic clock. In order to counter relativistic effects (see Section A), their clocks are running slower by design. As the signal needs time to arrive at the receivers, this has to be corrected as well. Hence errors in the satellite position also introduce an error in the current time. In the Ionosphere, there is a variable time delay which depends on the electron density and in the troposphere refraction causes issues.
- **Network latency**: Packages sent over the internet don't always take the same amount of time. Queuing is one factor that influences it.
- **Clock drift**: Between two synchronizations, the local clock ticks slightly slower or faster, than it should be.

IX. Representation

A. NTP Timestamp Format

The NTP timestamp format has 64-bit which are split into 32-bit unsigned seconds field spanning 136 years and a 32-bit fraction field resolving 232 picoseconds. As NTP uses an epoch of January 1$^{\text{st}}$ 1900 this yields an overflow in 2036.

B. NTP Date Format

The NTP date format has 128 bit. It includes a 64-bit signed seconds field spanning 584 billion years and a 64-bit fraction field resolving 0.05 attosecond (i.e., $0.5 \cdot 10^{-18}$).

C. Single Numbers

Once it is possible to measure the time duration's reliably, one can define a special date. Every point in time is then simply this date plus an offset.

For the Unix Timestamp, this is 1$^{\text{st}}$ of January, 1970 at midnight at UTC. Then the elapsed seconds are counted. A common variation is to count milliseconds. Those two timestamp formats can be recognized by typically having 10 characters or 13 characters.

Typically, the number of elapsed seconds is stored in a signed 32-bit integer. The problem with this approach is that we can only define seconds and only 2^{32} different values. Hence the first timestamp possible is -2^{31} which is 1901-12-13T20:45:52Z and the last possible time is $2^{31} - 1$ which is 2038-01-19T03:14:07Z. This is known as the Year 2038 problem. A simple solution is the upgrade to 64-bit integers which expands the borders to well Before Christ and beyond the year $10\,000$ even if the resolution was increased from seconds to microseconds. Another solution is to use a format which allows arbitrary size such as PyInt [MZ01], variable size integers (VarInts) of Go and Protocol Buffers [Lau18].

Similar to the Unix timestamp, the *Julian day* counts the number of days since the beginning of the Julian period. Julian day number 0 is assigned to the day starting at noon on Monday, January 1, 4713 BC [jul01]. Hence the date 2000-01-01T12:00Z has the Julian day $2\,451\,545$ [Sei05]. The Modified Julian Day (MJD) is defined as the Julian Day (JD) reduced by $2\,400\,000.5$ [Win]. Both time formats are relevant for astronomy and geodesy.

D. Multiple Numbers

Python stores datetimes by the different components (year, month day, hour, minute, second, μ-seconds) as shown in Table I. This is similar to what C stores [Sta05], with the notable exception of the range of seconds and the year representation. The number of second can be 60 in the case of leap seconds. For C, the Year 0 is 1990

Name	Bytes	Values
Year	2	1–9999
Month	1	1–12
Day	1	1–31
Hour	1	0–23
Minute	1	0–59
Second	1	0–59
μ-second	3	0–999 999

Table I: Structure of Pythons datetime format.

X. Recurring Events

It is a common task to implement software that has to run on re-occuring events: Paying the monthly rent in a banking system at the end of a month, marking a birthday as "occurs every year on the same day" in a calendar system, running a computationally intensive task at 02:00, sending out status reports every Monday at 08:00.

For example, birthdays have the following problem:

- The simple solution to add 365 days for a year is not good enough, because the current year might be a leap year and thus one would calculate the birthday one year off.
- Increasing the year by one does not work when the birthday is on the leap day.

This leads to the distinction between a `Duration` and a `Period` as explained in the Concepts section. Two common solutions for dealing with such problems are RRULES as specified in RFC-5545 and CRON.

CRON specifies those events as

```
M H D m w user command
```

where `M` is the minute, `H` is the hour, `D` is the day, `m` is the month and `w` is the weekday. Here are two examples:

- `23 13 * * *` repeats every day at 23:13
- `0 4 3 * *` repeats every 3rd of a month at 4:00

RRULES are more flexible. They define a timezone and a datetime at which they start, a RRULE which contains the frequency of recurrence and how long it lasts. One example is:

```
DTSTART;TZID=Europe/Rome:20181003T090000
RRULE:FREQ=DAILY;UNTIL=20191230T095500Z
```

End dates are not required, though. The following example defines an event that happens every second day:

```
DTSTART;TZID=Europe/Rome:20181003T090000
RRULE:FREQ=DAILY;INTERVAL=2
```

XI. Concepts

Jon Skeet's date and time API "Noda Time" distinguishes several date and time related concepts. This distinction makes it easier to understand allowed operations and allows the enforcement of them via compiler.

A. Instant

An *Instant* is a point in time. It is always the same world wide. A possible representation of an instant is in UTC, for example `2018-09-05 07:22:00Z` were `Z` is short for Zulu time. Another representation are Unix Timestamps, where the 1st of January 1970 at 00:00 with no offset is the fixed reference starting point.

B. Duration

A duration is elapsed time. It can be measured in seconds, minutes, standard days, or many others. Most notably, a year, a month and a week are not durations as they are based on calendar systems. For example, the Bahá'í calendar does not have the concept of a week.

C. Interval

An interval is a duration combined with an instant. For example, `[2018-09-05T07:22:00Z, 2018-09-05T07:23:00Z]` or `(2018-09-05T07:22:00Z, 1min)` both represent the same one minute interval in time.

D. Calendar System

A calendar system is a way of structuring local time. Examples are the Gregorian calendar, the Incan calendar, Solar Hijri calendar, and the Assyrian calendar. There are many more calendar systems, of which most are not in use anymore. Today, the Gregorian calendar is the dominating calendar system, but especially for holidays others are still in place. For example, the Pawukon calendar in Bali (Indonesia) or the Tamil calendar in India.

As the Gregorian calendar dominates the world today, this paper will not explain any specialties of calendar systems.

E. Date

A date is associated with a calendar system. In the Gregorian calendar, it is denoted by the year, a month and a day.

One could think that having a datetime where you set the time to any fixed value like `00:00:00` is equivalent to a date object. Bugs like [lev11] show that this is not the case. Somebody who was born in Auckland (New Zealand) on 1990-04-28 04:00:00 would not suddenly say he was born on 27th, just because he visits Pago Pago in the time zone `Pacific/Pago Pago`.

F. Time of the day

The local time or time of the day is what a watch shows.

G. Timezone-Unaware Fixed-Calendar Datetime

The time of the day and a date combined. As there is the date, a calendar system is implicitly used. As it is timezone-unaware, it is not associated with a time zone. This uses one single, fixed calendar system. `2018-12-31 23:58:53 (Gregorian calendar)` is one example. Note that this is ambiguous due to time zones, in contrast to an `Instant`.

H. Timezone-Unaware Variable-Calendar Datetime

Similar to time zones, there should are calendar-zones, that means regions on Earth which define a calendar system. Those calendar systems change. For example, in the UK in 1752, it was switched on September the 2nd of the Julian calendar to September the 14th of the Gregorian calendar. So the local date September the 3rd never occurred in the UK. Still, it is possible to define this day in both, the Gregorian and the Julian calendar. `2018-12-31 23:58:53 (United Kingdom/London calendar)` is one example. In contrast to timezone-unaware fixed-calendar datetime, this represents only times that people at the time actually had in their calendar. Just like the timezone-unaware fixed-calendar datetime, the timezone-unaware variable-calendar datetime is not an `Instant` due to the ambiguity introduced by not specifying a time zone.

