An Approach to Primary NTP by Using the LINUX Kernel

Carsten Rieck
Measurement Technology
SP Technical Research Institute of Sweden
Box 857, S-501 15 Borås, Sweden
carsten.rieck@sp.se

The popularity of NTP is unmatched by any other network timing protocol. A complete NTP implementation is usually realized in user space, but the nature of primary NTP makes it possible, even favorable from a performance point of view, to implement primary NTP within the kernel of a UNIX like operating system. Primary NTP is simply a time stamping interface between the physical clock and a network description in form of NTP-packets. This paper introduces the concept of KNTPD, a Kernel NTP Daemon for use as a Stratum1 only NTP server. It suggests a light weighted implementation using Linux netfilter hooks. KNTPD and its PPS-API is particularly suitable for small specialized devices, such as system on chip (SOC) solutions that integrate into timing products to provide high performance NTP capability. Examples could be industry atomic frequency standards used in the telecom industry, timescale devices, such as phase steppers, or GPS receivers.

I. INTRODUCTION

NTP [1] is the most popular and most mature timing protocol for computer synchronization used today. It has its origin in the late 1970ies and thus is one of the most long-lived Internet protocols. The latest official version of the protocol is NTPv3, which is accepted as a standard in RFC-1305 [2]. NTPv4 is a further development with improved algorithms, where a simplified version called SNTPv4 found its way into RFC4330 [3]. NTP-Software is usually implemented in user space, as for instance the reference implementation available from [4]. The overall performance of NTP is based on the assumption that network delays are symmetric, which they seldom are. Even though asymmetry is the most dominant accuracy limiting factor, this paper concentrates on the capabilities of the serving system in order to maximize throughput and transfer of the source accuracy to the network. The intrinsic weakness of NTP on a global scale will not be discussed.

A typical stratum 1 NTP server comprises a timing hardware interface, a computing platform with network communication capability, a POSIX operating system and the reference implementation [4]. Generally, such a combination is not favorable for small embedded systems, because of the restriction these imply with respect to size and resources. The paper focuses on an alternative Stratum1 only NTP implementation and an improvement of the general timing interface in order to simplify the server design and to maximize the performance. A primary NTP server that interfaces with a UTC time source does not need the entire lower stratum 'overhead' of the protocol. It merely acts an interface between the physical clock and the network where synchronization of the primary server is not the main objective. Apart from sanity checks of the physical clock it is not desirable to interfere with the timing source other than to transfer its accuracy and stability to the network.

TABLE I.
PROPERTIES OF THE DIFFERENT MODES OF NTP WITH RESPECT TO THE TIME SOURCE

	PRIMARY - physical	**SECONDARY** - logical
Nature	local to server	network
Source	a UTC source, e.g. a PPS signal	logical clocks combined with NTP algorithm
Policy	low	high
Time Performance	high (ns, µs)	moderate (µs) - low (ms)
SW Complexity	low	high
Performance Limitation	non real-time kernels, user space, clock interface	network: asymmetry, congestion

1-4244-0647-1/07/$20.00 ©2007 IEEE

NTP in user space adds for example overhead and timing uncertainties through the kernel-user-kernel-space interface and the user space process scheduling. These arguments would be viable to motivate the migration of primary NTP from user space to kernel space, since it offers a more undisturbed environment, which presumably can provide:

1. improved performance, and
2. more simplicity.

But: user space based applications providing networking services are generally preferable to Kernel based variants because:

1. kernel space coding is hard and limited, e.g. no Clib, kernel resources are limited, whereas user space is nearly 'endless' and flexible,
2. policy handling does not belong into the kernel.

Where 1.) mostly addresses programmer's capabilities and applications needs, does 2.) address the distinction between 'mechanism' and 'policy': a cornerstone of UNIX design. Information interpretation is clearly policy and this is what most network daemons do, including NTP. Table I indicates in some points the differences between primary and secondary NTP with respect to the time source. Policy in NTP is mostly applied by clock selection and the NTP algorithm of lower stratum NTP. Putting aside access control, authentication and sanity checks primary NTP applies much less policy than secondary NTP. It mainly provides a mechanism to timestamp and reply packets.

The use of direct synchronization within the kernel has already been proposed in RFC2783, "Pulse-Per-Second API for UNIX-Like Operating Systems"[a], see [5]. This API does not restrict itself to NTP, but applies a similar level of policy as primary NTP and seems to be acceptable to be part of the kernel. This paper goes one step further and proposes to even move the packet handling into the kernel, which still can be seen as a violation of kernel policies, but it can be justified with the advantages it offers: simplicity, flexibility and performance.

