GPSD Time Service HOWTO

Gary E. Miller <gem@rellim.com>, Eric S. Raymond <esr@thyrsus.com>
version 2.11, June 2018

Table of Contents

[Introduction](#)
[NTP with GPSD](#)
[GPS time](#)
[1PPS quality issues](#)
[Software Prerequisites](#)
- [Building gpsd ==](#)
- [Kernel support](#)
- [Time service daemon](#)
- [NTPSec](#)

[Choice of Hardware](#)
[Enabling PPS](#)
[Running GPSD](#)
[Feeding NTPD from GPSD](#)
[Feeding chrony from GPSD](#)
[Performance Tuning](#)
- [ARP is the sound of your server choking](#)
- [Watch your temperatures](#)
- [Powersaving is not your friend](#)

[NTP tuning and performance details](#)
- [NTP performance tuning](#)
- [Polling Interval](#)

[Chrony performance tuning](#)
[Providing local NTP service using PTP](#)
- [PTP with software timestamping](#)
- [PTP with hardware timestamping](#)

[Providing public NTP service](#)
[Acknowledgments](#)
[References](#)
[Changelog](#)

This document is mastered in asciidoc format. If you are reading it in HTML, you can find the original at the GPSD project website.

Introduction

GPSD, NTP and a GPS receiver supplying 1PPS (one pulse-per-second) output can be used to set up a high-quality NTP time server. This HOWTO explains the method and various options you have in setting it up.

Here is the quick-start sequence. The rest of this document goes into more detail about the steps.

1. Ensure that gpsd and either ntpd or chronyd are installed on your system. (Both gpsd and ntpd are pre-installed in many stock Linux distributions; chronyd is normally not.) You don't have to choose which to use yet if you have easy access to both, but knowing which alternatives are readily available to you is a good place to start.
2. Verify that your gpsd version is at least 3.17. Many problems are caused by the use of old versions. When in doubt, reinstall gpsd from the upstream source. Many distributions ship old and/or broken versions of gpsd.
3. Connect a PPS-capable GPS receiver to one of your serial or USB ports. A random cheap consumer-grade GPS receiver won't do; you may have to do some hunting to find a usable one.
4. Check that it actually emits PPS by pointing GPSD's gpsmon utility at the port. If it has a good (3D-mode) fix, lines marked "PPS" should scroll by in the packet-logging window. A new device out of the box may take up to 30 minutes for the first 3D fix. If gpsmon shows a 3D fix, but does not show PPS lines, try running ppscheck.
5. If you persistently fail to get live PPS, (a) you may have a skyview problem, (b) you may have a cabling problem, (c) your GPS may not support PPS, (d) you may have a gpsd or kernel configuration problem, (e) you may have a device problem, (e) there may be a bug in the core GPSD code used by gpsmon. These are listed in roughly decreasing probability. Troubleshoot appropriately.
6. Edit your ntpd or chronyd configuration to tell your NTP daemon to listen for time hints. (This step is somewhat tricky.)
7. Start up gpsd. If you are using ntpd, you can use ipcrm(1) to check that verify that the shared-memory segment that gpsd and ntpd want to use to communicate has been attached; or you can impatiently skip to the next step and look for the segment only if that fails.
8. Use ntpq or the chronyc sources command to verify that your device is feeding time corrections to your NTP daemon.
9. (Optional and challenging.) Hand-tune your installation for the best possible performance.

This document does not attempt to explain all the intricacies of time service; it is focused on practical advice for one specific deployment case. There is an introduction [TIME-INTRO] to basic concepts and terminology for those new to time service. An overview of the NTP protocols can be found at [WIKI-NTP], and the official NTP FAQ [NTP-FAQ] is probably as gentle an introduction to the NTP reference implementation as presently exists.

We encourage others to contribute additions and corrections.

Table 1. Units table

nSec	nanoSecond	1/1,000,000,000 of a second
uSec	microSecond	1/1,000,000 of a second
mSec	milliSecond	1/1,000 of a second

There are a few important terms we need to define up front. **Latency** is delay from a time measurement until a report on it arrives where it is needed. **Jitter** is short-term variation in latency. **Wobble** is a jitter-like variation that is long compared to typical measurement periods. **Accuracy** is the traceable offset from *true* time as defined by a national standard institute.

A good analogy to jitter vs wobble is changes in sea level on a beach. Jitter is caused by wave action, wobble is the daily effect of tides. For a time server, the most common causes of wobble are varying GPS satellite geometries and the effect of daily temperature variations on the oscillators in your equipment.

NTP with GPSD

See [TIME-INTRO] for a technical description of how NTP corrects your computer's clock against wobble. For purposes of this how-to, the important concepts to take way are those of time strata, servers, and reference clocks.

Ordinary NTP client computers are normally configured to get time from one or more Stratum 2 (or less commonly Stratum 3) NTP servers. However, with GPSD and a suitable GPS receiver, you can easily condition your clock to higher accuracy than what you get from typical Stratum 2; with a little effort, you can do better than you can get from most public Stratum 1 servers.

You can then make your high-quality time available to other systems on your network, or even run a public NTP server. Anyone can do this; there is no official authority, and any NTP client may choose to use your host as a server by requesting time from it. The time-service network is self-regulating, with NTP daemons constantly pruning statistical outliers so the timebase cannot be accidentally or deliberately compromised.

In fact many public and widely-trusted Stratum 1 servers use GPS receivers as their reference clocks, and a significant fraction of those use GPSD in the way we will describe here.

GPS time

The way time is shipped from GPS satellites causes problems to beware of in certain edge cases.

Date and time in GPS is represented as number of weeks from the start of zero second of 6 January 1980, plus number of seconds into the week. GPS time is **not** leap-second corrected, though satellites also broadcast a current leap-second correction which is updated on six-month boundaries according to rotational bulletins issued by the International Earth Rotation and Reference Systems Service (IERS).

The leap-second correction is only included in the multiplexed satellite subframe broadcast, once every 12.5 minutes. While the satellites do notify GPSes of upcoming leap-seconds, this notification is not necessarily processed correctly on consumer-grade devices, and may not be available at all when a GPS receiver has just cold-booted. Thus, reported UTC time may be slightly inaccurate between a cold boot or leap second and the following subframe broadcast.

GPS date and time are subject to a rollover problem in the 10-bit week number counter, which will re-zero every 1024 weeks (roughly every 19.6 years). The first rollover since GPS went live in 1980 was in Aug-1999, followed by Apr-2019, the next will be in Nov-2038 (the 32-bit and POSIX issues will probably be more important by then). The new "CNAV" data format extends the week number to 13 bits, with the first rollover occurring in Jan-2137, but this is only used with some newly added GPS signals, and is unlikely to be usable in most consumer-grade receivers currently.

For accurate time reporting, therefore, a GPS requires a supplemental time references sufficient to identify the current rollover period, e.g. accurate to within 512 weeks. Many GPSes have a wired-in (and undocumented) assumption about the UTC time of the last rollover and will thus report incorrect times outside the rollover period they were designed in.

For accurate time service via GPSD, you require three things:

- A GPS made since the last rollover, so its hidden assumption about the epoch will be correct.
- Enough time elapsed since a cold boot or IERS leap-second adjustment for the current leap-second to get update.
- A GPS that properly handles leap-second adjustments. Anything based on a u-blox from v6 onward should be good; the status of SiRFs is unknown and doubtful.

1PPS quality issues

GPSD is useful for precision time service because it can use the 1PPS pulse delivered by some GPS receivers to discipline (correct) a local NTP instance.

It's tempting to think one could use a GPS receiver for time service just by timestamping the arrival of the first character in the report on each fix and correcting for a relatively small fixed latency composed of fix-processing and RS232 transmission time.

At one character per ten bits (counting framing and stopbits) a 9600-bps serial link introduces about a mSec of latency **per character**; furthermore, your kernel will normally delay delivery of characters to your application until the next timer tick, about every 4 mSec in modern kernels. Both USB and RS232 will incur that approximately 5mSec-per-char latency overhead. You'll have to deal with this latency even on chips like the Venus 6 that claim the beginning of their reporting burst is synced to PPS. (Such claims are not always reliable, in any case.)

Unfortunately, fix reports are also delayed in the receiver and on the link by as much as several hundred mSec, and this delay is not constant. This latency varies (wobbles) throughout the day. It may be stable to 10 mSec for hours and then jump by 200mSec. Under these circumstances you can't expect accuracy to UTC much better than 1 second from this method.