I. Time zones

A time zone is a function of time. It maps UTC time to local time. See Section VI.

J. Period

A period is similar to a duration, but takes calendar expressions into account. Where durations have a direct and clear relationship to SI seconds, the length of period depends on the start time. For example, a period of one month added to the Date 2018-02-01 results in the date 2018-03-01. A period of a month added to that results in the 2018-04-01. So the same period added in February is only 28 days, but 31 days if added to first of March.

K. Operations

Date and time related objects have a lot of possible operations. For example, an interval can be shrunken from the end or symmetrically by a given duration. In most cases operations between two date/time objects are ambiguous. For this reason, the + and − operators should in many cases not be used. Please also note that adding durations like `5min` to anything which is not an instant is ambiguous, as local (timezone-unaware) time is not continuous. The simplest example for this is DST, where adding `5min` to the local time `2018-10-29T02:58` can result in `2018-10-29T02:03` (for example in Germany) or in `2018-10-29T03:03`.

XII. Common Problems

One of the best documented problems is that of the maximum representable time. On one form, it is known as the Year 2000 Problem (Millennium bug) if the year is stored only by the last two digits. In another form, it is known as the Year 2038 Problem if a signed 32-bit integer is used for Unix Timestamps. Similar, there is the problem of the minimum representable time. For this kind of bug, [Mil99] documents that in Waadt (Switzerland) for two days almost all hospitals could not check-in new patients, a waste-water spill of 15 million liters of water in Los Angeles, 20 000 credit card swipe machines were unusable [KC99]. 426 inhabitants of Munich (Germany) were sent payment orders almost 100 years ago [Mun99]. A similar bug happened to Telecom Italia. The structure of the bug was as follows:

```
dueDate ← serviceDate + 1 month   ▷ Overflow happened
if dueDate < currentDate then
    Send payment order
end if
```

Other common problems include leap year bugs:

- Microsoft Azure was taken offline by the leap year bug on 2012-02-29 00:00Z. A security certificate was created by incrementing the year by one on February 29th, but February 29th 2013 is an invalid date [Lai12]. Exactly the same bug was in CouchDB [Buc16].
- Dusseldorf International Airport refused to let 1200 pieces of luggage onto planes [Kan16].
- A lot more are mentioned in [Joh16a].

Calculating how much time passed: The idea that you can simply create two datetime objects, get the number of seconds / hours passed in between and by this calculating the number of years passed is wrong. DST, leap days and calendar changes can break things.

XIII. Solutions

In general, it is a good idea to use well-maintained packages based on the IANA time zone database for dealing with time zones. For representing and exchanging time-based data ISO-8601 with the offset and the time zone name or - if the local time is not relevant - in UTC is a good idea.

A. Storing an Instant

In practice it is often necessary to store when something happened: When did a customer book a hotel room? When was the payment sent? At which time does the plane land so that a taxi can be ordered?

For the decision what to store, it is important to be clear about how the instant is used. If it is only used to put the event in a larger context such as with log files or payment information, then it should be stored in UTC. For example, `2018-09-15T11:56:59Z`.

If the local time is important, then UTC with the offset should be stored, like in `2018-09-15T13:56:59+02:00`. If the instant should be used to calculate something based on it, then it is necessary to store the timezone as well. For example `2018-09-15T13:56:59+02:00`, `Europe/Berlin`.

B. Serialization of Instants

The recommended way to serialize instants is by using the Date Time String Format of ECMA-262 (JavaScript):

```
YYYY-MM-DDTHH:mm:ss.sssZ
```

The reason why this format is recommended are:

- **Standards**: The format is specified in a standard. It is also conform to ISO 8601 and equals the internet date-time format specified in RFC3339 [NK02].
- **Human readable**: In contrast to Unix Timestamps, it is fairly easy for humans to read this format.
- **Lexicographically Sortable**: It is not necessary to convert this representation. Due to this fact, even systems with bad datetime support can at least sort.

If local time is of interest, the IANA time zone name should be saved. `Europe/Berlin` is one example of such a time zone name.

Similarly, a date should be stored in the format `YYYY-MM-DD`.

C. Serialization of Durations

A common way to serialize durations is to use the smallest unit of interest and store an integer. Most of the time this unit will be seconds or milliseconds.

References

[And16] E. Andreou, "Cyprus' new division: two time zones now a reality," Oct. 2016. [Online]. Available: https://cyprus-mail.com/2016/10/30/cyprus-new-division-two-time-zones-now-reality/

[Asl12] H. Aslaksen, "Why is singapore in the "wrong" time zone?" Jun. 2012. [Online]. Available: http://www.math.nus.edu.sg/aslaksen/teaching/timezone.html

[BA13] Y. Ben-Ami, "The world's only ethnic time zone," Oct. 2013. [Online]. Available: https://972mag.com/the-worlds-only-ethnic-time-zone/81006/

[Bar07] I. Bartky, *One Time Fits All: The Campaigns for Global Uniformity*. Stanford University Press, 2007. [Online]. Available: https://books.google.de/books?id=rC6sAAAAIAAJ

[Bis79] W. Bisset, "A new look at the castle of good hope and its symbolic importance," *Scientia Militaria: South African Journal of Military Studies*, vol. 9, no. 3, pp. 1–6, 1979. [Online]. Available: https://www.ajol.info/index.php/smsajms/article/viewFile/144208/133876

[Bor11] L. Borchardt, "Eine reisesonnenuhr aus ägypten," *Zeitschrift für ägyptische Sprache und Altertumskunde*, vol. 49, no. 1-2, pp. 80–82, 1911.

[Bro16] S. Brochure, "The international system of units (si)[2006; updated in 2014]," *Bureau International des Poids et Mesures, F-92310 Sevres, France*, 2016. [Online]. Available: https://www.bipm.org/en/publications/si-brochure/second.html

[Buc16] M. Buckett, "favicon produces a stack trace on february the 29th." CouchDB Issue Tracker, Mar. 2016. [Online]. Available: https://issues.apache.org/jira/browse/COUCHDB-2956

[dIV18] A. de Iaco Veris, *Practical Astrodynamics*. Springer, 2018.

[dou] "February 30 was a real date," timeanddate.com. [Online]. Available: https://www.timeanddate.com/date/february-30.html

[DSZ16] B. Dowling, D. Stebila, and G. Zaverucha, "Authenticated network time synchronization." in *USENIX Security Symposium*, 2016, pp. 823–840.

[Eng88] R. K. Englund, "Administrative timekeeping in ancient mesopotamia," *Journal of the Economic and Social History of the Orient*, vol. 31, no. 2, pp. 121–185, 1988.

[EP55] L. Essen and J. Parry, "An atomic standard of frequency and time interval: a caesium resonator," *Nature*, vol. 176, no. 4476, p. 280, 1955.

[For85] P. Forman, "Atomichron®: the atomic clock from concept to commercial product," *Proceedings of the IEEE*, vol. 73, no. 7, pp. 1181–1204, 1985.

[GA05] B. Guinot and E. F. Arias, "Atomic time-keeping from 1955 to the present," *Metrologia*, vol. 42, no. 3, p. S20, 2005.

