An IEEE-1588 Compatible RADclock

Matthew Davis*, Benjamin Villain†, Julien Ridoux*, Anne-Cécile Orgerie‡, Darryl Veitch*
*Department of Electrical & Electronic Engineering, The University of Melbourne, Australia
davis@student.unimelb.edu.au, {jridoux, dveitch}@unimelb.edu.au
†LUNAM Université, Université de Nantes, IRCCyN UMR CNRS 6597, Nantes, France
benjamin.villain@gmail.com
‡Ecole Normale Supérieure, Lyon, France, annececile.orgerie@ens-lyon.fr

***Abstract*—The *RADclock* is an open source software clock that is highly robust to latency variability. A limitation up to now has been that it could only be used with NTP servers, and was unable to take advantage of IEEE-1588 enabled devices, including PTP masters and NICs with hardware timestamping. This paper benchmarks an early implementation of PTP support for *RADclock*, with and without hardware timestamping. We evaluate performance under both nominal and stressed conditions against alternative software clients *ptpd* and *timekeeper* and find that it compares very well.**

***Index Terms*—software clocks, *RADclock*, *ptpd*, *timekeeper*, IEEE 1588, latency.**

I. Introduction

The availability of a versatile and reliable software clock that is open source is highly desirable. Ideally, such a solution should be able to synchronise to both NTP servers and PTP masters, and be capable of exploiting hardware timestamps when available to achieve higher accuracy, but be accurate and reliable when using software timestamps. It must be robust to the high latency variability induced by network and operating system stress, and be available on major platforms.

No available solution fulfils all of the above needs. By far the most widely deployed is *ntpd* [1], which of course is open source and 'speaks' NTP, but it does not support PTP, and it can become unstable under latency stress [2]. The open source *ptpd* [3] speaks PTP and is available on a number of platforms (FreeBSD, NetBSD, Linux, cClinux), but it does not speak NTP, does not support the use of hardware (i.e. NIC based) timestamping (see however [4] for some early work in this direction, and also [5]) and can also be unstable under stress [6], [2], [4]. An alternative software PTP client is *timekeeper* [7]. It is available for Linux, speaks both NTP and PTP, and uses hardware timestamps, but is a commercial solution, and its performance under software timestamping is unknown. Finally, our own open source solution, *RADclock* [8], speaks NTP and is available on FreeBSD (*RADclock*'s kernel components will be in FreeBSD from version 10.0.) and Linux, but does not speak PTP. It is highly stable under stress, but has not supported the use of hardware timestamps.

This paper trials an early version of support for IEEE-1588v2 within *RADclock*, including limited support for hardware (HW) in addition to software (SW) timestamping of PTP packets. The hardware capability is built on top of the existing Linux support (see work of Cochran et. al. [9], [10]). With these extensions RADclock has the ability to operate, at least to some extent, across each of the (protocol, platform, timestamping) dimensions outlined above. By comparing against alternative implementations across a number of points in this space, under both lightly loaded and stressed scenarios, we show RADclock's potential to fulfil the role as an accurate and robust 'universal' synchronisation solution which is also open source. Along the way we provide interesting benchmarking of other solutions in various environments, including *timekeeper* as a PTP and NTP client with software timestamps. We consider the protocols (NTP, PTP), the platforms (FreeBSD 9.0, Linux 3.2.2), and the clients (*ptpd*2, *timekeeper*, *RADclock*). We do not benchmark *ntpd* since PTP is our main focus, and because it has been done before. As the servo designs of *ptpd* and *RADclock* have been described elsewhere, and *timekeeper*'s is proprietary, we omit them. We restrict to benchmarking over a LAN, since *ptpd* and *timekeeper* are not designed for the wider Internet.