For example: SiRF receivers, the make currently most popular in consumer-grade GPS receivers, exhibit a wobble of about 170mSec in the offset between actual top-of-second and the transmission of the first sentence in each reporting cycle.

To get accurate time, then, the in-band fix report from the GPS receiver needs to be supplemented with an out-of-band signal that has a low and constant or near-constant latency with respect to the time of of the fix. GPS satellites deliver a top-of-GPS-second notification that is nominally accurate to 50nSec; in capable GPS receivers that becomes the 1PPS signal.

1PPS-capable GPS receivers use an RS-232 control line to ship the 1PPS edge of second to the host system (usually Carrier Detect or Ring Indicator; GPSD will quietly accept either). Satellite top-of-second loses some accuracy on the way down due mainly to variable delays in the ionosphere; processing overhead in the GPS receiver itself adds a bit more latency, and your local host detecting that pulse adds still more latency and jitter. But it's still often accurate to on the order of 1 uSec.

Under most Unixes there are two ways to watch 1PPS; Kernel PPS (KPPS) and plain PPS latching. KPPS is an implementation of RFC 2783 [RFC-2783]. Plain PPS just references the pulse to the system clock as measured in user space. These have different error budgets.

Kernel PPS uses a kernel function to accurately timestamp the status change on the PPS line. Plain PPS has the kernel wake up the GPSD PPS thread and then the PPS thread reads the current system clock. As noted in the GPSD code, having the kernel do the time stamp yields lower latency and less jitter. Both methods have accuracy degraded by interrupt-processing latency in the kernel serial layer, but plain PPS incurs additional context-switching overhead that KPPS does not.

With KPPS it is very doable to get the system clock stable to ±1 uSec. Otherwise you are lucky to get ±5 uSec, and there will be about 20uSec of jitter. All these figures were observed on plain-vanilla x86 PCs with clock speeds in the 2GHz range.

All the previous figures assume you're using PPS delivered over RS232. USB GPS receivers that deliver 1PPS are rare, but do exist. Notably, there's the Navisys GR-601W/GR-701W/GR-801W [MACX-1]. In case these devices go out of production it's worth noting that they are a trivial modification of the stock two-chip-on-a-miniboard commodity-GPS-receiver design of engine plus USB-to-serial adapter; the GR-[678]01W wires a u-blox 6/7/8 to a Prolific Logic PL23203. To get 1PPS out, this design just wires the 1PPS pin from the GPS engine to the Carrier Detect pin on the USB adapter. (This is known as the "Macx-1 mod".)

With this design, 1PPS from the engine will turn into a USB event that becomes visible to the host system (and GPSD) the next time the USB device is polled. USB 1.1 polls 1024 slots every second. Each slot is polled in the same order every second. When a device is added it is assigned to one of those 1024 polling slots. It should then be clear that the accuracy of a USB 1.1 connected GPS receiver would be about 1 mSec.

As of mid-2016 no USB GPS receiver we know of implements the higher polling-rate options in USB 2 and 3 or the interrupt capability in USB 3. When one does, and if it has the Macx-1 mod, higher USB accuracy will ensue.

Table 2. Summary of typical accuracy

GPS atomic clock	±50nSec
KPPS	±1uSec
PPS	±5uSec
USB 1.1 poll interval	±1mSec
USB 2.0 poll interval	±100µSec (100000 nSec)
Network NTP time	~±30mSec [1]

Observed variations from the typical figure increase towards the bottom of the table. Notably, a heavily loaded host system can reduce PPS accuracy further, though not KPPS accuracy except in the most extreme cases. The USB poll interval tends to be very stable (relative to its 1mSec or 100µSec base).

Network NTP time accuracy can be degraded below RFC5905's "a few tens of milliseconds" by a number of factors. Almost all have more to do with the quality of your Internet connection to your servers than with the time accuracy of the servers themselves. Some negatives:

- Having a cable modem. That is, as opposed to DSL or optical fiber, which tend to have less variable latencies.
- Path delay asymmetries due to peering policy. These can confuse NTP's reconciliation algorithms.

With these factors in play, worst-case error can reach up to ±100mSec. Fortunately, errors of over ±100mSec are unusual and should occur only if all your network routes to servers have serious problems.

Software Prerequisites

If your kernel provides the RFC 2783 KPPS (kernel PPS) API, gpsd will use that for extra accuracy. Many Linux distributions have a package called "pps-tools" that will install KPPS support and the timepps.h header file. We recommend you do that. If your kernel is built in the normal modular way, this package installation will suffice.

Building gpsd ==

A normal gpsd build includes support for interpreting 1PPS pulses that is mostly autoconfiguring and requires no special setup. If the current system supports pps.

You can build a version stripped to the mimimum configuration required for time service. This reduces the size of the binary and may be helpful on embedded systems or for SBCs like the Raspberry Pi, Odroid, or BeagleBone. Only do this if you have serious size contraints, much functionality will be lost.

When gpsd is built with timeservice=yes:

1. The -n (nowait) option is forced: gpsd opens its command-line devices immediately on startup.
2. Forces the building of ntpshmmon, cgps and gpsmon. Those program would be built by default anyway, unless gpsdclients=no.
3. The configure will fail if pps is not available.
4. Most drivers will not be built. You must specify them when configuring.

To configure the minimal timeservice build:

```
scons timeservice=yes nmea0183=yes fixed_port_speed=9600 fixed_stop_bits=1
```

You may substitute a different GPS (e.g. "ublox" or "sirf"), You can also fix the serial parameters to avoid autobauding lag; the code assumes 8 bit bytes, so the above locks the daemon to 9600 8N1. Besides the daemon, this also builds gpsmon and ntpshmmon.

if you do not use timeservice=yes, then make sure the build is with pps=yes and ntpshm=yes (the default).

Kernel support

If you are scratch-building a Linux kernel, the configuration must include either these two lines, or the same with "y" replaced by "m" to enable the drivers as modules:

```
CONFIG_PPS=y
CONFIG_PPS_CLIENT_LDISC=y
```

Some embedded systems, like the Raspberry Pi, detect PPS on a GPIO line instead of on a serial port line. For those systems you will also need these two lines:

```
CONFIG_PPS_CLIENT_GPIO=y
CONFIG_GPIO_SYSFS=y
```

Your Linux distribution may ship a file /boot/config-XXX (where XXX is the name of a kernel) or one called /proc/config.gz (for the running kernel). This will have a list of the configuration options that were used to build the kernel. You can check if the above options are set. Usually they will be set to "m", which is sufficient.

NetBSD has included the RFC2783 Pulse Per Second API for real serial ports by default since 1998, and it works with ntpd. NetBSD 7 (forthcoming) includes RFC2783 support for USB-serial devices, and this works (with ntpd) with the GR-601W/GR-701W/GR-801W. However, gpsd's code interacts badly with the NetBSD implementation, and gpsd's support for RFC2783 PPS does not yet work on NetBSD (for serial or USB).

Other OSes have different ways to enable KPPS in their kernels. When we learn what those are, we'll document them or point at references.

Time service daemon

You will need to have either ntpd or chrony installed. If you are running a Unix variant with a package system, the packages will probably be named *ntp* (or *ntpsec*) and either *chrony* or *chronyd*.

Between ntpd and chrony, ntpd is the older and more popular choice - thus, the one with the best-established peer community if you need help in unusual situations. On the other hand, chrony has a reputation for being easier to set up and configure, and is better in situations where your machine has to be disconnected from the Internet for long enough periods of time for the clock to drift significantly.

ntpd and chrony have differing philosophies, with ntpd more interested in deriving consensus time from multiple sources while chrony tries to identify a single best source and track it closely.

A feature comparison, part of the chrony documentation, is at [CHRONY-COMPARE]. An informative email thread about the differences is [CHRONYDEFAULT]. If you don't already know enough about time service to have a preference, the functional differences between them are unlikely to be significant to you; flip a coin.

NTPSec

If you choose the ntpd option, it's best to go with the NTPsec version rather than legacy ntpd. NTPsec shares some maintainers with GPSD, and has some significant improvements in security and performance.

As of June 2019, NTPsec is available as a package in:

- OpenSUSE
- Debian (and variants like Ubuntu and Raspbian)
- Alpine
- archlinux

If it is not available as a package, you can build it from source, [GITLAB-SOURCE], it is not especially difficult.

Choice of Hardware

To get 1PPS to your NTP daemon, you first need to get it from a PPS-capable GPS receiver. As of early 2015 this means either the previously mentioned GR devices or a serial GPS receiver with 1PPS.