[GE17] E. Gedda and A. Eriksson, "Practical analysis of the precision time protocol under different types of system load," 2017.

[Goo18] "Leap smear," Google Public NTP, Aug. 2018. [Online]. Available: https://developers.google.com/time/smear

[HKS+01] M. Hentschel, R. Kienberger, C. Spielmann, G. A. Reider, N. Milosevic, T. Brabec, P. Corkum, U. Heinzmann, M. Drescher, and F. Krausz, "Attosecond metrology," *Nature*, vol. 414, no. 6863, p. 509, 2001.

[Hun17] R. Hunt, "Keeping time with amazon time sync service," AWS News Blog, Nov. 2017. [Online]. Available: https://aws.amazon.com/de/blogs/aws/keeping-time-with-amazon-time-sync-service/

[IER13] "Measuring the irregularities of the earth's rotation," 2013. [Online]. Available: https://www.iers.org/IERS/EN/Science/EarthRotation/EarthRotation.html

[Joh16a] M. Johnson, "List of 2016 leap day bugs," Feb. 2016. [Online]. Available: https://codeofmatt.com/2016/02/29/list-of-2016-leap-day-bugs/

[Joh16b] ——, “Time zone chaos inevitable in egypt,” Jul. 2016. [Online]. Available: https://codeofmatt.com/2016/07/01/time-zone-chaos-inevitable-in-egypt/

[jul01] “Resolution b1 on the use of julian dates,” XXIII. International Astronomical Union General Assembly, Jan. 2001. [Online]. Available: https://www.iers.org/IERS/EN/Science/Recommendations/resolutionB1.html

[Kan16] A. Kannenberg, “Flughafen-software kennt schalttag nicht - 1200 koffer gestrandet,” Feb. 2016. [Online]. Available: https://www.heise.de/newsticker/meldung/Flughafen-Software-kennt-Schalttag-nicht-1200-Koffer-gestrandet-3121045.html

[KC99] P. Kelso and J. Cassey, “Bug’s first strike as swipe machines fail,” *The Guardian*, 1999. [Online]. Available: https://www.theguardian.com/technology/1999/dec/30/hacking.security1

[kul18] kulseran, “Why could it be necessary to set a wrong time in production?” Reddit, Sep. 2018. [Online]. Available: https://www.reddit.com/r/learnprogramming/comments/9ebq7s/why_could_it_be_necessary_to_set_a_wrong_time_in/e5noo0c/

[Lai12] B. Laing, “Summary of windows azure service disruption on feb 29th, 2012,” Mar. 2012. [Online]. Available: https://azure.microsoft.com/en-us/blog/summary-of-windows-azure-service-disruption-on-feb-29th-2012/

[Lau18] B. Lau, “Protocol buffers encoding: Base 128 varints,” Google, Tech. Rep., 2018. [Online]. Available: https://developers.google.com/protocol-buffers/docs/encoding#varints

[Leaa] “Leap seconds,” USNO. [Online]. Available: https://www.usno.navy.mil/USNO/time/master-clock/leap-seconds

[Leab] “Leap seconds in utc until 31 december 2018.” [Online]. Available: ftp://ftp2.bipm.org/pub/tai/publication/leaptab/leaptab.pdf

[lev11] levi, “Is the javascript date object always one day off?” StackOverflow, Sep. 2011. [Online]. Available: IstheJavascriptdateobjectalwaysonedayoff?

[LMS05] P. Leach, M. Mealling, and R. Salz, “A universally unique identifier (uuid) urn namespace,” IETF RFC, Jul. 2005. [Online]. Available: https://tools.ietf.org/rfc/rfc4122.txt

[M+85] D. Mills *et al.*, “Network time protocol,” RFC 958, M/A-COM Linkabit, Tech. Rep., 1985.

[McC09] D. McCarthy, “Coordinated universal time (utc),” 18th meeting of the CCTF, Jun. 2009. [Online]. Available: https://www.bipm.org/cc/CCTF/Allowed/18/CCTF_09-32_noteUTC.pdf

[McI90] J. McIlroy, “Network radio - new time and frequency distribution system,” in *ENG INF*, no. 40, 1990, p. 5.

[Mil99] “Millennium bug: Immer mehr störungen werden bekannt,” *Spiegel Online*, 1999. [Online]. Available: http://www.spiegel.de/netzwelt/tech/millennium-bug-immer-mehr-stoerungen-werden-bekannt-a-58093.html

[MMBK10] D. Mills, J. Martin, J. Burbank, and W. Kasch, “Network time protocol version 4: Protocol and algorithms specification,” Tech. Rep., 2010.

[Mun99] “München vom millennium bug heimgesucht,” *Spiegel Online*, 1999. [Online]. Available: http://www.spiegel.de/netzwelt/tech/jahr-2000-vorbereitung-muenchen-vom-millennium-bug-heimgesucht-a-56756.html

[MZ01] G. v. R. Moshe Zadka, “Pep 237 – unifying long integers and integers,” Python Software Foundation, Tech. Rep., 2001. [Online]. Available: https://www.python.org/dev/peps/pep-0237/

[NK02] C. Newman and G. Klyne, “Date and time on the internet: Timestamps (rfc 3339),” *The Internet Society.*, Jul. 2002. [Online]. Available: https://tools.ietf.org/html/rfc3339#page-8

[NYHR05] C. Neuman, T. Yu, S. Hartman, and K. Raeburn, “The kerberos network authentication service (v5),” ietf.org, Jul. 2005. [Online]. Available: https://tools.ietf.org/rfc/rfc4120.txt

[Ols09] A. D. Olson, “Sources for time zone and daylight saving time data,” ucla.edu, May 2009. [Online]. Available: http://web.cs.ucla.edu/~eggert/tz/tz-link.htm

[Ope18] “Seconds since the epoch,” The Open Group Base Specifications Issue 7, 2018. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xbd_chap04.html#tag_21_04_16

[Sch18] Schwern, “How important is local time for security?” StackExchange, Sep. 2018. [Online]. Available: https://security.stackexchange.com/a/193679/3286

[Sei05] P. K. Seidelmann, *Explanatory supplement to the astronomical almanac*. University Science Books, 2005.

[Sta05] I. Standard, “Programming languages—c,” *INTERNATIONAL STANDARD ISO/IEC*, vol. 9899, 2005. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[Ste07] D. D. P. Stern, “The sundial,” planetary.org, 2007. [Online]. Available: https://www-spof.gsfc.nasa.gov/stargaze/Sundial.htm

[Sus12a] N. Sussman, “Falsehoods programmers believe about time,” Infinite Undo! Blog, Jun. 2012. [Online]. Available: https://infiniteundo.com/post/25326999628/falsehoods-programmers-believe-about-time

[Sus12b] ——, “More falsehoods programmers believe about time; “wisdom of the crowd” edition,” Infinite Undo! Blog, Jun. 2012. [Online]. Available: https://infiniteundo.com/post/25509354022/more-falsehoods-programmers-believe-about-time

[Tay14] A. Taylor, “The surprising political importance of crimea’s shift to moscow time,” The Washington Post, Mar. 2014.

[TM99] K. Terplan and P. A. Morreale, *The telecommunications handbook*. CrC Press, 1999.

[Tom15] W. W. Tomlinson, *The North Eastern Railway: its rise and development*. A. Reid, limited, 1915.