The PTP support for RADclock is preliminary in several respects. First, it is not a full IEEE-1588v2 implementation. It is restricted to master-slave mode and supports only what is needed to allow RADclock to obtain the timestamps it needs from a given master. Second, our implementation makes no use of the Sync messages multicast from the master. Instead, timestamps are collected from periodic (Delay_Req, Delay_Resp) message pairs to recreate a bidirectional exchange paradigm suited to *RADclock*'s existing algorithm [8], which is used without change. Other PTP implementations make use of both kinds of message in (increasingly) roughly equal proportions. In future work we will extend *RADclock*'s feed-forward approach to work with uni-directional data in order to benefit from Sync messages, and in particular, from the low latency of the two-step clock's Follow_Up timestamps.

Despite the above limitations, our implementation is sufficient to allow successful synchronisation to a PTP master with the same high robustness established in previous work [8]. We find that under nominal conditions the accuracy compares well with that of the other solutions, even without the benefit of the Follow_Up timestamps, and very well when HW timestamps are used. Under stressed conditions, we find *RADclock* to be far more robust, in particular when using SW timestamps.

The paper is structured as follows. Section II provides needed background on our experimental testbed and our 1588 implementation. Section III presents the results, a set of comparisons of *RADclock* performance versus that of other clients covering the key dimensions. We conclude in Section IV.

II. Background

In this section we describe our test hardware and software, *RADclock*'s PTP support, hardware and software timestamping, and our measurement methodology.

A. Testbed

Experiments were conducted with two test hosts with identical hardware, *dingo* and *dugong*, each 3.0GHz Xeon Quad Cores, running FreeBSD 9.0 and Linux 3.2.2 (Ubuntu 12.04 LTS) respectively. The integrated NICs were Broadcom Corporation's NetXtreme II BCM5716 Gig-Ethernet (rev 20).

In each host the PTP-capable Intel I350 Ethernet cards were installed, together with the latest *igb* driver (version 3.3.6) provided by the Linux 3.2.2 kernel. On FreeBSD hardware timestamping was not supported, and we used the card as a normal NIC only. On Linux hardware timestamping was supported, however some driver modifications were needed to support *RADclock*'s requirements, as described below.

As an external hardware reference we used a 3.7GE DAG card [11] to timestamp packets passing in and out of the host, via a 100Mbps Ethernet hub as tap. The PPS input to the DAG was provided by a PRS-10 rubidium atomic clock, synchronised to a roof mounted Trimble Acutime Gold GPS receiver, resulting in a final precision around 40 ns off UTC.

We use two reference time servers. The first is a Symmetricom *SyncServer 350* (with roof mounted GPS antenna), used as both an NTP stratum-1 server and a PTP grandmaster (master-slave mode). The second is a PC based stratum-1 NTP server, running *ntpd* with PPS from the atomic clock as input.

Network configuration: the NIC is connected to the tap hub, which in turn is connected to the DAG and a commodity Gigabit Ethernet switch, off which hangs the reference time servers. The minimum host$\mapsto$server network latency, as measured between the SyncServer 350 (using the Delay_Resp and Follow_Up event timestamps) and the DAG, is only 10 µs from the host to the server (outgoing direction), and 18 µs in the incoming direction.

B. RADclock PTP support

The *RADclock* synchronisation algorithms are currently based on a client-initiated bidirectional timestamp exchange paradigm, as commonly used by NTP clients. Here a timing message, initiated by the client, is timestamped by the client as it leaves (T_a), by the server as it arrives (T_b) and then leaves (T_e), and finally at the client as it returns (T_f), resulting in a four-tuple *stamp*: $\{T_a, T_b, T_e, T_f\}$ available at the client, which forms the basic unit of input to the algorithm. The PTP message types were not explicitly designed with this paradigm in mind, however they can be used to support it as follows, through using the Delay_Req and Delay_Resp messages.