You can find 1PPS-capable devices supported by GPSD at [HARDWARE]. Note that the most popular consumer-grade GPS receivers do not usually deliver 1PPS through USB or even RS232. The usual run of cheap GPS mice won't do. In general, you can't use a USB device for time service unless you know it has the Macx-1 mod.

In the past, the RS232 variant of the Garmin GPS-18 has been very commonly used for time service (see [LVC] for a typical setup very well described). While it is still a respectable choice, newer devices have better sensitivity and signal discrimination. This makes them superior for indoor use as time sources.

In general, use a GPS receiver with an RS232 interface for time service if you can. The GR-601W was designed (by one of the authors, as it happens) for deployment with commodity TCP/IP routers that only have USB ports. RS232 is more fiddly to set up (with older devices like the GPS-18 you may even have to make your own cables) but it can deliver three orders of magnitude better accuracy and repeatability - enough to meet prevailing standards for a public Stratum 1 server.

Among newer receiver designs the authors found the u-blox line of receivers used in the GR-[678]01W to be particularly good. Very detailed information on its timing performance can be found at [UBLOX-TIMING]. One of us (Raymond) has recent experience with an eval kit, the EVK 6H-0-001, that would make an excellent Stratum 0 device.

Both the EVK 6H and GR-601W are built around the LEA-6H module, which is a relatively inexpensive but high-quality navigation GPS receiver. We make a note of this because u-blox also has a specialized timing variant, the LEA 6T, which would probably be overkill for an NTP server. (The 6T does have the virtue that you could probably get a good fix from one satellite in view once it knows its location, but the part is expensive and difficult to find.)

Unfortunately as of early 2015 the LEA-6H is still hard to find in a packaged RS232 version, as opposed to a bare OEM module exporting TTL levels or an eval kit like the EVK 6H-0-001 costing upwards of US$300. Search the web; you may find a here-today-gone-tomorrow offer on alibaba.com or somewhere similar.

The LEA-6T, and some other higher-end GPS receivers (but not the LEA-6H) have a stationary mode which, after you initialize it with the device's location, can deliver time service with only one good satellite lock (as opposed to the three required for a fix in its normal mode). For most reliable service we recommend using stationary mode if your device has it. GPSD tools don't yet directly support this, but that capability may be added in a future release.

The design of your host system can also affect time quality. The ±5uSec error bound quoted above is for a dual-core or better system with clock in the 2GHz range on which the OS can schedule the long-running PPS thread in GPSD on an otherwise mostly unused processor (the Linux scheduler, in particular, will do this). On a single-core system, contention with other processes can pile on several additional microseconds of error.

If you are super-serious about your time-nuttery, you may want to look into the newest generation of dedicated Stratum 1 microservers being built out of open-source SBCs like the Raspberry Pi and Beaglebone, or sometimes with fully custom designs. A representative build is well described at [RPI].

These microserver designs avoid load-induced jitter by being fully dedicated to NTP service. They are small, low-powered devices and often surprisingly inexpensive, as in costing less than US$100. They tend to favor the LEA-6H, and many of them use preinstalled GPSD on board.

Enabling PPS

You can determine whether your GPS receiver emits 1PPS, and gpsd is detecting it, by running the gpsmon utility (giving it the GPS receiver's serial-device path as argument). Watch for lines of dashes marked *PPS* in the packet-logging window; for most GPS receiver types there will also be a "PPS offset:" field in the data panels above showing the delta between PPS and your local clock.

If you don't have gpsmon available, or you don't see PPS lines in it, you can run ppscheck. As a last resort you can gpsd at -D 5 and watch for PPS state change messages in the logfile.

If you don't see evidence of incoming PPS, here are some trouble sources to check:

1. The skyview of your GPS receiver may be poor. Suspect this if, when you watch it with gpsmon, it wanders in and out of having a good 3D fix. Unfortunately, the only fix for this is to re-site your GPS where it can see more sky; fortunately, this is not as common a problem as it used to be, because modern receivers are often capable of getting a solid fix indoors.
2. If you are using an RS232 cable, examine it suspiciously, ideally with an RS232 breakout box. Cheap DB9 to DB9 cables such as those issued with UPSes often carry TXD/RXD/SG only, omitting handshake lines such as DCD, RI, and DSR that are used to carry 1PPS. Suspect this especially if the cable jacket looks too skinny to hold more than three leads!
3. Verify that your gpsd and kernel were both built with PPS support, as previously described in the section on software prerequisites.
4. Verify that the USB or RS232 device driver is accepting the ioctl that tells it to wait on a PPS state change from the device. The messages you hope **not** to see look like "KPPS cannot set PPS line discipline" and "PPS ioctl(TIOCMIWAIT) failed". The former can probably be corrected by running as root; the latter (which should never happen with an RS232 device) probably means your USB device driver lacks this wait capability entirely and cannot be used for time service.
5. If you have a solid 3D fix, a known-good cable, your software is properly configured, the wait ioctl succeeded, but you still get no PPS, then you might have a GPS receiver that fails to deliver PPS off the chip to the RS232 or USB interface. You get to become intimate with datasheets and pinouts, and might need to acquire a different GPS receiver.

Running GPSD

If you're going to use gpsd for time service, you must run in -n mode so the clock will be updated even when no clients are active. This option is forced if you built GPSD with timeservice=yes as an option.

Note that gpsd assumes that after each fix the GPS receiver will assert 1PPS first and ship sentences reporting time of fix second (and the sentence burst will end before the next 1PPS). Every GPS we know of does things in this order. (However, on some very old GPSes that defaulted to 4800 baud, long sentence bursts - notably those containing a skyview - could slop over into the next second.)

If you ever encounter an exception, it should manifest as reported times that look like they're from the future and require a negative fudge. If this ever happens, please report the device make and model to the GPSD maintainers so we can flag it in our GPS hardware database.

There is another possible cause of small negative offsets which shows up on the GR-601W: implementation bugs in your USB driver, combining with quantization by the USB poll interval. This doesn't mean the u-blox 6 inside it is actually emitting PPS after the GPS timestamp is shipped.

In order to present the smallest possible attack surface to privilege-escalation attempts, gpsd, if run as root, drops its root privileges very soon after startup - just after it has opened any serial device paths passed on the command line.

Thus, KPPS can only be used with devices passed that way, not with GPSes that are later presented to gpsd by the hotplug system. Those hotplug devices may, however, be able to use plain, non-kernel PPS. gpsd tries to automatically fall back to this when absence of root permissions makes KPPS unavailable.

In general, if you start gpsd as other than root, the following things will happen that degrade the accuracy of reported time:

1. Devices passed on the command line will be unable to use KPPS and will fall back to the same plain PPS that all hotplug devices must use, increasing the associated error from $_{1\ \text{uSec to about}}$ 5 uSec.
2. gpsd will be unable to renice itself to a higher priority. This action helps protect it against jitter induced by variable system load. It's particularly important if your NTP server is a general-use computer that's also handling mail or web service or development.
3. The way you have to configure ntpd and chrony will change away from what we show you here; ntpd will need to be told different shared-memory segment numbers, and chronyd will need a different socket location.

You may also find gpsd can't open serial devices at all if your OS distribution has done "secure" things with the permissions.

Feeding NTPD from GPSD

Most Unix systems get their time service through ntpd, a very old and stable open-source software suite which is the reference implementation of NTP. The project home page is [NTP.ORG]. We recommend using NTPsec, a recent fork that is improved and security-hardened [NTPSEC.ORG].

When gpsd receives a sentence with a timestamp, it packages the received timestamp with current local time and sends it to a shared-memory segment with an ID known to ntpd, the network time synchronization daemon. If ntpd has been properly configured to receive this message, it will be used to correct the system clock.

When in doubt, the preferred method to start your timekeeping is:

```
$ su - (or sudo -s )
# killall -9 gpsd ntpd
# gpsd -n /dev/ttyXX
# sleep 2
# ntpd -gN
# sleep 2
# cgps
```

where /dev/ttyXX is whatever 1PPS-capable device you have. In a binary-package-based Linux distribution it is probable that ntpd will already have been launched at boot time.

It's best to have gpsd start first. That way when ntpd restarts it has a good local time handy. If ntpd starts first, it will set the local clock using a remote, probably pool, server. Then ntpd has to spend a whole day slowly resynching the clock.