II. CONCEPT

Linux is an open source POSIX compliant[b] UNIX like kernel, which has grown in maturity, popularity and acceptance over the last decade. Since the 2.4 series the Linux kernel offers Netfilter as an API for filtering and mangling traversing traffic through the TCP/IP stack. The Netfilter framework [6-8] offers a well defined set of interfaces within the Linux Network stack. It exists at Layer 3 (network) in both IPv4 and IPv6, as well at Layer 2 (link) for the ARP and Bridging facilities. It is usually used for fire-walling and NAT, but is a very powerful tool for manipulating raw data packets in Kernel- (and User-) space. By registering a Netfilter hook NTP code within the kernel can:

1. alter every incoming NTP packet on the fly,
2. deny it further traversal to user space and
3. redirect it to the transmitting queue in order to reply it to the client again.

Figure1 depicts the Linux TCP/IP network stack with the approximate Netfilter hook points. Most suitable to use are the hooks at the network layer, but working at the link layer might provide further performance advantages although with the penalty of increased complexity. Considering IPv4, two hook points are available for KNTPD: `NF_IP_PRE_ROUTING` and `NF_IP_LOCAL_IN`. The following presumes a valid NTP timestamp with relation to one of the system counters, e.g. the TSC, that can be used to interpolate between timestamp updates. Source of time should be a reference pulse on a PPS interface, but can in the simplest case be the system time, which is kept at Stratum0 by other means. As soon as KNTPD gets hold of a packet `skb` it estimates the packets arrival time at the host. Unless the packet is NTP or intended for that host the packet is returned to the stack for continued traversal. If it is a valid NTP packet then KNTPD is free to change IP and UDP information, i.e. to exchange SRC and DST IP:PORT and to modify certain other fields in the IP header. Before KNTPD can modify the NTP header in the UDP payload it must make a reasonable estimation of the transmit time of the packet. After NTP modification new checksums for UDP/IP have to be calculated. By modifying the Netfilter `nf_info` structure KNTPD alters the path the packet travels in the network stack after reinjecting to it. The last thing to do is to signal the hook that KNTPD has taken care of the packet. The listing below shows a typical NTP exchange on the network layer.

```
REQUEST:
45 00 00 4c 00 00 40 00   IP Header 20 bytes
40 11 36 53 01 01 01 29           1.1.1.41 (server)->
01 01 01 24                       1.1.1.36 (server)

00 7b 00 7b 00 38 58 20   UDP Header 8bytes

e3 00 04 fa               from LI to PREC
00 01 00 00               Root Delay
00 01 00 00               Root Dispersion
00 00 00 00               Reference ID
00 00 00 00 00 00 00 00   REFERENCE
00 00 00 00 00 00 00 00   ORGIGINAL
00 00 00 00 00 00 00 00   RECEIVE
c9 fb bb 06 58 60 dc b9   TRANSMIT

ANSWER:
45 10 00 4c 00 bf 40 00   IP Header 20 bytes
40 11 35 84 01 01 01 24           1.1.1.36 (server)->
01 01 01 29                       1.1.1.41 (client)

00 7b 00 7b 00 38 04 98   UDP Header 8bytes

24 02 04 ee               from LI to PREC
00 00 00 0b               Root Delay
00 00 07 db               Root Dispersion
01 01 01 28               Reference ID
c9 fb ba 1f 32 91 31 ec   REFERENCE
c9 fb bb 06 58 60 dc b9   ORGIGINAL
c9 fb bb 07 71 a8 ac 5c   RECEIVE
c9 fb bb 07 71 aa a3 ad   TRANSMIT

diff: transmit-receive = 0x1f7551 = 128849
           => 1/(2^32) s * 128849 = ca 30us
```