$\mathbf{T_a}$: the host client generates and timestamps Delay_Req.
$\mathbf{T_b}$: the master timestamps the arrival of Delay_Req. Being an *event* message, the timestamp is accurate, and is returned to the client via the Delay_Resp message.
$\mathbf{T_e}$: we use Delay_Resp as the returning side of the bidirectional exchange. As a *general* message, it is not timestamped at the master. Instead we set $T_e = T_b + d^{\rightarrow}$, where $d^{\rightarrow} > 0$ is a constant representing the minimum time taken to generate and then send Delay_Resp.
$\mathbf{T_f}$: the client timestamps the arrival of Delay_Resp.

Our use of a virtual surrogate for T_e requires further discussion. First, note that in the case of NTP servers where T_e and therefore $d^{\rightarrow}$ are known, we find that $d^{\rightarrow}$ is the range $\approx [15, 100]$ µs depending on server, and on a given server has low variability (interquartile range (iqr) of 5-20 µs). This typically only accounts for a fraction of the in-server latency, for example in PC's, T_e is a userland software timestamp (a one-step clock paradigm) and so the time spent in the kernel is not included. Since in addition the *RADclock* algorithm does not use $d^{\rightarrow} = T_e - T_b$ for filtering nor does it subtract it out from round-trip-times (RTT), its replacement by a constant has very little impact on the synchronisation algorithm's behaviour compared to the usual case with well behaved NTP servers. Effectively, part of the in-server latency is simply reallocated (implicitly) to the network return path to the client. The only real impact is that the path asymmetry A (the difference in the minimum one-way delays) will be smaller than it would have been, as $d^{\rightarrow}$ will underestimate the true value, which adds a constant error to the clock (path asymmetry of A imparts a constant error of $A/2$ on a clock). By using the DAG to accurately measure the asymmetry component residing in the network plus server, this effect can be evaluated in our testbed.

In this paper we set $d^{\rightarrow} = 5$ µs, a very conservative value well under what we have observed in NTP servers of various types. Crucially, it underestimates the true minimum time to generate and send the Delay_Resp, and so will not generate a causality violation at the client.

The actual timestamping of PTP packets by *RADclock* is achieved by *RADclock*'s existing modified *bpf* mechanism (see below). However, modifications to *RADclock*'s 'network layer' were needed to support the above bidirectional scheme, in particular since the use of different port numbers on the incoming and outgoing directions, and multicasting, adds greater complexity compared to NTP. A reliable matching mechanism for timestamps from the two directions was written, based on a synchronizing thread taking independent timestamp queues maintained by per-direction threads, and safely merging them into a time-ordered queue of valid four-tuple stamps, suitable as input to the algorithm itself. The scheme is robust to simultaneously active clients, multiple PTP slaves and masters on the network, and lost, duplicated and reordered messages.

As mentioned earlier no modifications to *RADclock*'s synchronisation algorithm were needed. Some parameter values however were altered to match a typical PTP environment. With a period of the order of 1[sec] for Delay_Req messages, we set the *offset_win* [8] parameter controlling drift estimation to 256[sec] compared to the more usual 1024[sec] used with periods of 16[sec] or more.

C. Timestamping Methodology

Software timestamping is the only option available for *ptpd*, and traditionally for *RADclock* (and *ntpd*). Hardware timestamping is possible with *timekeeper* on certain hardware under Linux, and now under *RADclock* for the Intel I350, also under Linux.

RADclock **SW:** Under both FreeBSD and Linux, *RADclock* timestamps the UDP packets carrying the Delay_Req and Delay_Resp messages, or NTP packets, via the kernel's *bpf* mechanism. These timestamps are 'raw', that is hardware counter readings (here of the TSC register), made off copies of the associated packets. They enjoy low latency under each platform, although the implementations and detailed characteristics are quite different. Kernel patches enable *RADclock* to access these raw timestamps from userland.