[vD16] C. von Delbrück, “Bekanntmachung über die vorverlegung der stunden während der zeit vom 1. mai bis 30. september 1916.” in *Reichs-Gesetzblatt*. Reichsamt des Inneren, Apr. 1916, no. 67, p. 243.

[Wei12] H. Weibel, “IEEE 1588 standard for a precision clock synchronization protocol and synchronous ethernet,” Nov. 2012. [Online]. Available: http://www.in2p3.fr/actions/formation/Numerique12/IEEE_1588_Tutorial_IN2P3_Handout.pdf

[Wik18a] Wikipedia contributors, “Daylight saving time by country,” Wikipedia, Aug. 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Daylight_saving_time_by_country&oldid=856565108

[Wik18b] ——, “Gregorian calendar,” Wikipedia, Sep. 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Gregorian_calendar&oldid=857369994

[Wil89] G. Wilkins, “The iau style manual,” *IAU Transactions XXB S*, vol. 23, 1989. [Online]. Available: https://www.iau.org/publications/proceedings_rules/units/

[Win] G. M. R. Winkler, “Modified julian date.” [Online]. Available: https://tycho.usno.navy.mil/mjd.html

APPENDIX

LEAP SECONDS

The time it takes earth to rotate once is not constant [IER13] and in average closer to 86400.002s than to $86400 = 24{\cdot}60{\cdot}60$s. As tidal forces slow it down over time, UT1 keeps getting slower while TAI is always the same length. In order to keep the difference between UT1 and TAI below 0.9 seconds, the International Earth Rotation and Reference Systems Service (IERS) announces leap seconds.

Since the introduction of the leap second concept in 1972 until December 2018, 28 leap seconds were added [Leab]. The extra second is displayed on UTC clocks as 23:59:60. Due to the fact that leap seconds cannot be calculated in advance, but are announced about 6 months before they happen, it is not possible to accurately calculate the distance to an UTC timestamp more than 6 months in the future.

Another issue leap seconds can cause is by their implementation: Most developers assume that the seconds part of a timestamp is in $\{0, \ldots, 59\}$, hence having a value of 60 is problematic. Another possible implementation is to repeat second 59, hence effectively making it last two seconds. This would mean the instant serialization format `YYYY-MM-DDTHH:mm:ss.sssZ` as introduced in Section XIII-B would become ambiguous. Google implemented a third solution: Leap Smearing [Goo18]. The idea is to change the duration of all seconds within several hours instead of adding a leap second. Amazon uses leap smearing as well [Hun17].

TIME DILATION

Time dilation is the effect that the elapsed time measured by two observers is different. This can either be due to a velocity difference relative to each other or due to being differently situated relative to a gravitational field. This effect always happens, but is in most cases not noticeable as the effect is too small and thus hidden by the inaccuracies of the used clocks.

One application where it can be seen are global navigation satellite systems (GNSS) such as GPS. GPS works by 31 satellites in orbit so that at any point on earth at least 4 satellites are visible. The GPS-satellites all have atomic clocks. Their signals contain the exact time when the signal was send. As the satellites move at approximately $14\,000\,\mathrm{km\,h^{-1}}$ they gradually fall behind clocks on earth-based receivers:

$$t = T_0 \cdot \left(\frac{1}{\sqrt{1 - \frac{v^2}{c^2}}} - 1\right) = T_0 \cdot 8.4135 \cdot 10^{-5}$$

Each day, it will be approximately $+7\,\mu\mathrm{s}$.

Time dilation does not only happen when an object has high speed, but also when an object is closer to something heavy. For example, on earth clocks tick slower than on the satellites which are in an orbit of $20\,180\,\mathrm{km}$.

POSSIBLE ATTACKS BY TIME MANIPULATION

Offline: Local System Time Manipulation

Some mobile games have counters which define when you get resources. The simplest way to implement this is by using the current system time and comparing it with a stored value of the past system time.

The same attack is used to prolong 30-day trial software.

This attack can be prevented by loading the system time from a server through a secured connect.

NTP and Certificate Validation

SSL certificates have a time when they become valid and when they expire. If an attacker gets an old certificate and can manipulate the time on the machine, they can make the local machine think it is valid.

Replay-Attacks

Network protocols such as the Kerberos authentication protocol [NYHR05] contain timestamps to prevent replay attacks.

Log poisoning

An attacker which has control over the clock than change where a log entry appears.

Random Number Generators

Generators for pseudo-random numbers need a seed to start. This seed can be based on the local system time. If an attacker has control over it, he can control the output of the random number generator [Sch18].

Manipulation of UUID Generation

UUID version 1 and version 2 take a timestamp as input for the generation of IDs [LMS05]. As pointed out in [Sch18], an attacker could generate a UUID which is equal to an identifier to an administrator and thus gain more privileges.

Periodic Jobs

Another potential issue if an attacker can manipulate time pointed out by [Sch18] is changing the behavior of periodic jobs. Either an attacker can cause them to run much more often and at times where the server load is higher, or the attacker can prevent such jobs to execute potentially critical maintenance tasks. A Watchdog is one example of such a periodic job that can be system-critical.

Date, Time and Datetime Parsing

Parsing anything related to dates is highly ambiguous. For example, the format `06/04/03` could mean

- 2016-04-03
- 1916-04-03
- 2003-04-06
- 2003-06-04

Besides those inherent ambiguities, JavaScript adds more:

```
// Strings passed to Date are treated as a year
d1 = new Date("0");
// Sat Jan 01 2000 00:00:00 GMT+0100 (CET)

// Numbers passed to Date are treated
// as a timestamp
d2 = new Date(0);
// Thu Jan 01 1970 01:00:00 GMT+0100 (CET)

// The first number is treated as the year,
// the second as the month
d3 = new Date(0, 0);
// Mon Jan 01 1900 00:00:00 GMT+0100 (CET)

d4 = new Date(0, 0, 0);
//Sun Dec 31 1899 00:00:00 GMT+0100 (CET)
```

Programming Languages and Software Packages for Time Zones

As a developer how creates software which does in some way use local time, it is important to understand how time zone updates come into the software.

In order to test them, Table II can be used. Depending on the use case, the following combinations should be tested:

- T1 and T2: The local day 2011-12-30 does not exist for the `Pacific/Apia` timezone. Similarly for T3 and T4, the local time `2018-03-25 02:30:00` does not exist for `Europe/Germany`.
- T3 and T4: Although only one second passed, more than an hour passed in the local time.
- T6 and T7: Two different instants map to the same local time.

	UTC	Timezone	Local Datetime
T1	2011-12-30 09:59:00	Pacific/Apia	2011-12-29T23:59:00-10:00
T2	2011-12-30 10:00:00	Pacific/Apia	2011-12-31T00:00:00+14:00
T3	2018-03-25 00:59:59	Europe/Germany	2018-03-25T01:59:59+01:00
T4	2018-03-25 01:00:00	Europe/Germany	2018-03-25T03:00:00+02:00
T5	2017-10-29 01:00:00	Europe/Germany	2017-10-29T02:00:00+01:00
T6	2017-10-29 00:59:59	Europe/Germany	2017-10-29T02:59:59+02:00
T7	2017-10-29 01:59:59	Europe/Germany	2017-10-29T02:59:59+01:00

Table II: Example values for UTC and local time

The Linux operating system makes use of the IANA time zone database. It's a package called `tzdata` on Debian, Ubuntu and Arch Linux. Those packages are updated several times a year.