[a] also called *nanokernel*
[b] actually towards compliant

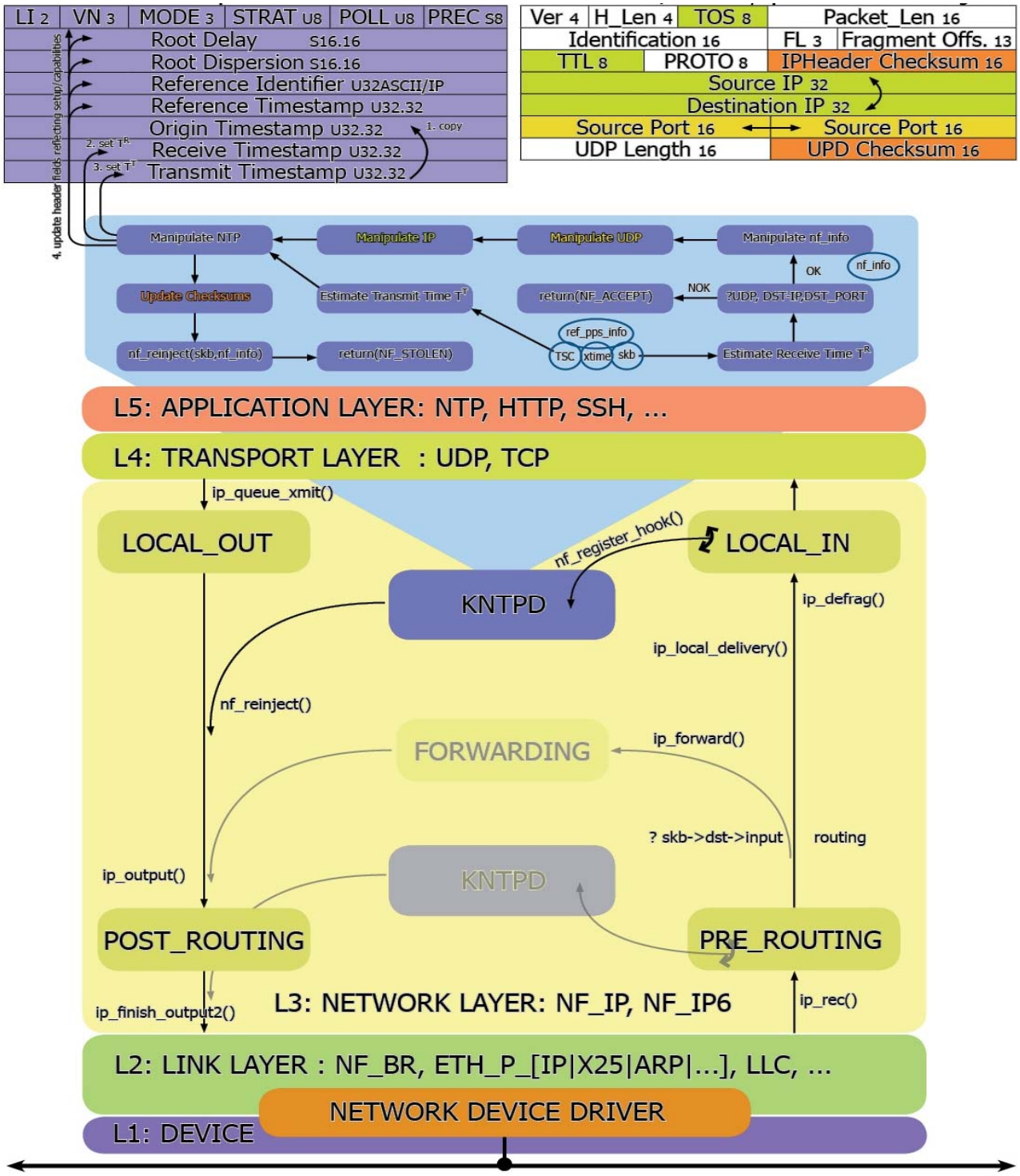

Figure 1. The very simplified Linux TCP/IP stack, with the rough postions of the Netfilter hooks, which can be used to change a packet's content and path. This is a suggestion of how a kernel based primary NTP server could be implemented. The figure shows KNTPD at the possible hook points. In order to further reduce unpredictable delays it is in principle possible to use the concept at L2. KNTPD can even co-exist with userspace NTP acting as an accelerator for serving packets while the userspace daemon is used to act as a network client keeping the system clock in phase and frequency.

III. PPS API

A second measure for improved performance is to offer a kernel API similar to RFC2783, possibly extending RFC2783, for use with the physical PPS interface that connects the physical timing signal and KNTPD. Most of the external clock's stability is usually lost on the interface between clock and the non-real-time kernel due to the fact that the kernel takes its time to react to interruptions, e.g. PPS IRQs. Figure 2 shows an example of the IRQ latency of an unloaded system with a high spread of latency values, which is unacceptable for a high performance timing interface. Two techniques that reduce/avoid latency problems are suggested here:

1. Figure 3: Latency counters - PPS interfaces interrupt the host system in order to relate the external reference to the host clock. A counter on the PPS interface side is used to estimate the latency of the OS answering IRQs. The technique can also be used with polling devices in case IRQs are not available. Many popular timing interfaces used today, such as the famous PPS gadget box, lack the latency counting capability. But latency counting hardware can be attached to many types of different hardware interfaces improving their performance significantly. The OS timing is decoupled from the timing in KNTPD. Software independent counters, e.g. the TSC, are used to interpolate between reference tics. In case host synchronization is required this can easily be achieved with RFC2783.
2. Figure 4: Direct time stamping circumvents latency by instantly generating NTP compatible timestamps during fast device reads. Modern interface technology offers the bandwidth that is needed to implement a few tenths of ns resolution. Even here the system clock is completely unsynchronized and has to be corrected by other means if required.