RADclock **HW:** The *igb* driver for the Intel I350 NIC can access raw counter timestamps of PTP packets from the card. Under Linux we modify the driver to maintain a 64 bit *FFcounter*, a feed-forward compatible version of the counter required by *RADclock*, namely one which is cumulative and does not roll over [12]. This *FFcounter* nominally counts nanoseconds since initialisation of the NIC, but is freerunning, and has a resolution of 8 ns. The RADclock is based on the driver's *FFcounter* **only**, there is no conversion or interpolation to the TSC or any other counter.

On the incoming side, *RADclock*'s network layer gains access to the *FFcounter* timestamps of PTP packets from *libpcap* via *bpf*, through a new member we add to the *skbuff* (the datastructure that carries all information about a packet).

On the outgoing side *RADclock* is the owner of the application layer socket sending the Delay_Req, and so can pass a line of communication (a pointer) to itself to the driver via the *skbuff*. We modify the driver so that just before passing the Delay_Req to the NIC the driver reads the NIC's counter and forms an *FFcounter* timestamp, then immediately accesses the *errorqueue* of the socket and places the timestamp there (the *errorqueue* is an existing socket mechanism that bypasses the networking layer, normally used to record error events). It is very important to note that this design is 'causality-preserving': the timestamp is guaranteed to be made before the packet is sent.

ptpd **SW:** Under both FreeBSD and Linux *ptpd* uses the *so_timestamp* mechanism, which gives access, when the *so_timestamp* socket option is set, to timestamps from the platform's system clock via a received message system call (*recvmsg*). For outgoing packets a *so_timestamp* is obtained via a copy sent back up the IP stack through the multicast loopback interface. If this copy is lost, a userland system clock timestamp is used instead. The system clock is disciplined by *ptpd* itself. We use the latest version of *ptpd*2.

timekeeper **HW:** Typically *timekeeper* is used with NICs which support hardware timestamping. Although the source code is unavailable to us, we believe that NIC counter timestamps are obtained from the Intel I350 via the driver in much the same way as described above for *RADclock*. They are then used to both read and discipline the system clock which produces the final packet timestamps.

timekeeper **SW:** To successfully use *timekeeper* based on SW timestamps, we found it necessary to ensure that HW timestamping was unavailable. To convince the kernel and timekeeper unequivocally of this fact, we bypassed the Intel I350 NIC and used the Broadcom NIC. As the software is proprietary, under SW timestamping we are not certain if the *so_timestamp* or userland timestamping locations are used, or some other approach. Either way, *timekeeper* disciplines the system clock and uses it for timestamping.

D. Experimental Methodology

Our methodology for fair clock comparison derives from that of previous work, notably [13]. Briefly, it is based on a series of UDP test packets sent from the host to an echo server on the LAN and back, as a set of triggers for timestamps made by multiple clocks within the host in both the incoming and outgoing directions, and external to it at the DAG card. To this end all packets are timestamped, not only timing packets. Client clock errors are measured by comparing with DAG.

Results will be presented only for the direction where they suffer the least noise from operating system latency. The best direction is a function of the platform selected and the nature (SW or HW) of the timestamps. Under Linux, polling of the NIC on the incoming side (the default behaviour), while good for throughput performance, is very damaging for the latency noise of SW timestamps. We accordingly compare based on the outgoing side. Under FreeBSD the problem is not so severe but again the latency noise is lower on the outgoing side. With HW timestamping the incoming side is preferable, and each clock can access *exactly* the same event timestamped with the same hardware counter – a perfect shared-event comparison.

Side by side comparison of clocks is ideal both for fairness and to help in interpreting the underlying causes of observed behaviour. Various compatibility issues meant that it was not possible to compare 3 or more in this way. Comparisons across different experiments are possible, but due care is needed as important parameters which impact performance such as server, OS, or NIC typically vary.

For a nominally fair comparison in terms of the volume of timestamp data input to each client, we equalise the polling periods as follows.

ptpd:	1 Sync message per second;
RADclock:	1 Delay_Req or NTP packet per second;
timekeeper:	1 Sync or NTP per second nominally. We observed that 3 NTP packets were actually sent.