In **Python**, the core module `datetime` should be used if possible. A notable exception are time zones which are handled with `pytz` and dates before 1 AD or after the year 9999 need other packages.

In **JavaScript**, the problems around the language core have been addressed with libraries. `Sugar.js` can be used for date parsing from natural language and `Moment.js` for all of the rest. The Moment Timezone package directly includes the IANA timezone database.

In **.NET**, nodatime is recommended.

PHP uses its `timezonedb` internal module https://pecl.php.net/package/timezonedb.

Java has an internal Date class which which had issues that were adressed by *Joda Time*. With Java SE 8, users are asked to migrate to `java.time` (JSR-310). Time4J is another possibility. The JRE directly contains the timezone data, but it can be updated with TZUpdater.

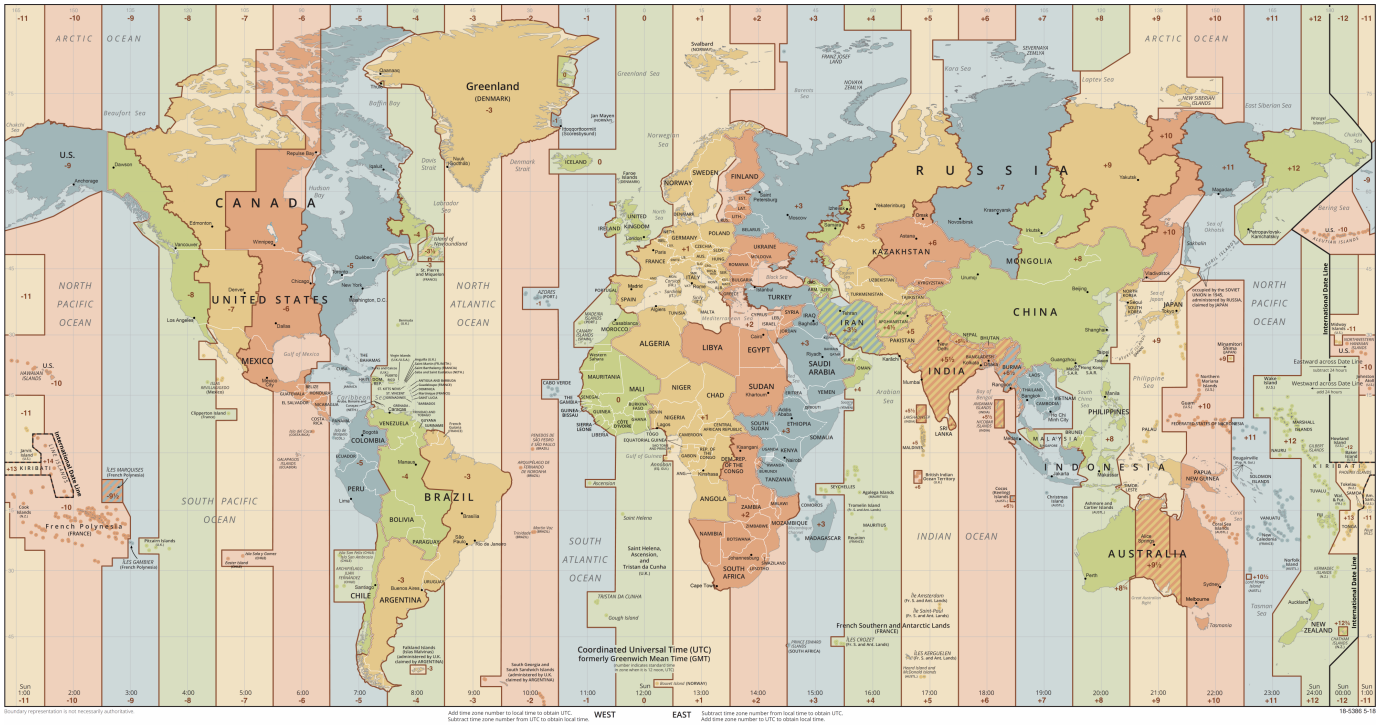


Figure 3: A map of the world's UTC offsets.

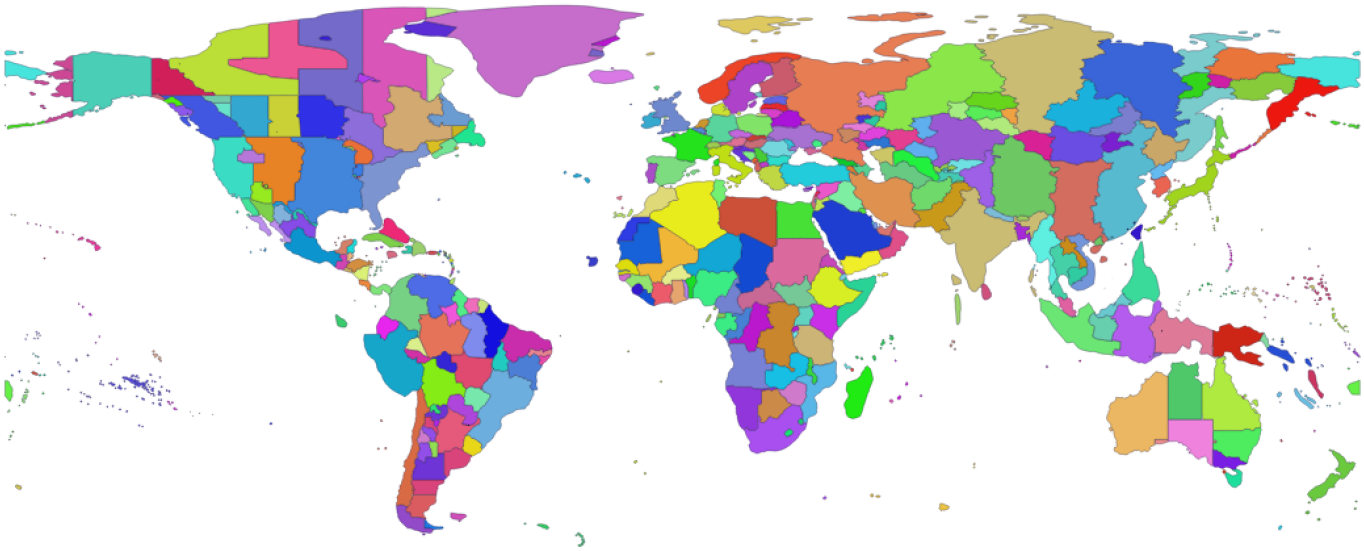


Figure 4: A map of the world's time zones as of the 2017a release of the timezone database. Author: Evan Siroky

COMPARATIVE TIME-TABLE,

SHOWING THE TIME AT THE PRINCIPAL CITIES OF THE UNITED STATES.

COMPARED WITH NOON AT WASHINGTON, D. C.

There is no "Standard Railroad Time" in the United States or Canada; but each railroad company adopts independently the time of its own locality, or of that place at which its principal office is situated. The inconvenience of such a system, if system it can be called, must be apparent to all, but is most annoying to persons strangers to the fact. From this cause many miscalculations and misconnections have arisen, which not unfrequently have been of serious consequence to individuals, and have, as a matter of course, brought into disrepute all Railroad Guides, which of necessity give the local times. In order to relieve, in some degree, this anomaly in American railroading, we present the following table of local time, compared with that of Washington, D. C.