Several PPS modules can co-exists with each other but only one is governing the KNTPD timing at any time. Redundant local time sources can be used to monitor the active PPS link.

IV. ROADMAP

The current development concentrates on a recent 2.6 kernel with support for PPSkit-light as a base. So far the project has confirmed the general applicability of the concept and plans to fully implement it. Further investigation will be done on a link layer implementation. Both the PPS interface and the packet handling are to be independently implemented as separate kernel modules. The configuration of the daemon and the PPS modules will be done via the `/proc` file system.

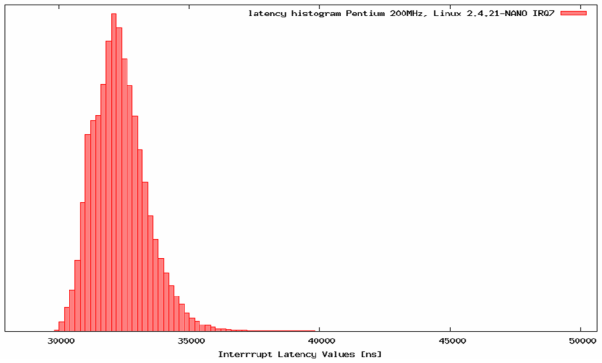

Figure 2. A typical example of the interrupt delay. The tail of the distribution is cut-off: min: 23 µs, max: 9963 µs, median: ca 32 µs.

V. CONCLUSION

Netfilter allows an effective and thin implementation of the required parts of a Stratum1 NTP server within the kernel space. The main advantage of such an approach is performance, which can be seen as improved throughput and a better timing accuracy at the server. KNTPD and its PPS-API are particularly suitable for small specialized devices, such as system on chip (SOC) solutions that integrate into timing products providing high performance NTP capability. Examples could be industry atomic frequency standards used in the telecom industry, timescale devices, such as phase steppers, or GPS receivers as long as they use the Linux Kernel as part of their implementation.

ACKNOWLEDGMENT

To the Open Source Community for their spirit, creativity and all the fancy free software out there! To the colleagues at SP for their input and support! To remember Mattias Nilsson who worked with NTP at SP.

REFERENCES

[1] D.L.Mills, "Computer Network Time Synchronization - The Network Time Protocol", CRC Press, 2006

[2] D.L.Mills "RFC1305 - NTPv3", http://rfc-editor.org/

[3] D.L.Mills "RFC4330 - SNTPv4", http://rfc-editor.org/

[4] http://ntp.org

[5] J. Mogul et al. "RFC2783 - PPS API for UNIX", http://rfc-editor.org/

[6] http://netfilter.org

[7] C.Benvenuti "Understanding LINUX Network Internals", O'REILLY, 2006

[8] R.Russel, H.Welte "Linux Netfilter Hacking HOWTO"

[9] W.R.Stevens "TCP/IP Illustrated, Volume1: The Protocols", Addison Wesley, 1994

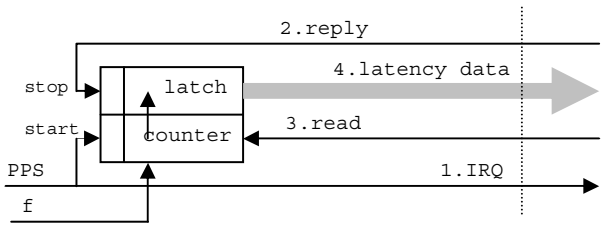

Figure 3. A latency counting PPS interface device: the PPS IRQ triggers the OS to reply, stop the latency counter and relate the read value to a system counter.

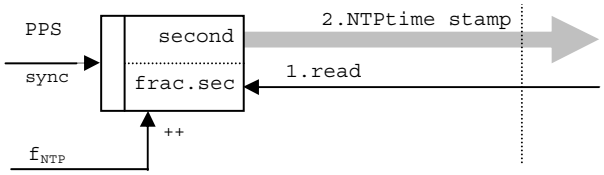

Figure 4. Direct NTP timestamping: f_{NTP} is a synthesized frequency that machtes a binary frequency necessary to increment the counter.