Note that *RADclock* is naturally disadvantaged in that each of *ptpd* and *timekeeper* also benefit from Delay_Req messages.

III. Results

In each experiment we first collect data under light load (of both machine and network), then 2 to 4 hours later network stress is added during a 2 hour period, before reverting. Stress is applied through using `scp` to transfer large files (rate capped at 45Mbps) across the hub, resulting in higher delay variability for timing packets. The 90th percentiles of RTT increased from: NTP: $0.14 \rightarrow 10.7$ ms, PTP SW: $0.34 \rightarrow 10.8$ ms, PTP HW: $0.24 \rightarrow 0.40$ ms. The test host is not the target of the transfer so the host itself is not stressed, but each packet is filtered and timestamped by the NIC, which may stress its timestamping path. We use both the Broadcom and Intel NIC. Under BSD polling is disabled by default, and the Intel has good latency characteristics, whereas the Broadcom has considerable noise. Under Linux polling is in effect and both NICs have nasty behaviour, which impacts SW timestamping.

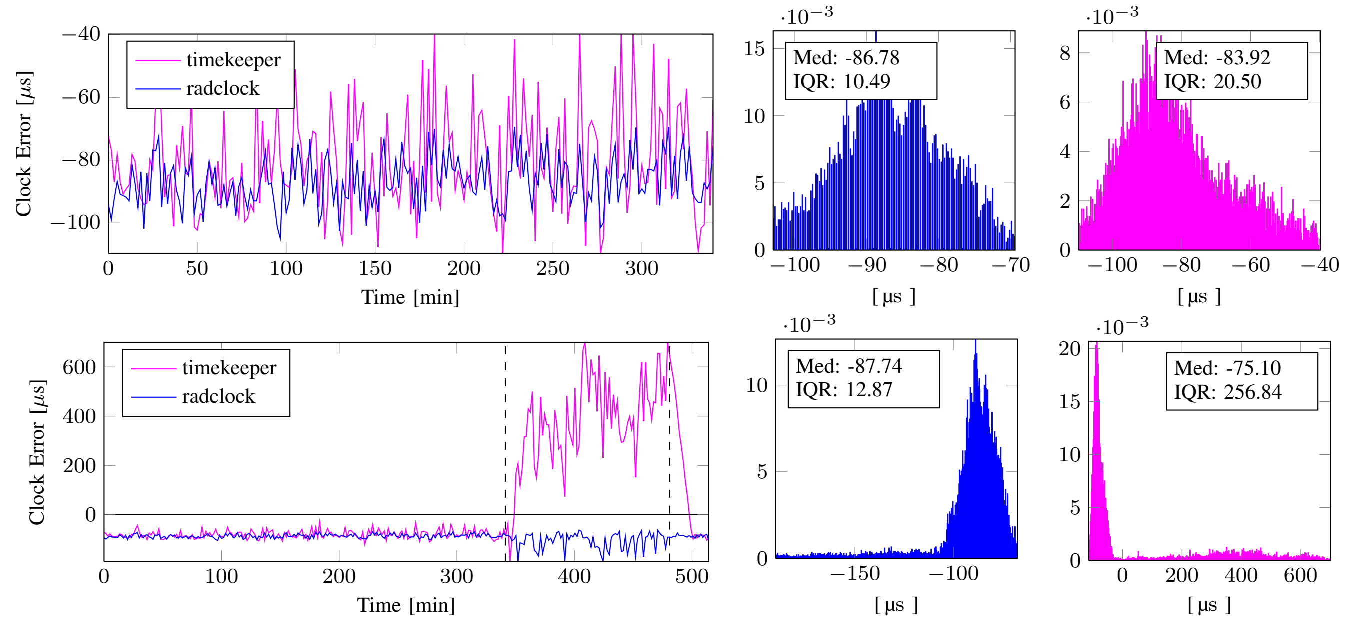

Fig. 1. *RADclock* (NTP) vs. *timekeeper* (NTP) on Linux. NTP server: PC stratum-1. NIC: Broadcom. Timestamping: SW. Top row: timeseries and histograms from initial unstressed period. Bottom: timeseries for full experiment, histograms from stressed portion **only**. Vertical lines mark the stress period.