If you're using dhcp3-client to configure your system, make sure you disable /etc/dhcp3/dhclient-exit-hooks.d/ntp, as dhclient would restart ntpd with an automatically created ntp.conf otherwise - and gpsd would not be able to talk with ntpd any more.

While gpsd may be runnable as non-root, you will get significantly better accuracy of time reporting in root mode; the difference, while almost certainly insignificant for feeding Stratum 1 time to clients over the Internet, may matter for PTP service over a LAN. Typically only root can access kernel PPS, whereas in non-root mode you're limited to plain PPS (if that feature is available). As noted in the previous section on 1PPS quality issues, this difference has performance implications.

The rest of these setup instructions will assume that you are starting gpsd as root, with occasional glances at the non-root case.

Now check to see if gpsd has correctly attached the shared-memory segments in needs to communicate with ntpd. ntpd's rules for the creation of these segments are:

Segments 0 and 1
: Permissions are 0600 - other programs can only read and write this segment when running as root.

Segments 2, 3 and above
: Permissions are 0666 - other programs can read and write as any user. If ntpd has been configured to use these segments, any unprivileged user is allowed to provide data for synchronization.

Because gpsd can be started either as root or non-root, it checks and attaches the more privileged segment pair it can - either 0 and 1 or 2 and 3.

For each GPS receiver that gpsd controls, it will use the attached ntpshm segments in pairs (for coarse clock and pps source, respectively) starting from the first found segments.

To debug, try looking at the live segments this way

```
# ipcs -m
```

If gpsd was started as root, the results should look like this:

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x4e545030 0          root       700        96         2
0x4e545031 32769      root       700        96         2
0x4e545032 163842     root       666        96         1
0x4e545033 196611     root       666        96         1
```

For a bit more data try this:

```
cat /proc/sysvipc/shm
```

If gpsd cannot open the segments, check that you are not running SELinux or apparmor. Either may require you to configure a security exception.

If you see the shared segments (keys 1314148400 — 1314148403), and no gpsd or ntpd is running then try removing them like this:

```
# ipcrm -M 0x4e545030
# ipcrm -M 0x4e545031
# ipcrm -M 0x4e545032
# ipcrm -M 0x4e545033
```

Here is a minimal sample ntp.conf configuration to work with GPSD run as root, telling ntpd how to read the GPS notifications

```
pool us.pool.ntp.org iburst

driftfile /var/lib/ntp/ntp.drift
logfile /var/log/ntp.log

restrict default kod nomodify notrap nopeer noquery
restrict -6 default kod nomodify notrap nopeer noquery
restrict 127.0.0.1 mask 255.255.255.0
restrict -6 ::1

# GPS Serial data reference (NTP0)
server 127.127.28.0
fudge 127.127.28.0 time1 0.9999 refid GPS
```

```
# GPS PPS reference (NTP1)
server 127.127.28.1 prefer
fudge 127.127.28.1 refid PPS
```

The number "0.9999" is a placeholder, to be explained shortly. It is **not a number to be used in production** - it's too large. If you can't replace it with a real value, it would be best to leave out the clause entirely so the entry looks like

```
fudge 127.127.28.0 refid GPS
```

This is equivalent to declaring a time1 of 0.

The pool statement adds a variable number of servers (often 10) as additional time references needed by ntpd for redundancy and to give you a reference to see how well your local GPS receiver is performing. If you are outside of the USA replace the pool servers with one in your local area. See [USE-POOL] for further information.

The pool statement, and the driftfile and logfile declarations after it, will not be strictly necessary if the default ntp.conf that your distribution supplies gives you a working setup. The two pairs of server and fudge declarations are the key.

ntpd can be used in Denial of Service (DoS) attacks. To prevent that, but still allow clients to request the local time, be sure the restrict statements are in your ntpd config file. For more information see [CVE-2009-3563].

Users of ntpd versions older than revision ntp-4.2.5p138 should instead use this ntp.conf, when gpsd is started as root:

```
pool us.pool.ntp.org iburst

driftfile /var/lib/ntp/ntp.drift
logfile /var/log/ntp.log

restrict default kod nomodify notrap nopeer noquery
restrict -6 default kod nomodify notrap nopeer noquery
restrict 127.0.0.1 mask 255.255.255.0
restrict -6 ::1

# GPS Serial data reference (NTP0)
server 127.127.28.0 minpoll 4 maxpoll 4
fudge 127.127.28.0 time1 0.9999 refid GPS

# GPS PPS reference (NTP1)
server 127.127.28.1 minpoll 4 maxpoll 4 prefer
fudge 127.127.28.1 refid PPS
```

Users of ntpd versions prior to ntp-4.2.5 do not have the "pool" option. Alternative configurations exist, but it is recommended that you upgrade ntpd, if feasible.

The magic pseudo-IP address 127.127.28.0 identifies unit 0 of the ntpd shared-memory driver (NTP0); 127.127.28.1 identifies unit 1 (NTP1). Unit 0 is used for in-band message timestamps and unit 1 for the (more accurate, when available) time derived from combining in-band message timestamps with the out-of-band PPS synchronization pulse. Splitting these notifications allows ntpd to use its normal heuristics to weight them.

Different units - 2 (NTP2) and 3 (NTP3), respectively - must be used when gpsd is not started as root. Some GPS HATs put PPS time on a GPIO pin and will also use unit 2 (NTP2) for the PPS time correction.

With this configuration, ntpd will read the timestamp posted by gpsd every 64 seconds (16 if non-root) and send it to unit 0.

The number after the parameter time1 (0.9999 in the example above) is a "fudge", offset in seconds. It's an estimate of the latency between the time source and the *real* time. You can use it to compensate out some of the fixed delays in the system. An 0.9999 fudge would be ridiculously large.

You may be able to find a value for the fudge by looking at the entry for your GPS receiver type on [HARDWARE]. Later in this document we'll explain methods for estimating a fudge factor on unknown hardware.

There is nothing magic about the refid fields; they are just labels used for generating reports. You can name them anything you like.

When you start gpsd, it will wait for a few good fixes before attempting to process PPS. You should run gpsmon or cgps to verify your GPS receiver has a 3D lock before worrying about timekeeping.

After starting (as root) ntpd, then gpsd, a listing similar to the one below should appear as the output of the command "ntpq -p" (after allowing the GPS receiver to acquire a 3D fix). This may take up to 30 minutes if your GPS receiver has to cold-start or has a poor skyview.

```
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
xtime-a.timefreq .ACTS.           1 u   40   64  377   59.228   -8.503   0.516
-bonehed.lcs.mit 18.26.4.106      2 u   44   64  377   84.259    4.194   0.503
+clock.sjc.he.ne .CDMA.           1 u   41   64  377   23.634   -0.518   0.465
+SHM(0)          .GPS.            0 l   50   64  377    0.000    6.631   5.331
```

The line with refid ".GPS." represents the in-band time reports from your GPS receiver. When you are getting PPS then it may look like this:

```
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
xtime-a.timefreq .ACTS.           1 u   40   64  377   59.228   -8.503   0.516
-bonehed.lcs.mit 18.26.4.106      2 u   44   64  377   84.259    4.194   0.503
+clock.sjc.he.ne .CDMA.           1 u   41   64  377   23.634   -0.518   0.465
+SHM(0)          .GPS.            0 l   50   64  377    0.000    6.631   5.331
*SHM(1)          .PPS.            0 l   49   64  377    0.000    0.222   0.310
```

Note the additional ".PPS." line.

If the value under "reach" for the SHM lines remains zero, check that gpsd is running; cgps reports a 3D fix; and the *-n* option was used. Some GPS recievers specialized for time service can report time with signal lock on only one satellite, but with most devices a 3D fix is required.

When the SHM(0) line does not appear at all, check your ntp.conf and the system logs for error messages from ntpd.

Notice the 1st and 3rd servers, stratum 1 servers, disagree by more than 8 mSec. The 1st and 2nd servers disagree by over 12 mSec. Our local PPS reference agrees to the clock.sjc.he.net server within the expected jitter of the GR-601W in use.

When no other servers or local reference clocks appear in the NTP configuration, the system clock will lock onto the GPS clock, but this is a fragile setup - you can lose your only time reference if the GPS receiver is temporarily unable to get satellite lock.

You should always have at least two (preferably four) fallback servers in your ntpd.conf for proper ntpd operation, in case your GPS receiver fails to report time. The *pool* command makes this happen. And you'll need to adjust the offsets (fudges) in your ntp.conf so the SHM(0) time is consistent with your other servers (and other local reference clocks, if you have any). We'll describe how to diagnose and tune your server configuration in a later section.

Also note that after cold-starting ntpd it will calibrate for up to 15 minutes before it starts adjusting the clock. Because the frequency error estimate ("drift") that NTP uses isn't right when you first start NTP, there will be a phase error that persists while the frequency is estimated. So if your clock is a little slow, then it will keep getting behind, and the positive offset will cause NTP to adjust the phase forward and also increase the frequency offset error. After a day or so or maybe less the frequency estimate will be very close and there won't be a persistent offset.

The GPSD developers would like to receive information about the offsets (fudges) observed by users for each type of receiver. If your GPS receiver is not present in [HARDWARE], or doesn't have a recommended fudge, or you see a fudge value very different from what's there, send us the output of the "ntpq -p" command and the make and type of receiver.

Feeding chrony from GPSD

chrony is an alternative open-source implementation of NTP service, originally designed for systems with low-bandwidth or intermittent TCP/IP service. It interoperates with ntpd using the same NTP protocols. Unlike ntpd which is designed to always be connected to multiple internet time sources, chrony is designed for long periods of offline use. Like ntpd, it can either operate purely as a client or provide time service. The chrony project has a home page at [CHRONY]. Its documentation includes an instructive feature comparison with ntpd at [CHRONY-COMPARE].

gpsd, when run as root, feeds reference clock information to chronyd using a socket named /var/run/chrony.ttyXX.sock (where ttyXX is replaced by the GPS receiver's device name. This allows multiple GPS receivers to feed one chronyd.

No gpsd configuration is required to talk to chronyd. chronyd is configured using the file /etc/chrony.conf or /etc/chrony/chrony.conf. Check your distributions documentation for the correct location. To get chronyd to connect to gpsd using the basic ntpd compatible SHM method add this to use this basic chrony.conf file:

```
server 0.us.pool.ntp.org
server 1.us.pool.ntp.org
server 2.us.pool.ntp.org
server 3.us.pool.ntp.org

driftfile /var/lib/chrony/drift

allow

# set larger delay to allow the NMEA source to overlap with
# the other sources and avoid the falseticker status
refclock SHM 0 refid GPS precision 1e-1 offset 0.9999 delay 0.2
refclock SHM 1 refid PPS precision 1e-9
```

You need to add the "precision 1e-9" on the SHM 1 line as chronyd fails to read the precision from the SHM structure. Without knowing the high precision of the PPS on SHM 1 it may not place enough importance on its data.

If you are outside of the USA replace the pool servers with one in your local area. See [USE-POOL] for further information.

The offset option is functionally like ntpd's time1 option, used to correct known and constant latency.

The allow option allows anyone on the internet to query your server's time.

To get chronyd to connect to gpsd using the more precise socket method add this to your chrony.conf file (replacing ttyXX with your device name).

If running as root:

```
server 0.us.pool.ntp.org
server 1.us.pool.ntp.org
server 2.us.pool.ntp.org
server 3.us.pool.ntp.org

driftfile /var/lib/chrony/drift

allow

# set larger delay to allow the NMEA source to overlap with
# the other sources and avoid the falseticker status
refclock SHM 0 refid GPS precision 1e-1 offset 0.9999 delay 0.2
refclock SOCK /var/run/chrony.ttyXX.sock refid PPS
```

If not running as root change the "refclock SOCK" line to:

```
refclock SOCK /tmp/chrony.ttyXX.sock refid PPS
```

See the chrony man page for more detail on the configuration options [CHRONY-MAN].

Finally note that chronyd needs to be started before gpsd so the socket is ready when gpsd starts up.

If running as root, the preferred starting procedure is:

```
$ su - (or sudo -s )
# killall -9 gpsd chronyd
# chronyd -f /etc/chrony/chrony.conf
# sleep 2
# gpsd -n /dev/ttyXX
# sleep 2
# cgps
```

After you have verified with cgps that your GPS receiver has a good 3D lock you can check that gpsd is outputting good time by running ntpshmmon.

```
# ntpshmmon
ntpshmmon version 1
#      Name   Seen@               Clock                Real               L Prec
sample NTP0 1461537438.593729271 1461537438.593633306 1461537438.703999996 0 -1
sample NTP1 1461537439.000421494 1461537439.000007374 1461537439.000000000 0 -20
sample NTP0 1461537439.093844900 1461537438.593633306 1461537438.703999996 0 -1
sample NTP0 1461537439.621309382 1461537439.620958240 1461537439.703999996 0 -1
sample NTP1 1461537440.000615395 1461537439.999994105 1461537440.000000000 0 -20
sample NTP0 1461537440.122079148 1461537439.620958240 1461537439.703999996 0 -1
^C
```

If you see only "NTP2", instead, you forgot to go root before starting gpsd.

Once ntpshmmon shows good time data you can see how chrony is doing by running *chronyc sources*. Your output will look like this:

```
# chronyc sources

210 Number of sources = 7
MS Name/IP address         Stratum Poll Reach LastRx Last sample
===============================================================================
#- GPS                           0   4   377    12  +3580us[+3580us] +/- 101ms
#* PPS                           0   4   377    10    -86ns[ -157ns] +/- 181ns
^? vimo.dorui.net                3   6   377    23   -123ms[ -125ms] +/- 71ms
^? time.gac.edu                  2   6   377    24   -127ms[ -128ms] +/- 55ms
^? 2001:470:1:24f::2:3           2   6   377    24   -122ms[ -124ms] +/- 44ms
^? 142.54.181.202                2   6   377    22   -126ms[ -128ms] +/- 73ms
```

The stratum is as in ntpq. The Poll is how many seconds elapse between samples. The Reach is as in ntpq. LastRx is the time since the last successful sample. Last sample is the offset and jitter of the source.

To keep chronyd from preferring NMEA time over PPS time, you can add an overlarge fudge to the NMEA time. Or add the suffix *noselect* so it is never used, just monitored.

Performance Tuning

This section is general and can be used with either ntpd or chronyd. We'll have more to say about tuning techniques for the specific implementations in later sections.

The clock crystals used in consumer electronics have two properties we are interested in: accuracy and stability. **Accuracy** is how well the measured frequency matches the number printed on the can. **Stability** is how well the frequency stays the same even if it isn't accurate. (Long term aging is a third property that is interesting, but ntpd and chrony both a use a drift history that is relatively short; thus, this is not a significant cause of error.)

Typical specs for oscillator packages are 20, 50, 100 ppm. That includes everything; initial accuracy, temperature, supply voltage, aging, etc.

With a bit of software, you can correct for the inaccuracy. ntpd and chrony both call it **drift**. It just takes some extra low order bits on the arithmetic doing the time calculations. In the simplest case, if you thought you had a 100 MHz crystal, you need to change that to something like 100.000324. The use of a PPS signal from gpsd contributes directly to this measurement.

Note that a low drift contributes to stability, not necessarily accuracy.

The major source of instability is temperature. Ballpark is the drift changes by 1 PPM per degree C. This means that the drift does not stay constant, it may vary with a daily and yearly pattern. This is why the value of drift the ntpd uses is calculated over a (relatively) short time.

So how do we calculate the drift? The general idea is simple. Measure the time offset every N seconds over a longer window of time T, plot the graph, and fit a straight line. The slope of that line is the drift. The units cancel out. Parts-per-million is a handy scale.

How do you turn that handwaving description into code? One easy way is to set N=2 and pick the right T, then run the answer through a low pass filter. In that context, there are two competing sources of error. If T is too small, the errors on each individual measurement of the offset time will dominate. If T is too big, the actual drift will change while you are measuring it. In the middle is a sweet spot. (For an example, see [ADEV-PLOT].)

Both ntpd and chrony use this technique; ntpd also uses a more esoteric form of estimation called a "PLL/FLL hybrid loop". How T and N are chosen is beyond the scope of this HOWTO and varies by implementation and tuning parameters.

If you turn on the right logging level ("statistics loopstats peerstats" for ntpd, "log measurements tracking" for chronyd) , that will record both offset, drift, and the polling interval. The ntpd stats are easy to feed to gnuplot, see the example script in the GPSD contrib directory. The most important value is the offset reported in the 3rd field in loopstats and the last field in tracking.log. With gnuplot you can compare them (after concatenating the rotated logs):