NOON AT WASHINGTON, D. C.		NOON AT WASHINGTON, D. C.		NOON AT WASHINGTON, D. C.	
Albany, N. Y.	12 14 P.M.	Indianapolis, Ind.	11 26 A.M.	Philadelphia, Pa.	12 08 P.M.
Augusta Ga.	11 41 A.M.	Jackson, Miss.	11 08 "	Pittsburg, Pa.	11 48 A.M.
Augusta, Me.	11 31 "	Jefferson, Mo.	11 00 "	Plattsburg, N. Y.	12 15 P.M.
Baltimore, Md.	12 02 P.M.	Kingston, Can.	12 02 P.M.	Portland, Me.	12 28 "
Beaufort, S. C.	11 47 A.M.	Knoxville, Tenn.	11 33 A.M.	Portsmouth, N. H.	12 25 "
Boston, Mass.	12 24 P.M.	Lancaster, Pa.	12 03 P.M.	Pra. du Chien, Wis.	11 04 A.M.
Bridgeport, Ct.	12 16 "	Lexington, Ky.	11 31 A.M.	Providence, R. I.	12 23 P.M.
Buffalo, N. Y.	11 53 A.M.	Little Rock, Ark.	11 00 "	Quebec, Can.	12 23 "
Burlington, N. J.	12 09 P.M.	Louisville, Ky.	11 26 "	Racine, Wis.	11 18 A.M.
Burlington, Vt.	12 16 "	Lowell, Mass.	12 23 P.M.	Raleigh, N. C.	11 53 "
Canandaigua, N. Y.	11 59 A.M.	Lynchburg, Va.	11 51 A.M.	Richmond, Va.	11 58 "
Charleston, S. C.	11 49 "	Middletown, Ct.	12 18 P.M.	Rochester, N. Y.	11 57 "
Chicago, Ill.	11 18 "	Milledgeville, Ga.	11 35 A.M.	Sacketts H'bor, NY.	12 05 P.M.
Cincinnati, O.	11 31 "	Milwaukee, Wis.	11 17 A.M.	St. Anthony Falls,	10 56 A.M.
Columbia, S. C.	11 44 "	Mobile, Ala.	11 16 "	St. Augustine, Fla.	11 42 "
Columbus, O.	11 36 "	Montpelier, Vt.	12 18 P.M.	St. Louis, Mo.	11 07 "
Concord, N. H.	12 23 P.M.	Montreal, Can.	12 14 "	St. Paul, Min.	10 56 "
Dayton, O.	11 32 A.M.	Nashville, Tenn.	11 21 A.M.	Sacramento, Cal.	9 02 "
Detroit, Mich.	11 36 "	Natchez, Miss.	11 03 "	Salem, Mass.	12 26 P.M.
Dover, Del.	12 06 P.M.	Newark, N. J.	12 11 P.M.	Savannah, Ga.	11 44 A.M.
Dover, N. H.	12 37 "	New Bedford, Mass.	12 25 "	Springfield, Mass.	12 18 P.M.
Eastport, Me.	12 41 "	Newburg, N. Y.	12 12 "	Tallahassee, Fla.	11 30 A.M.
Frankfort, Ky.	11 30 A.M.	Newburyport, Ms.	12 25 "	Toronto, Can.	11 51 "
Frederick, Md.	11 59 "	Newcastle, Del.	12 06 "	Trenton, N. J.	12 10 P.M.
Fredericksburg, Va.	11 58 "	New Haven, Conn.	12 17 "	Troy, N. Y.	12 14 "
Frederickton, N. Y.	12 42 P.M.	New London, "	12 20 "	Tuscaloosa, Ala.	11 18 A.M.
Galveston, Texas	10 49 A.M.	New Orleans, La.	11 08 A.M.	Utica, N. Y.	12 08 P.M.
Gloucester, Mass.	12 26 P.M.	Newport, R. I.	12 23 P.M.	Vandalia, Ill.	11 18 A.M.
Greenfield, "	12 18 "	New York, N. Y.	12 12 "	Vincennes, Ind.	11 19 "
Hagerstown, Md.	11 58 A.M.	Norfolk, Va.	12 03 "	Wheeling, Va.	11 45 "
Halifax, N. S.	12 54 P.M.	Northampton, Ms.	12 18 "	Wilmington, Del.	12 06 P.M.
Harrisburg, Pa.	12 01 "	Norwich, Ct.	12 20 "	Wilmington, N. C.	11 56 A.M.
Hartford, Ct.	12 18 "	Pensacola, Fla.	11 20 A.M.	Worcester, Mass.	12 21 P.M.
Huntsville, Ala.	11 21 A.M.	Petersburg, Va.	11 59 "	York, Pa.	12 02 "

By an easy calculation, the difference in time between the several places above named may be ascertained. Thus, for instance, the difference of time between New York and Cincinnati may be ascertained by simple comparison, that of the first having the Washington noon at 12 12 P. M., and of the latter at 11 31 A. M.; and hence the difference is 43 minutes, or, in other words, the noon at New York will be 11.17 A. M. at Cincinnati, and the noon at Cincinnati will be 12 43 P. M. at New York. Remember that places *West* are "slower" in time than those *East*. and *vice versa*.

Figure 5: Local times in 1857 for rail passengers.

DATETIME FORMATTING

Many languages support the following strings for formatting datetime, as specified in [Ope18].

Formatter		Meaning	Example 1	More examples
%%		A literal '%' character.	%	
%Z		Time zone name (empty string if the object is naive).	UTC	-10
%z		UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).	+0000	+0200, -1000
%x		Locale's appropriate date representation.	01/03/19	
%X		Locale's appropriate time representation.	18:19:34	
%c		Locale's appropriate date and time representation.	Thu Jan 3 18:19:34 2019	
%Y		Year with century as a decimal number.	2018	1, …, 10 000
%y		Year without century as a zero-padded decimal number.	18	00, …, 99
%b		Month as locale's abbreviated name.	Sep	Jan, …, Oct, Nov, Dec
%B		Month as locale's full name.	September	October, Oktober
%m	†	Month as a zero-padded decimal number.	09	01, 02, 03, …, 12
%W	†Ⓢ	Week number of the year. Monday as the first day of the week.	52	00, …, 51
%U	†Ⓢ	Week number of the year. Sunday as the first day of the week.	12	00, …, 51
%a		Weekday as locale's abbreviated name.	Mon	Tue, Wed, Thu, Fri, Sat, Sun
%A		Weekday as locale's full name.	Monday	Tuesday, Wednesday
%w		Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0	1, 2, 3, 4, 5, 6
%d	†	Day of the month as a zero-padded decimal number.	30	01, 02, …, 31
%j	†	Day of the year as a zero-padded decimal number.	243	000, …, 366
%p		Locale's equivalent of either AM or PM.	AM	AM,PM
%H	†	Hour (24-hour clock) as a zero-padded decimal number.	20	01, …, 24
%I	†	Hour (12-hour clock) as a zero-padded decimal number.	08	01, …, 12
%M	†	Minute as a zero-padded decimal number.	10	00, …, 59
%S	†	Second as a zero-padded decimal number.	05	00, …, 59

†: Zero Padding §: A decimal number Ⓢ: All days in a new year preceding the first day of the week are considered to be in week 0.