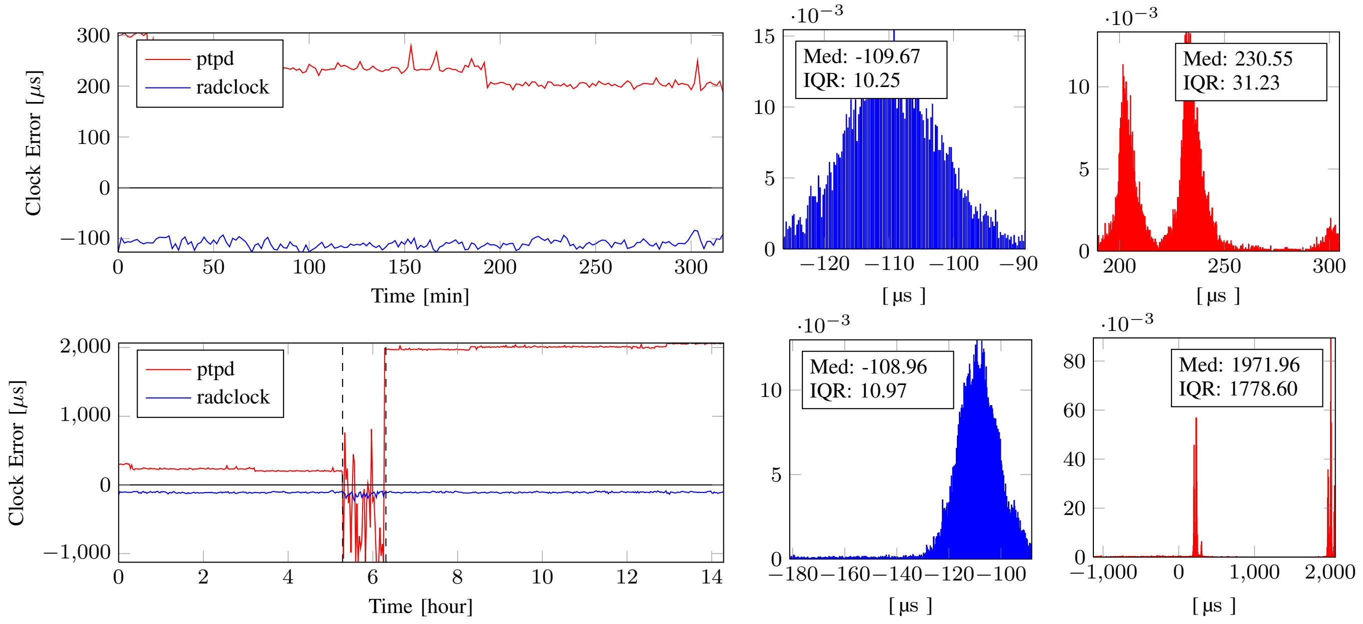

Fig. 2. *RADclock* (NTP) vs. *ptpd* (1588) on Linux. NTP server: PC stratum-1. PTP master: SyncServer 350. NIC: Broadcom. Timestamping: SW.

A. *RADclock Speaks NTP*

In this section we perform comparisons where *RADclock* is acting as an NTP client.

We begin by comparing *RADclock* and *timekeeper* (necessarily on Linux), each synchronizing to the PC based stratum-1 NTP server. Figure 1 shows the resulting timeseries of clock errors and associated histograms. The top row focuses over the nominal period at the beginning of the experiment only. In the bottom row, the timeseries shows the entire experiment before, during (between the vertical lines) and after the stress period, but the histograms are for the stress period **only**. This format is used in all the results below. We see that *timekeeper* has higher error variability (iqr) under nominal conditions than *RADclock*, and was much more affected by stress. The *RADclock* median error is consistent with the asymmetry seen by DAG.

