Geoff Huston
January 2017

# A Postscript to the Leap Second

The inexorable progress of time clocked past the New Year and at 23:59:60 on the 31st December 2016 UTC the leap second claimed another victim. This time Cloudflare described how the Leap Second caused some DNS failures in Cloudflare's infrastructure (https://blog.cloudflare.com/how-and-why-the-leap-second-affected-cloudflare-dns/).

What is going on here? It should not have been a surprise, yet we still see failing systems.

Let's look at time on Unix systems, as I suspect that in the case of the Unix time functions, the system is trying to do meet two goals at once and there are cases where these goals appear to be in conflict. (To be a little clearer, since "Unix" itself is a very broad church, we are specifically referring to the POSIX.1 specification of time, as commonly implemented on Unix platforms.)

The first objective of the time subsystem is to provide a monotonically increasing reference register that is intended to be aligned to standard (SI) seconds. This can be used to measure the duration between two moments in time. The underlying assumption here is that the reference register value increases in a manner such that the seconds register is incremented once every standard (SI) second. Successive calls to this register should always retrieve a value that is no less than previous call on the same register.

The second is to return the calendar time, or in the parlance used by the manual page on FreeBSD: "the value of time in seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time." (man -3 time) The essential, but unstated in this case, qualification is that this count of seconds since the nominated epoch excludes he 27 leap seconds that have occurred since this epoch date.

This second objective certainly facilitates a programmer-friendly way of representing time. Whenever you add 86,400 to a *time()* value you get a date that corresponds to "tomorrow", as all UTC days are exactly 86,400 seconds long. (Let's not talk about local timezones and daylight savings here, as that's a needless complication in this context!)

So how do these systems react to leap seconds? If a leap second is being inserted into the time stream then at the start of the leap second the fractional counter is reset to zero, but the second country is held steady.

| **UTC TIME** | **System Time** |
|---|---|
| 31/12/2016 23:59:59.0 | 1483228799.00 |
| 31/12/2016 23:59:59.5 | 1483228799.50 |
| 31/12/2016 23:59:60.0 | 1483228799.00 |
| 31/12/2016 23:59:60.5 | 1483228799.50 |
| 01/01/2017 00:00:00.0 | 1483228800.00 |

For example, here's an extract of a DNS query log from a Linux Debian system taken across the December 31 leap second (the first column shows the system time of the query):

```
1483228799.906831 query: 0du-results-ub83a85b3.dotnxdomain.net. IN DS -ED
1483228799.942041 query: 0du-results-u745a9cee.dotnxdomain.net. IN A -EDC
1483228799.019576 query: 0di-u4da59be0.dotnxdomain.net. IN AAAA -ED
1483228799.019754 query: 0di-u4da59be0.dotnxdomain.net. IN A -ED
```

If you look at just the integer part of the time value you still see a monotonically increasing sequence of integer values. But if you are looking at a higher level of resolution that includes fractions of a second (such as the *clock_gettime()* call on FreeBSD) that gives a time value as a seconds and fractions counter, the 'compound' fractional time value jumps backward and we have broken the assumption of a monotonically increasing value space for time.

What is going on is that the *time()* function does not in fact report the accumulated count of SI standard seconds since the epoch of 1 January 1970. What this function appears to report is 86,400 times the number of UTC days since the epoch, plus the number of SI seconds since midnight UTC, to a maximum value of 86,400.

How can we fix this?

One approach, the topic of a long-standing debate at the ITU-R, is to remove leap seconds entirely and let the time value drift away from the earth's period of rotation. But something jars about this approach. Midnight would no longer occur on the exact middle of the night, and, over time, the UTC clock would appear to drift forward in time as compared to the rotation of the earth on its own axis.

Can we have the best of both worlds? Why don't we operate a system that use an internal counter that accumulates the exact number of SI seconds since some epoch time. This is essentially the Atomic Time base (TAI). In this framework, all time durations would be exactly equivalent, as measured by system time, and the system time counters would never have to be adjusted to accommodate leap seconds, as TAI time is independent of the earth's astronomical time. If an application wishes to report a time as a calendar time in UTC time it would need to convert the time to UTC time using some form of table-based conversion using a table of leap seconds.

The issue with this second approach is that the conversion between this internal TAI-related time and UTC calendar time is not known for arbitrary dates into the future. Leap Second advice is only published some six months before the change to UTC, as the exact changes in the period of the earth's rotation are not known far into the future. For example, the Unix time value of 2,000,000,000.5 corresponds to the UTC calendar time of 18/05/2033 03:33:20.5. It we define the TAI-related time as the number of SI seconds since the same epoch of 1 January 1970 then we cannot accurately translate this time value SI time value of 2,000,000,000.5 to UTC time. There have been 27 leap seconds between the epoch time and the start of 2017, so this TAI-related time would probably be no later than 18/05/2033 03:32:53.5 (it is possible for leap seconds to be removed from UTC time in response to the earth increasing its rotation speed, but this is a far less likely occurrence than the observed slowing). However, we do not know how many additional leap seconds will be added to, or removed from, UTC between now and 2033, so we cannot accurately map this future SI second count to a future UTC time.

Another way to remove the leap second jump is to use a leap second smear. NTP already uses this to bring a local clock into sync with the NTP time base, by performing a sequence of small changes to the local system clock over time. For example, if we were to alter the system clock by one hundredth of a millisecond every second, then a one second change will be "smeared" over a period of 27 hours and just under 50 minutes. This approach avoids a "jump" in the time base, but at the cost of the assumption that every second is exactly the same second in duration. The other issue here is that there is no single standard way of performing such a "smearing" of a leap second, and a collection of supposedly synchronised systems can fall out of time sync by up to a second as a result of this deliberate adjustment to the local time.

Nirvana here is where all systems treat time in precisely the same way, where system time never goes backward, and where there is an unambiguous mapping between this internal time value and the calendar UTC time for current times and times that extend into the future.

But we can't have all of that. What we have instead is a number of different approaches to handling time, some of which admit sequential time readings that jump backwards, some of which apply a variable definition to the duration of a timed second, and some of which have a very limited view into the future. In some ways in a highly connected environment this variation in behaviour across our computer systems, with its consequent variations in time is more challenging than the single set of compromises made by any single approach. Any time-sensitive application needs to use a code base that can cope with all of these approaches, as it will not know in advance how any particular host system will handle leap seconds.

Inevitably, there will be leap seconds to come in UTC. And there will continue to be some applications that will be surprised when their system time jumps backwards. And other applications will be perplexed when some seconds last longer than others.  And other applications will get confused when their accurate time is different from the supposedly accurate time of their time peers. And at least in the foreseeable future of time, in the styles that our computers have been coded to understand it, nothing is going to change.

## Author

*Geoff Huston* B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region. He has been closely involved with the development of the Internet for many years, particularly within Australia, where he was responsible for building the Internet within the Australian academic and research sector in the early 1990's. He is author of several Internet-related books, and is active in the area of Internet Standards in the IETF. He has worked as an Internet researcher, as an ISP systems architect and a network operator at various times.

*www.potaroo.net*

## Disclaimer