NTP And Windows History
Table of Contents

NTP And Windows History .................................................................................................................. 1
Windows 3.x, 9.x, and ME ................................................................................................................. 1
Windows NT, 2000, XP, and Server 2003 ......................................................................................... 1
Windows Vista, Windows 7, and Server 2008 .................................................................................. 2
Windows 8, Server 2012, and Windows 10 (probably also Server 2016) ...................................... 3
Leap Second Support in ntpd for Windows ....................................................................................... 3
NTP And Windows History

Windows 3.x, 9.x, and ME

Windows 3.x, Windows 9x, and Windows ME were like MS-DOS with a GUI on top. The DOS time was derived from a 1.193181 MHz clock which was divided by a 16 bit counter, so the counter overflow rate was 1.193181 MHz / 65536, i.e. about 18,2065 Hz. So the timer tick interval was about 55 milliseconds, and there was no API provided by the operating system to adjust the system time smoothly. The only way to do so would have been to fiddle with the timer/counter hardware directly.

At that time Meinberg provided a TSR program for DOS in assembly language, which hooked itself into the DOS timer tick interrupt, so that it was periodically activated in the background at each timer tick. Whenever a certain number of timer ticks had passed, the TSR read the time from one of the Meinberg PCI cards (or ISA cards at that time) and set the system time with it, if DOS was not “busy”. This TSR also worked well on these Windows versions.

Windows NT, 2000, XP, and Server 2003

Only with Windows NT a set of new APIs for timekeeping was introduced.

GetSystemTimeAsFileTime() can be used to read a system time stamp as FILETIME structure, which represents the time as a 64 bit number of 100 ns units (hecto-nanoseconds, HNS) since January 1, 1601 (UTC). However, the problem with this call is that even though the FILETIME structure provides 100 ns resolution, the time returned by this API call is always the same during a timer tick interval, and after the next timer tick the returned value jumps by the amount of the timer tick interval. So the system time can't be read very precisely using this call.

A program like ntpd in the role of a time client takes one time stamp of its system time when it sends a request packet to a server, and another time stamp when it receives the reply from the server. However, the difference between the 2 time stamps was either just 0, if the reply arrived during the same timer tick interval, or even some milliseconds according to the timer tick interval, if a timer tick interrupt occurred between the calls, even if the time stamps were taken only a few microseconds after each other.

The same happened when ntpd was acting as a time server, and one time stamp was taken when an NTP request packet came in, and another one when the reply packet to the client went out.

The time stamp interval in Windows NT and Server 2003 is usually 15.625 ms, so computations based on these coarse time stamps are not very precise, especially if you take into account that ntpd usually considers a 128 ms offset as large enough to step the system time instead of slewing it.

This is why ntpd tried to extrapolate the time between two timer ticks using the Windows QueryPerformanceCounter() (QPC) API call and a Windows timer callback function. This worked well if the Windows kernel called the registered callback function always quickly after a timer tick had occurred, but this could not be controlled by ntpd, and could vary depending on the system load.
Also, the QPC function could use any of the timer chips available in the CPU or on the mainboard. Many of the CPUs which were current at that time had time stamp counters (TSCs) which were not synchronized across several cores in the same physical CPU chip, and the TSC clock frequency was bound to the CPU clock frequency, which could change whenever the CPU was clocked down for power saving. So this could lead to to strange results of the time extrapolation, and eventually you had to configure Windows manually to use another timer circuit for the QPC API, e.g. the ACPI power management timer (PMTIMER).

The API calls introduced with Windows NT to adjust the system time smoothly were GetSystemTimeAdjustment() which returns the standard time adjustment value as well as the value which is currently in effect, and SetSystemTimeAdjustment() which can be used to set a new time adjustment value.

The standard time adjustment value is determined by Windows at startup, and time synchronization software can add some number to the standard value to make the system time increase faster, or subtract some value to make it slower.

For example, under Windows XP the standard timer tick interval was 156250 HNS, or 15.625 ms, which corresponds to 1/64 Hz. If SetSystemTimeAdjustment() is called with 156250+1 then the system time gains 100 ns per 15.625 ms until another values is passed to the kernel. So if a correction of +1 is too fast for very accurate adjustments you can only do some kind of “pulse width modulation”, i.e. set the time adjustment to 156250+1 for a certain time interval, then revert back to 156250+0 which is too slow, then again set it to 156250+1 again, and so on.

Windows has the concept of a multimedia timer, which was required to play audio or video contents well. Any application can ask Windows to set the “Multimedia Timer” to highest resolution, and if an application did this first then the timing inside the Windows kernel was obviously switched to a higher rate, e.g. a 1 ms tick interval, and the switching caused the extrapolated Windows system time to jump by up to more than 100 ms. When the last application which had used the MM timer was terminated then the kernel restored the timer tick interval to the default value, and the extrapolated time seemed to step back by the same amount as it had stepped forth before. This is why ntpd has a command line option -M which lets ntpd itself set the MM timer to highest resolution, so it isn't affected anymore if another application also does this.

Windows Vista, Windows 7, and Server 2008

Starting with Windows Vista the real timer tick was obviously decreased from about 16 ms to 1 ms. This is also the resolution which you can get with the callback function mentioned above, so current versions of ntpd don't use the extrapolation feature the program determines that the time increments in 1 ms steps or less.

However, also in Windows Vista a bug was introduced, which is also in Windows 7, and in Windows Server 2008: “SetSystemTimeAdjustment May Lose Adjustments Less than 16” https://support.microsoft.com/de-de/kb/2537623

This means if the time synchronization software makes small time adjustments to slew the system time smoothly then this has no effect at all. As a consequence the control loop in ntpd could get unstable. Current versions of ntpd (namely 4.2.8+) have a workaround for this bug which makes
adjustment always by at least 16, but eventually only for a short period of time.

**Windows 8, Server 2012, and Windows 10 (probably also Server 2016)**

Starting with Windows 8 another API call `GetSystemTimePreciseAsFileTime()` was introduced. This call also returns a `FILETIME` structure, but should really return time stamps with 100 ns resolution, so time differences against refclocks or time stamps in NTP packets can be computed much more precisely.

If you distribute a precompiled binary for Windows then you don't know on which Windows version the executable will finally be run, so `ntpd` checks at startup if this new API call is supported, and uses it, and otherwise falls back to using the legacy function.

Unfortunately there is still no new API call to apply a time adjustment, so this is still limited to 100 ns per ~15 ms, and the controlling program has to take care that adjustments are canceled when the computed time offset has been compensated.

**Leap Second Support in ntpd for Windows**

Even in Windows 10 there is **still no leap second support** in the Windows kernel, so `ntpd` has another built-in workaround to handle this. To insert a leap second, the approach is to reduce the timer tick value for an interval of 2 seconds such that the Windows system time increases only “half speed”, so after 2 real seconds have passed the system time has only gained one second, and thus is aligned again with the UTC time after the inserted leap second.

— Martin Burnicki 2016-09-02 11:41