Still on Linux, in Figure 2 we compare the performance of *ptpd* (necessarily as a PTP client with SW timestamps), using the SyncServer 350 as a master, against *RADclock* in the same configuration as before. When the stress period begins we see a dramatic level shift in *ptpd*'s error of 2 ms after some oscillations, and even after it is over *ptpd* does not recover to its previous level. In contrast *RADclock* is only mildly affected, with almost identical iqr, and recovers seamlessly. Nominal median errors reflect the different timestamping locations.

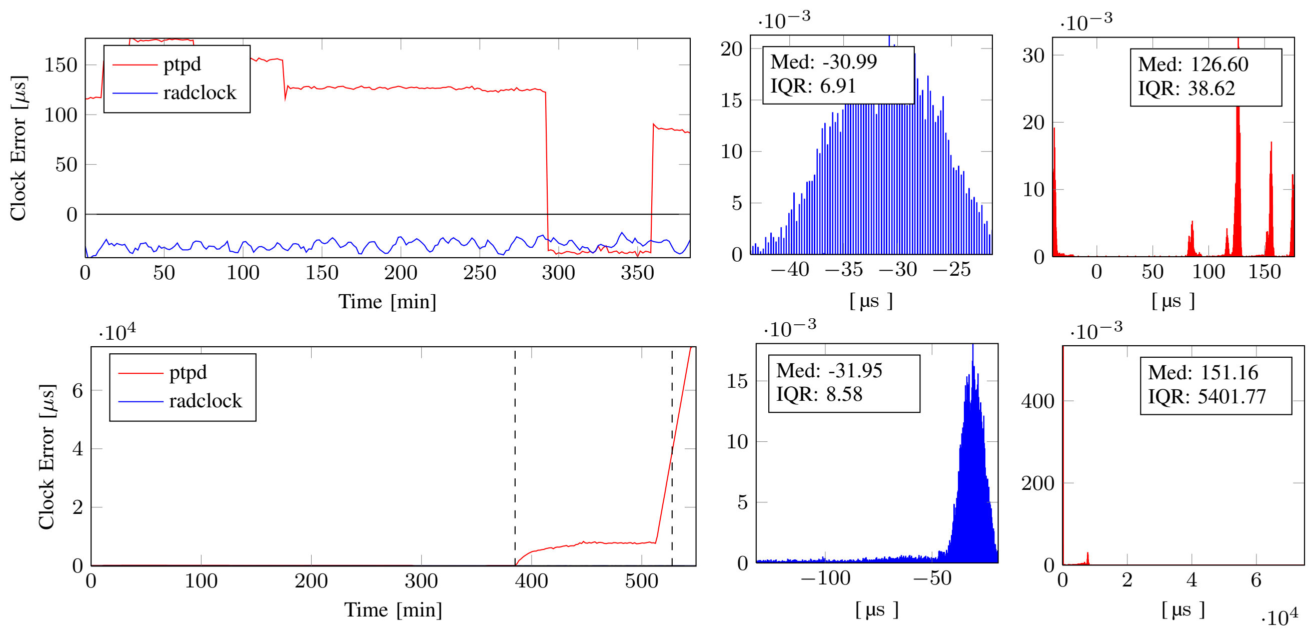

Fig. 3. *RADclock* (NTP) vs. *ptpd* (1588) on BSD. NTP server: PC stratum-1. PTP master: SyncServer 350. NIC: Intel. Timestamping: SW.