```
plot "tracking.log" using 7 with lines, "loopstats" using 3 with lines
```

While your NTP daemon (ntpd or chrony) is adjusting the polling interval, it is assuming that the drift is not changing. It gets confused if your drift changes abruptly, say because you started some big chunk of work on a machine that's usually idle and that raises the temperature.

Your NTP daemon writes out the drift every hour or so. (Less often if it hasn't changed much to reduce the workload on flash file systems.) On startup, it reloads the old value.

If you restart the daemon, it should start with a close old drift value and quickly converge to the newer slightly different value. If you reboot, expect it to converge to a new/different drift value and that may take a while depending on how different the basic calibration factors are.

ARP is the sound of your server choking

By default ntpd and chronyd poll remote servers every 64 seconds. This is an unfortunate choice. Linux by default only keeps an ARP table entry for 60 seconds, anytime thereafter it may be flushed.

If the ARP table has flushed the entry for a remote peer or server then when the NTP server sends a request to the remote server an entire ARP cycle will be added to the NTP packet round trip time (RTT). This will throw off the time measurements to servers on the local lan.

On a RaspberryPi ARP has been shown to impact the remote offset by up to 600 uSec in some rare cases.

The solution is the same for both ntpd and chronyd, add the "maxpoll 5" command to any 'server" or "peer directive. This will cause the maximum polling period to be 32 seconds, well under the 60 second ARP timeout.

Watch your temperatures

The stability of the system clock is very temperature dependent. A one degree change in room temperature can create 0.1 ppm of clock frequency change. Once simple way to see the effect is to place your running NTP server inside bubble wrap. The time will take a quick and noticeable jump.

If you leave your NTP server in the bubble wrap you will notice some improved local and remote offsets.

Powersaving is not your friend

Normally enabling power saving features is a good thing: it saves you power. But when your CPU changes power saving modes (cstates for Intel CPUs) the impact on PPS timing is noticeable. For some reason the NO_HZ kernel mode has a similar bad effect on

timekeeping.

To improve your timekeeping, turn off both features on Intel CPUs by adding this to your boot command line:

```
nohz=off intel_idle.max_cstate=0
```

For ARM, be sure NO_HZ is off:

```
nohz=off
```

You will also need to select the *performance* CPU governor to keep your CPU set to the maximum speed for continuous usage. How you see and set your governor will be distribution specific. The easiest way it to recompile your kernel to only provide the performance governor.

NTP tuning and performance details

This section deals specifically with ntpd. It discusses algorithms used by the ntpd suite to measure and correct the system time. It is not directly applicable to chronyd, although some design considerations may be similar.

It is hard to optimize what you can't visualize. The easiest way to visualize ntpd performance is with ntpviz from [NTPSEC.ORG]. Once you are regularly graphing your server performance it is much easier to see the results of changes.

NTP performance tuning

For good time stability, you should always have at least four other servers in your ntpd or chrony configuration besides your GPS receiver - in case, for example, your GPS receiver is temporarily unable to achieve satellite lock, or has an attack of temporary insanity. You can find public NTP servers to add to your configuration at [USE-POOL].

To minimize latency variations, use the national and regional pool domains for your country and/or nearby ones. Your ntp.conf configuration line should probably look like this

```
pool us.pool.ntp.org iburst
```

where "us" may be replaced by one of the zone/country codes the Pool project supports (list behind the "Global" link at [ZONES]). The "pool" tag expands to four randomly chosen servers by default. "iburst" implements a fast start algorithm that also weeds out bad servers.

Note that a server can be a poor performer (what the NTP documentation colorfully calls a "falseticker") for any of three reasons. It may be shipping bad time, or the best routes between you and it have large latency variations (jitter), or it may have a time-asymmetric route, to you (that is, B-to-A time is on average very different from A-to-B time). Asymmetric routing is the most common cause of serious problems.

The standard tool for tuning ntpd is "ntpq" ("NTP query program"). To show a list of all servers declared in ntp.conf and their statistics, invoke it with the "-p" option. On a sample system configured with 7 servers from the NTP pool project and one PPS GPS receiver attached via RS232, this is the output:

```
$ ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
-arthur.testserv 162.23.41.56    2 u   62   64  377    5.835   -1.129   8.921
-ntppublic.uzh.c 130.60.159.7    3 u   62   64  377    6.121   -4.102   6.336
-smtp.irtech.ch  162.23.41.56    2 u   35   64  377   15.521   -1.677   8.678
+time2.ethz.ch   .PPS.           1 u   27   64  377    5.938   -1.713  16.404
-callisto.mysnip 192.53.103.104 2 u   53   64  377   49.357   -0.274   5.125
-shore.naturalne 122.135.113.81 3 u   22   64  377   14.676   -0.528   2.601
-ntp.univ-angers 195.220.94.163 2 u   41   64  377   40.678   -1.847  13.391
+SHM(0)          .GPS.          0 l    4   64  377    0.000   34.682   7.952
*SHM(1)          .PPS.          0 l    3   64  377    0.000   -2.664   0.457
```

The interesting columns are "remote", "st", "reach" and "offset".

"remote" is the name of the remote NTP server. The character in its first column shows its current state: "-" or "x" for out-of-tolerance servers, "+" for good servers ("truechimers"), and "*" for the one good server currently used as the primary reference. The calculations used to determine a server's state are outside the scope of this document; details are available in NTPv4 RFC 5905.

"st" shows the remote server's stratum.

"reach" is the octal representation of the remote server's reachability. A bit is set if the corresponding poll of the server was successful, i.e. the server returned a reply. New poll results are shifted in from the least significant bit; results older than 8 polls are discarded. In the absence of network problems, this should show "377".

"offset" shows the mean offset in the times reported between this local host and the remote server in milliseconds. This is the value that can be fudged with the "time1" parameter of the GPS server line in ntp.conf. If the offset is positive, reduce the time1 value and vice versa.

The asterisk in this example indicates that ntpd has correctly preferred .PPS. over .GPS., as it should. If for some reason it locks on to GPS time as a preferred source, you can add an overlarge fudge to the NMEA time to discourage it. Or add the suffix *noselect* so GPS time is never used, just monitored.

A more detailed description of the output is available at [NTPQ-OUTPUT].

In order to determine the correct GPS offset, do one of the following:

Peerstats-based procedure

1. Add these lines to ntp.conf:

```
statsdir /var/log/ntpstats/
statistics peerstats
filegen peerstats file peerstats type day enable
```

This enables logging of the peer server statistics.

1. Make sure the directory exists properly. For ntpd as root do:

```
   # mkdir -p /var/log/ntpstats
   # chown ntp:ntp /var/log/ntpstats
```

1. Start ntpd and let it run for at least four hours. Periodically check progress with "ntpq -p" and wait until change has settled out.
2. Calculate the average GPS offset using this script (a copy is included as contrib/ntpoffset in the GPSD distribution):

```
awk '
     /127\.127\.28\.0/ { sum += $5 * 1000; cnt++; }
     END { print sum / cnt; }
' </var/log/ntpstats/peerstats
```

This prints the average offset.

1. Adjust the "time1" value for unit 0 of your ntp.conf (the non-PPS channel) by subtracting the average offset from step 4.
2. Restart ntpd.

Loopstats-based procedure

Recall that magic pseudo-IP address 127.127.28.0 identifies unit 0 of the ntpd shared-memory driver (NTP0); 127.127.28.1 identifies unit 1 (NTP1). Unit 0 is used for in-band message timestamps (IMT) and unit 1 for the (more accurate, when available) time derived from combining IMT with the out-of-band PPS synchronization pulse. Splitting these notifications allows ntpd to use its normal heuristics to weight them.

We assume that the 1PPS signal, being just one bit long, and directly triggering an interrupt, is always on time (sic). Correcting for latency in the 1PPS signal is beyond the scope of this document. The IMT, however, may be delayed, due to it being emitted, copied to shared memory, etc.

Based on advice and script fragments on the GPSD list, the following may help to calculate the *time1* factor. You may need to modify these scripts for your particular setup.

These scripts are for the combination of GPSD and ntpd. If you use chronyd, you **will** need to modify these, at the least.

ntpviz procedure

If all this calculating and graphing looks painful, then grab a copy of ntpviz from [NTPSEC.ORG]. ntpviz generates lots of pretty graphs and html pages. It even calculates the correct IMT offset, and other performance metrics for you.

Format of the loopstats and peerstats files

The following is incorporated from the ntpd website, see [NTP-MONOPT]

loopstats

Record clock discipline loop statistics. Each system clock update appends one line to the loopstats file set:

Example: 50935 75440.031 0.000006019 13.778 0.000351733 0.013380 6

Item	Units	Description
50935	MJD	date
75440.031	s	time past midnight (UTC)

0.000006019	s	clock offset
13.778	PPM	frequency offset
0.000351733	s	RMS jitter
0.013380	PPM	RMS frequency jitter (aka wander)
6	log2 s	clock discipline loop time constant

peerstats

Record peer statistics. Each NTP packet or reference clock update received appends one line to the peerstats file set:

Example: 48773 10847.650 127.127.4.1 9714 -0.001605376 0.000000000 0.001424877 0.000958674

Item	Units	Description
48773	MJD	date
10847.650	s	time past midnight (UTC)
127.127.4.1	IP	source address
9714	hex	status word
-0.001605376	s	clock offset
0.000000000	s	roundtrip delay
0.001424877	s	dispersion
0.000958674	s	RMS jitter

Measurement of delay

There are three parts to measuring and correcting for the delay in processing the 1PPS signal.

1. Running ntpd without using the IMT (but using the 1PPS time)
2. Measuring the delay between the two messages
3. Applying the correction factor

We assume that you have successfully integrated GPSD with ntpd already. You should also have a decent set of NTP servers you are syncing to.

1. Running ntpd without IMT

Locate the line in your ntp.conf that refers to the SHM0 segment and append *noselect* to it. As an example, the first two lines in the sample above will become:

```
server 127.127.28.0 minpoll 4 maxpoll 4 noselect
fudge 127.127.28.0 time1 0.420 refid GPS
```

ntpd will now continue to monitor the IMT from GPSD, but not use it for its clock selection algorithm. It will still write out statistics to the peerstats file. Once ntpd is stable (a few hours or so), we can process the peerstats file.

1. Measuring the delay between the two messages

From the *peerstats* file, extract the lines corresponding to 127.127.28.0

```
grep 127.127.28.0 peerstats > peerstats.shm
```

You can now examine the offset and jitter of the IMT. [ANDY-POST] suggests the following gnuplot fragment (you will need to set output options before plotting).

```
        set term gif
        set output "fudge.gif"
```

If your gnuplot has X11 support, and you do not wish to save the plot, the above may not be required. Use:

```
        set term x11
```

Now plot the GPSD shared memory clock deviations from the system clock. (You will get the GPSD shared memory clock fudge value estimate from this data when NTP has converged to your satisfaction.)

```
        gnuplot> plot "peerstats.shm" using ($2):($5):($8) with yerrorbars
        gnuplot> replot "peerstats.shm" using ($2):($5) with lines
```

1. Applying the correction factor

By examining the plot generated above, you should be able to estimate the offset between the 1PPS time and the GPS time.

If, for example, your estimate of the offset is -0.32s, your time1 fudge value will be *0.32*. Note the change of sign.

Polling Interval

ntpd seems to better use a PPS refclock when the polling interval is as small as possible. The ntpd default minpoll is 6, and can be set to as low as 4. NTPsec versions 0.9.5 and above of ntpd allow you to set minpoll and maxpoll as low as 0. Changing minpoll from 4 to 3, 2, 1 or maybe even a,0 may reduce your PPS jitter by over a factor of 4.

It change may require several hours for ntpd to converge with the new settings. Use ntpviz to find the best poll interval for your system.

The value that yields the lowest jitter may not be the one that yields the best Local Clock Frequency Offset.

```
server 127.127.28.1 minpoll 0 maxpoll 0 prefer
```

Chrony performance tuning

The easiest way to determine the offset with chronyd is probably to configure the source whose offset should be measured with the noselect option and a long poll, let chronyd run for at least 4 hours and observe the offset reported in the chronyc sourcestats output. If the offset is unstable, wait longer. For example:

SHM 0 configured as: refclock SHM 0 poll 8 filter 1000 noselect

```
# chronyc sourcestats
210 Number of sources = 6
Name/IP Address            NP  NR  Span  Frequency  Freq Skew  Offset  Std Dev
==============================================================================
SHM0                       21   9   85m      4.278      4.713   +495ms  8896us
SHM1                       20   8   307      0.000      0.002     +0ns   202ns
mort.cihar.com             21   8   72m      0.148      0.798   +668us   490us
vps2.martinpoljak.net       6   4   17m    -53.200    141.596    -24ms    15ms
ntp1.kajot.cz              25  16   91m     -0.774      1.494    -11ms  1859us
ntp1.karneval.cz           17  10   89m      0.127      0.539  -4131us   574us
```

In this case (Garmin 18x) the offset specified in the config for the SHM 0 source should be around 0.495.

Providing local NTP service using PTP

By now if you have a good serial PPS signal your local clock should have jitter on the order of 1 uSec. You do not want the hassle of placing a GPS receiver on each of your local computers. So you install chrony or ntp on your other hosts and configure them to use your NTP PPS server as their local server.

With your best setup on a lightly loaded GigE network you find that your NTP clients have jitter on the order of 150 uSec, or 150 times worse than your master. Bummer, you want to do much better, so you look to the Precision Time Protocol [PTP] for help. PTP is also known as IEEE 1588.

With PTP you can easily synchronize NTP hosts to 5 uSec with some generic NIC hardware and newer Linux kernels. Some of the Ethernet drivers have been modified to time stamp network packets when sending and receiving. This is done with the new SO_TIMESTAMPING socket option. No hardware support is required.

A more recent addition is PTP Hardware Clock (PHC) support. This requires hardware support in the NIC.

Software timestamping is more mature, available on more NICs, and almost as accurate as hardware timestamping. Try it first. This HOWTO will build on those results.

One final wrinkle before proceeding with PTP. Ethernet ports have something called [EEE] (IEEE 802.3az). Percentage wise EEE can save 50% of the Ethernet energy needs. Sadly this is 50% of an already small energy usage. Only important in large data centers. EEE can be very disruptive to timekeeping. Up to almost 1 Sec of errors in offset, wander and jitter. To see if you have EEE enabled, and then turn it off:

```
# ethtool --show-eee eth0
EEE Settings for eth0:
        EEE status: enabled - inactive
```

```
        Tx LPI: 0 (us)
        Supported EEE link modes:  100baseT/Full
                                   1000baseT/Full
        Advertised EEE link modes:  100baseT/Full
                                    1000baseT/Full
        Link partner advertised EEE link modes:  Not reported
# ethtool --set-eee eth0 eee off
# ethtool --show-eee eth0
EEE Settings for eth1:
        EEE status: disabled
        Tx LPI: disabled
        Supported EEE link modes:  100baseT/Full
                                   1000baseT/Full
        Advertised EEE link modes:  Not reported
        Link partner advertised EEE link modes:  Not reported
```

PTP with software timestamping

To start you need to verify that your running Linux kernel configuration includes these two lines, or the same with "y" replaced by "m" to enable the drivers as modules:

```
CONFIG_NETWORK_PHY_TIMESTAMPING=y
PTP_1588_CLOCK=y
```

Then you need to verify that your Ethernet driver supports PTP by running this command as root:

```
# ethtool -T eth0 | fgrep SOFTWARE
        software-transmit     (SOF_TIMESTAMPING_TX_SOFTWARE)
        software-receive      (SOF_TIMESTAMPING_RX_SOFTWARE)
        software-system-clock (SOF_TIMESTAMPING_SOFTWARE)
```

If the result includes those three lines then you have support for software PTP timestamping. We will leave hardware timestamping for later.

Next you will need the [LINUX-PTP] package, just follow the simple instructions on their web page to download, compile and install on your NTP server and its slaves. Be sure to also follow their instructions on how to configure your Linux kernel.

In this setup we will just use the ptp4l program. This program measures the delay and offset between a master and slaves and shares that information with chronyd or ntpd using an SHM. Since gpsd also uses SHM be very careful not to have the two SHM servers stepping on the same shmid.

If you are using ntpd, then add the last three lines below to your master ntp.conf file to configure the SHM.

```
# GPS Serial data reference (NTP0)
server 127.127.28.0
fudge 127.127.28.0 time1 0.9999 refid GPS

# GPS PPS reference (NTP1)
server 127.127.28.1 prefer
fudge 127.127.28.1 refid PPS

# local PTP reference (NTP2)
server 127.127.28.2
fudge 127.127.28.2 refid PTP
```

If you are using chronyd, then add the last one line below to your master chronyd.conf file to configure the SHM.