Falsehood Programmer Believes about time

The following contains the falsehood believes stated in [Sus12a] and possible reasons why they are false:

1) There are always 24 hours in a day.
   → Due to DST, countries like Germany have one day in the year with 23 and one with 25 hours.
2) Months have either 30 or 31 days.
   → February has 28 or 29 days.
3) Years have 365 days.
   → Leap years have 366 days.
4) February is always 28 days long.
   → February has 28 or 29 days.
5) Any 24-hour period will always begin and end in the same day (or week, or month).
   → Not when DST is and not when you start at any time different from 00:00.
6) A week always begins and ends in the same month.
   → Similar to Item 5, weeks don't align with borders of the month.
7) A week (or a month) always begins and ends in the same year.
   → Similar to Item 6, weeks don't align with borders of the year.
8) The machine that a program runs on will always be in the GMT time zone.
   → My current machine is on CEST. On Linux systems, you can check it with `date +"%Z %z"`
9) Ok, that's not true. But at least the time zone in which a program has to run will never change.
   → The user could change it, for example because of political disputes as in Israel.
10) Well, surely there will never be a change to the time zone in which a program hast to run in production.
11) The system clock will always be set to the correct local time.
    → Clock drift makes this hard.
12) The system clock will always be set to a time that is not wildly different from the correct local time.
    → The operating system or the software being used might not know about time zone changes, such as the one in Samoa in 2011.
13) If the system clock is incorrect, it will at least always be off by a consistent number of seconds.
    → Due to clock drift all clocks will go wrong. The amount by which they are wrong will also change.
14) The server clock and the client clock will always be set to the same time.
    → This is hard to achieve due to different time zones, simple clock errors and potential hacks. For example, some people change their clocks by five minutes to not be late.
15) The server clock and the client clock will always be set to around the same time.
    → A persons internal clock can just be broken. If it is set to 1970-01-01, this is one indicator. Or the client can try to manipulate the server.
16) Ok, but the time on the server clock and time on the client clock would never be different by a matter of decades.
    → Same as for Item 15 - manipulation can happen.
17) If the server clock and the client clock are not in sync, they will at least always be out of sync by a consistent number of seconds.
    → Clock drift
18) The server clock and the client clock will use the same time zone.
19) The system clock will never be set to a time that is in the distant past or the far future.
    → Testing can be a reason to do it. Or users trying to force undesirable behavior.
20) Time has no beginning and no end.
    → The Unix Timestamp begins on 2018-01-01 00:00:00 UTC and ends in 2038 for 32-bit systems.
21) One minute on the system clock has exactly the same duration as one minute on any other clock.
    → Clock drift is the simple reason why this is wrong. Deliberately slowing down clocks such as for GPS satellites to counter relativity is the other explanation.
22) Ok, but the duration of one minute on the system clock will be pretty close to the duration of one minute on most other clocks.
    → There was a bug on KVM on CentOS. See [Sus12a] for details.
23) Fine, but the duration of one minute on the system clock would never be more than an hour.
    → This can happen due to DST. Then the clocks switch from 01:59:59 to 03:00:59 within a minute. And of course the bug mentioned in Item 22.
24) You can't be serious.
25) The smallest unit of time is one second.
26) Ok, one millisecond.
27) It will never be necessary to set the system time to any value other than the correct local time.
28) Ok, testing might require setting the system time to a value other than the correct local time but it will never be necessary to do so in production.
29) Time stamps will always be specified in a commonly-understood format like 1339972628 or 133997262837.
    → 091630Z JUL 11 represents 2011-07-09T16:30Z in date-time group.

30) Time stamps will always be specified in the same format.
→ The developers can change their mind. For example, in the 1980s, a YY format might have made sense. With the year 2000 approaching developers could have changed their mind.

31) Time stamps will always have the same level of precision.
→ Unix-Timestamps come seconds level precision and in millisecond-level precision.

32) A time stamp of sufficient precision can safely be considered unique.
→ The time 2017-10-29 02:04:55 in the time zone Europe/Berlin is inherently non-unique due to DST. The hour from 2am to 3am exists twice. No level of precision changes that it is duplicated. The information if it is DST or not is necessary to solve this ambiguity.

33) A timestamp represents the time that an event actually occurred.
→ Timestamps are calculated based on a computers local clock. This clock likely is not completely synchronized with TAI, meaning it is not the time when it occurred. As pointed out by [kul18], in distributed systems this problem is faced.

34) Human-readable dates can be specified in universally understood formats such as 05/07/11.
→ This format is not universally understood. Some people use DD/MM/YY and others use MM/DD/YY. The best solution is `YYYY-MM-DD HH:mm:ss+Z`.

The follwoing datetime fallacies are from [Sus12b]. Answers were added:

1) The offsets between two time zones will remain constant.
→ No, due to DST and due to political considerations such as the one by Samoa in 2011.

2) OK, historical oddities aside, the offsets between two time zones won't change in the future.

3) Changes in the offsets between time zones will occur with plenty of advance notice.
→ See [Joh16b]: The Egyptian Cabinet announced on April 29th, 2016 that daylight saving time was to take effect starting July 7th. On June 27th the Egyptian Parliament voted to abolish daylight saving time completely.

4) Daylight saving time happens at the same time every year.
→ DST is applied at the last Sunday in March in Germany at 2am. This means the calendar day changes. It is also important to note that it was applied in April between 1916 and 1945.

5) Daylight saving time happens at the same time in every time zone.
→ For Germany, it is the last Sunday in March at 01:00 UTC. For Mexico it is the first Sunday in April.

6) Daylight saving time always adjusts by an hour.
→ On Lord Howe Island (Australia) clocks are set only 30 minutes forward from LHST (UTC+10:30) to LHDT (UTC+11) during DST.

7) Months have either 28, 29, 30, or 31 days.
→ The day after September 2, 1752 in the UK was September 14 as regulated in the Calendar (New Style) Act 1750. This was when the UK switched from Julian Calendar to the Gregorian Calendar. Hence the UK skipped 11 days in September 1752 and thus had a shorter calendar year.

8) The day of the month always advances contiguously from N to either N+1 or 1, with no discontinuities.
→ See Item 7

9) There is only one calendar system in use at one time.
→ On Bali, three calendar systems are in use: (1) The Gregorian Calendar, (2) the Balinese pawukon calendar, and (3) Balinese saka calendar

10) There is a leap year every year divisible by 4.
→ Except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400.

11) Non leap years will never contain a leap day.
→ A full rotation of the earth around the sun does not always take the same time. It averages around $365.242\,19$ days. Including the leap year rules, the Gregorian calendar has 365.2425 days, meaning it is 0.00031 days too fast. After 3225 years this sums up to a day. As the Gregorian calendar was introduced in 1582 this makes a leap year necessary for the year 4807, assuming it was correct in 1582.

12) It will be easy to calculate the duration of x number of hours and minutes from a particular point in time.
→ It might be impossible, it only a UTC time with offset is given. While it is easy to add the duration to a given UTC time, it is impossible without the timezone to know the local time. The reason are offset changes, most notably by UTC:

13) The same month has the same number of days in it everywhere.
→ See Item 7

14) Unix time is completely ignorant about anything except seconds.

15) Unix time is the number of seconds since Jan 1st 1970.
→ Unix time ignores leap seconds.

16) The day before Saturday is always Friday.
→ No, because Samoa changed its time zone from -11 to +13 and skipped Friday.

17) Contiguous time zones are no more than an hour apart.
→ No, because of the International Date Line.