```
refclock SHM 0 refid GPS precision 1e-1 offset 0.9999 delay 0.2
refclock SHM 1 refid PPS precision 1e-9
refclock SHM 2 refid PTP precision 1e-9
```

To configure the master ptp4l, create a new file /usr/local/etc/ptp4l.conf with these contents:

```
[global]
# only syslog every 1024 seconds
summary_interval 10

# send to to chronyd/ntpd using SHM 2
clock_servo ntpshm
ntpshm_segment 2
```

```
# set our priority high since we have PPS
priority1 10
priority2 10

[eth0]
```

Now as root on the master, start the ptp4l daemon:

```
# ethtool --set-eee eth0 eee off
# ptp4l -S -f /usr/local/etc/ptp4l.conf &
```

Configuration of the master server is now complete. Now to configure the slaves. If the slaves also have PPS then configure them as masters. Otherwise you will stomp on your SHMs.

If you are using ntpd, then add the last three lines below to your master ntp.conf file to configure your one and only SHM.

```
# local PTP reference (NTP0)
server 127.127.28.0
fudge 127.127.28.0 refid PTP
```

If you are using chronyd, then add the one line below to your master chronyd.conf file to configure your one and only SHM.

```
refclock SHM 0 refid PTP precision 1e-9
```

To configure the slave ptp4l, create a new file /usr/local/etc/ptp4l.conf with these contents:

```
[global]
# only syslog every 1024 seconds
summary_interval 10

# send to to chronyd/ntpd using SHM 0
clock_servo ntpshm
ntpshm_segment 0

[eth0]
```

Now as root on the slave, as with the master, turn off EEE and start the ptp4l daemon:

```
# ethtool --set-eee eth0 eee off
# ptp4l -S -f /usr/local/etc/ptp4l.conf &
```

Configuration of the slave server is now complete. Follow the earlier procedures for checking the jitter on the SHM on the slaves. Give it a few hours to settle and your hosts will now be synced to around 5 uSec.

PTP with hardware timestamping

Some NICs requires two additional kernel options. Just in case, verify that your running Linux kernel configuration includes these lines, or the same with "y" replaced by "m" to enable the drivers as modules:

```
CONFIG_DP83640_PHY=y
CONFIG_PTP_1588_CLOCK_PCH=y
```

Then you need to verify that your Ethernet driver supports PTP by running ethtool as root and verify at least the following lines are present in the output:

```
# ethtool -T eth0
        hardware-transmit     (SOF_TIMESTAMPING_TX_HARDWARE)
        hardware-receive      (SOF_TIMESTAMPING_RX_HARDWARE)
        all                   (HWTSTAMP_FILTER_ALL)
```

Your NIC may have more features, and your driver may support them for better results.

In the software timestamping above the ptp4l program took care of all steps to determine the slave offset from the master and feeding that to a SHM for ntpd or chronyd to use.

In hardware timestamping mode ptp4l will continue to perform most of the work. An additional program, phc2sys, will take over the duties of reading the hardware timestamps from the NIC, computing the offset, and feeding that to the SHM.

phc2sys will use the SHM exactly as ptp4l did previously so no change is required to your ntpd or chronyd configuration.

To keep things simple, for now, we will not touch the already configured and working software timestamping master server. We will proceed to configure a slave.

To configure the slave ptp4l, edit your /usr/local/etc/ptp4l.conf to remove the ntpshm options:

```
[global]
# only syslog every 1024 seconds
summary_interval 10

clock_servo linreg

[eth0]
```

Now as root on the slave, as with the master, turn off EEE and start the ptp4l daemon:

```
# ethtool --set-eee eth0 eee off
# ptp4l -H -f /usr/local/etc/ptp4l.conf &
# sleep 3
# phc2sys -a -r -E ntpshm -m -M 0 &
```

Configuration of the slave server is now complete. Follow the earlier procedures for checking the jitter on the SHM on the slaves.

Sadly, theory and practice diverge here. I have never succeeded in making hardware timestamping work. I have successfully trashed my host system clock. Tread carefully. If you make progress please pass on some clue.

Providing public NTP service

[NTP-FAQ] has good advice on things to be sure you have done - and are ready to do - before becoming a public server. One detail it doesn't mention is that you'll need to un-firewall UDP port 123. The NTP protocol does not use TCP, so no need to unblock TCP port 123.

If and when you are ready to go public, see [JOIN-POOL].

Acknowledgments

Beat Bolli <bbolli@ewanet.ch> wrote much of the section on NTP performance tuning. Hal Murray <hmurray@megapathdsl.net> wrote much of the section on NTP working and performance details. Sanjeev Gupta <ghane0@gmail.com> assisted with editing. Shawn Kohlsmith <skohlsmith@gmail.com> tweaked the Bibliography. Jaap Winius <jwinius@rjsystems.nl> cleaned up some terminology.

The loopstats-based tuning procedure for ntpd was drafted by Sanjeev Gupta <ghane0@gmail.com>, based on discussions on the GPSD list [GPSD-LIST] in Oct and Nov 2013. Code examples are based on work by Andy Walls <andy@silverblocksystems.net>. A copy of the original email can be found at [ANDY-POST]. A thorough review was contributed by Jaap Winius <jwinius@rjsystems.nl>.

References

- [TIME-INTRO] Introduction to Time Service
- [WIKI-NTP] Network Time Protocol
- [NTP-FAQ] NTP FAQ
- [RFC-2783] RFC 2783
- [RFC-5905] RFC 5905
- [MACX-1] Navisys GR-601W u-blox-6 "Macx-1" USB GPS receiver
- [CHRONY-COMPARE] ntpd (comparison with chrony)
- [CHRONYDEFAULT] https://lists.fedoraproject.org/pipermail/devel/2010-May/135679.html
- [HARDWARE] Compatible Hardware
- [UBLOX-TIMING] GPS-based timing considerations with u-blox 6 receivers
- [RPI] The Raspberry Pi as a Stratum-1 NTP Server
- [NTP.ORG] Home of the Network Time Protocol project
- [NTPSEC.ORG] Wecome to NTPsec
- [USE-POOL] How do I use pool.ntp.org?
- [CVE-2009-3563] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3563

- [CHRONY] Chrony Home
- [CHRONY-MAN] https://chrony.tuxfamily.org/manual.html
- [ADEV-PLOT] Allan deviation and Averaging
- [ZONES] https://www.pool.ntp.org/zone
- [NTPQ-OUTPUT] ntpq output description
- [JOIN-POOL] How do I join pool.ntp.org?
- [ANDY-POST] Clarifications needed for the time-service HOWTO
- [NTP-MONOPT] NTP Monitoring
- [GPSD-LIST]gpsd-dev Archives
- [PTP] PTP
- [LINUX-PTP] Linux PTP
- [EEE] Energy-Efficient Ethernet
- [LVC] https://www.rjsystems.nl/en/2100-ntpd-garmin-gps-18-lvc-gpsd.php
- [GITLAB-SOURCE] NTPSec source on Gitlab

Changelog

1.1, Nov 2013
: Initial release.

1.2, Aug 2014
: Note that NetBSD now has PPS support.

1.3, Aug 2014
: Add a note about the GR-601W.

1.4, Dec 2014
: Cleaned up Bibliography

2.0, Feb 2015
: More about troubleshooting PPS delivery. Folded in Sanjeev Gupta's Calibration Howto describing the loopstats-based procedure. Added preliminary information on PTP.

2.1 Mar 2015
: More on PTP. Added link to Jaap Winius's page on GPS-18 setup.

2.2 Mar 2015
: Detailed explanation of NTP has moved to a new page, Introduction to Time Service.

2.3 Mar 2015
: Use the NTP accuracy estimate from RFC 5905.

2.4 Mar 2015
: Removed some typos, corrected formatting, and minor changes. A bit more specificity about root vs. non-root.

2.5 Apr 2016
: Note the existence of the GR-701W.

2.6 May 2016
: New section on GPS time. Note the existence of the GR-801W. Describe the special timeserver build of GPSD. Recommend NTPsec. Add Macx-1 link. Add sections on ARP and temperature problems

2.7 June 2016
: Add section on avoiding power saving.

2.8 July 2016
: Mention required version of gpsd Fix Typos.

2.9 August 2016
: Fix typos.

2.10 September 2016
: Mention ntpviz Recommend minpoll=maxpoll=0 for PPS refclocks Recommend NTPsec.

2.11 June 2019
: Check all links, and update to https where possible

---

1. RFC5905 says "a few tens of milliseconds", but asymmetric routing can produce 100mSec offset

Version 2.11
Last updated 2019-10-13 02:49:56 UTC