18) Two time zones that differ will differ by an integer number of half hours.
→ No, because Nepal has UTC+0545

19) Okay, quarter hours.
→ Currently, this is true. Historically, Calcutta time had UTC+5:53:20 until 1948.

20) Okay, seconds, but it will be a consistent difference if we ignore DST.
→ We have to ignore DST, leap seconds and changes of the offset due to political or economic reasons.

21) If you create two date objects right beside each other, they'll represent the same time.
→ No, because the command needs time to execute.

22) You can wait for the clock to reach exactly HH:MM:SS by sampling once a second.
→ No, because commands take time to execute and due to leap seconds or DST.

23) If a process runs for n seconds and then terminates, approximately n seconds will have elapsed on the system clock at the time of termination.
→ No, because of DST.

24) Weeks start on Monday.
→ In the US, weeks start on Sundays. Also, the Hebrew calendar starts at Sunday and proceed to Saturday.

25) Days begin in the morning.
→ The Hebrew calendar day starts at sunset.

26) Holidays span an integer number of whole days.
→ The Hebrew calendar defines the day by sunset. This means Holidays don't span an integer amount of standard days.

27) The weekend consists of Saturday and Sunday.
→ For more than 20 countries, including India, Mexico and Egypt, this is false.

28) It's possible to establish a total ordering on timestamps that is useful outside your system.
→ Local clocks can be synchronized via NTP, but there is always a finite accuracy. This means although the locally created UTC timestamps indicate event A happened before event B, due to their finite precision it can be the other way around.

29) The local time offset (from UTC) will not change during office hours.
→ People work at all times (night shifts) and although DST happens often on weekends at night, there is no guarantee that it will always be like this everywhere.

30) `Thread.sleep(1000)` sleeps for 1000 milliseconds.

31) `Thread.sleep(1000)` sleeps for $\geq$ 1000 milliseconds.
→ See https://stackoverflow.com/q/11376307

32) There are 60 seconds in every minute.
→ No, due to leap seconds and DST.

33) Timestamps always advance monotonically.
→ No. When Unix time was invented, leap seconds did not exist.

34) GMT and UTC are the same time zone.
→ No, UTC is a standard.

35) Britain uses GMT.
→ No, in the summer Britain uses BST - British Summer Time.

36) Time always goes forwards.
→ No, due to DST.

37) The difference between the current time and one week from the current time is always $7 \cdot 86400$ seconds.
→ No, due to DST and leap seconds.

38) The difference between two timestamps is an accurate measure of the time that elapsed between them.

39) 24:12:34 is a invalid time.
→ Time notations like this are used to linearize times. This notation makes it easier to calculate how much time has passed.

40) Every integer is a theoretical possible year.
→ The Gregorian calendar started in the year 1582.

41) If you display a datetime, the displayed time has the same second part as the stored time.
→ No, because Time zones can have an offset on the seconds-level.

42) Or the same year.
→ If it is 2018-12-30 23:30:00Z, then it is already the first day of 2019 in Germany.

43) But at least the numerical difference between the displayed and stored year will be less than 2.

44) If you have a date in a correct YYYY-MM-DD format, the year consists of four characters.
→ No, if it is before the year 1000 or after the year 9999. Or if it is before the year -999.

45) If you merge two dates, by taking the month from the first and the day/year from the second, you get a valid date.
→ No, because February only has 28 or 29 days.

46) But it will work, if both years are leap years.
→ No, because one date could be 2018-01-30 and the other one could be 2018-02-26.

47) If you take a W3C published algorithm for adding durations to dates, it will work in all cases.

48) The standard library supports negative years and years above 10000.
→ Python does not support years below 1 and above 9999.

49) Time zones always differ by a whole hour.
→ No, because Nepal has UTC+0545

50) If you convert a timestamp with millisecond precision to a date time with second precision, you can safely ignore the millisecond fractions.
→ No, because of rounding mistakes.

51) But you can ignore the millisecond fraction, if it is less than 0.5.

52) Two-digit years should be somewhere in the range 1900-2099.
→ No, it also could be in 1800-1899.

53) If you parse a date time, you can read the numbers character for character, without needing to backtrack.

54) But if you print a date time, you can write the numbers character for character, without needing to backtrack.

55) You will never have to parse a format like —12Z or P12Y34M56DT78H90M12.345S.

56) There are only 24 time zones.
→ The Wikipedia article "List of time zone abbreviations" lists 199 time zones and the time zone library `pytz` knows about 591 time zones (`pytz.all_timezones`).

57) Time zones are always whole hours away from UTC.
→ No, because Nepal has UTC+0545

58) Daylight Saving Time (DST) starts/ends on the same date everywhere.
→ No, Germany and Mexico differ.

59) DST is always an advancement by 1 hour.
→ In 1933, Singapore added +0:20 hours for DST [Asl12].

60) Reading the client's clock and comparing to UTC is a good way to determine their time zone.
→ No, because of DST.

61) The software stack will/won't try to automatically adjust for time zone/DST.

62) My software is only used internally/locally, so I don't have to worry about time zones.
→ Users want their local time being used on their machine.

63) My software stack will handle it without me needing to do anything special.

64) I can easily maintain a time zone list myself.
→ During this article, it should have become clear that this is a hard task.

65) All measurements of time on a given clock will occur within the same frame of reference.
→ See GPS / time dilatation.

66) The fact that a date-based function works now means it will work on any date.
→ No, because there are so many special cases.

67) Years have 365 or 366 days.
→ See Item 7

68) Each calendar date is followed by the next in sequence, without skipping.
→ See Item 7

69) A given date and / or time unambiguously identifies a unique moment.
→ Due to multiple calendar systems notations like 1752-08-08 are ambiguous. Also, notations like 07/06/17 are highly ambiguous as it could be 2017-06-07, 2017-07-06, 1917-06-07, 1917-07-06, 2007-06-17.

70) Leap years occur every 4 years.
→ Except for years divisible by 100.

71) You can determine the time zone from the state/province.
→ The time zone in the Palestinian territories (West Bank, Gaza Stripe) entered DST in 2011

72) You can determine the time zone from the city/town.
→ When people have different political interests, they can choose different time zones as well. One example is the Russian annexation of Crimea [Tay14].

73) Time passes at the same speed on top of a mountain and at the bottom of a valley.
→ No, due to relativistic time dilation.

74) One hour is as long as the next in all time systems.
→ Today, we mostly use temporal hours: We define one second. Then an hour is simply $60 \cdot 60$ seconds. Another option are seasonal hours. You have sunrise and sunset. Then you divide both, night and day, by 12. Then you have 24 hours. But those hours depend on the time of the year and on the latitude.

75) You can calculate when leap seconds will be added.
→ Leap seconds are necessary because earths rotation is influenced by various factors. On the long run, tidal forces slow down earth rotation. This means UT1 keeps getting slower while TAI is always the same length. In order to keep the difference between UT1 and TAI below 0.9 seconds, the International Earth Rotation and Reference Systems Service (IERS) announces leap seconds.

76) The precision of the data type returned by a `getCurrentTime()` function is the same as the precision of that function.

77) Two subsequent calls to a `getCurrentTime()` function will return distinct results.
→ The system clock could be resetted between the calls.

78) The second of two subsequent calls to a `getCurrentTime()` function will return a larger result.
→ See Item 77.

79) The software will never run on a space ship that is orbiting a black hole.