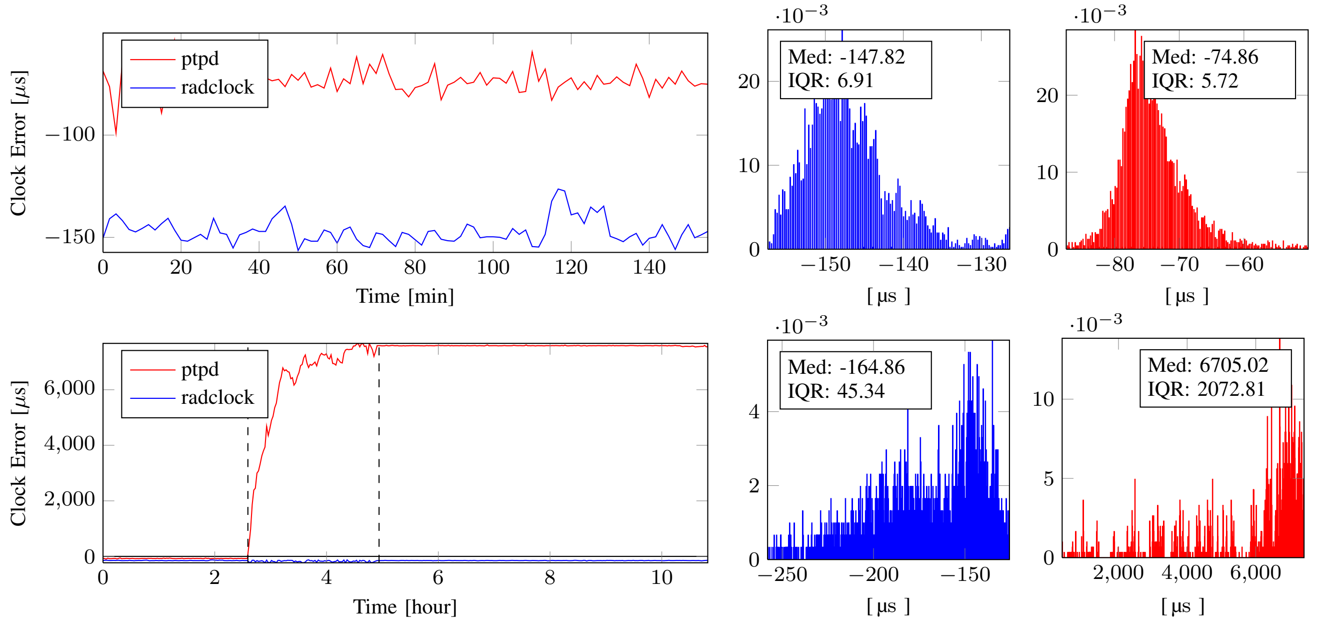

Fig. 4. *RADclock* (1588) vs. *ptpd* (1588) on Linux. PTP master: SyncServer 350. NIC: Broadcom. Timestamping: SW.

Again with *ptpd*, with the same master and with *RADclock* again in the same configuration, but now on FreeBSD, Figure 3 shows a jump in *ptpd* error well beyond the millisecond range in response to the stress, and quite complex behaviour even before it. Afterward an instability sets in and error grows linearly. As on Linux *RADclock* is largely unaffected.

The performance of *RADclock* as an NTP client seen here is consistent with earlier published work, including comparisons with *ptpd* [6]. Re-examining it here is worthwhile because the specific comparisons are new, *ptpd* has undergone considerable development in the interim, and the comparisons with *timekeeper* have not been made before. In the next section we test *RADclock* under entirely new conditions.

B. *RADclock Speaks 1588 with SW timestamping*

We now perform a comparison where *RADclock* is acting as a 1588 client with SW timestamping, with the SyncServer 350 as master.

Figure 4 shows that the *RADclock* PTP implementation is working, and that its performance is comparable in iqr to that of *ptpd* under nominal conditions. The difference in median error can be attributed to the different timestamping locations together with the additional asymmetry associated with our use of Delay_Resp as described in Section II-B. Under stress both clocks are significantly affected, but *ptpd* much more so, with errors growing to 6 ms, and a failure to recover when nominal conditions returned.

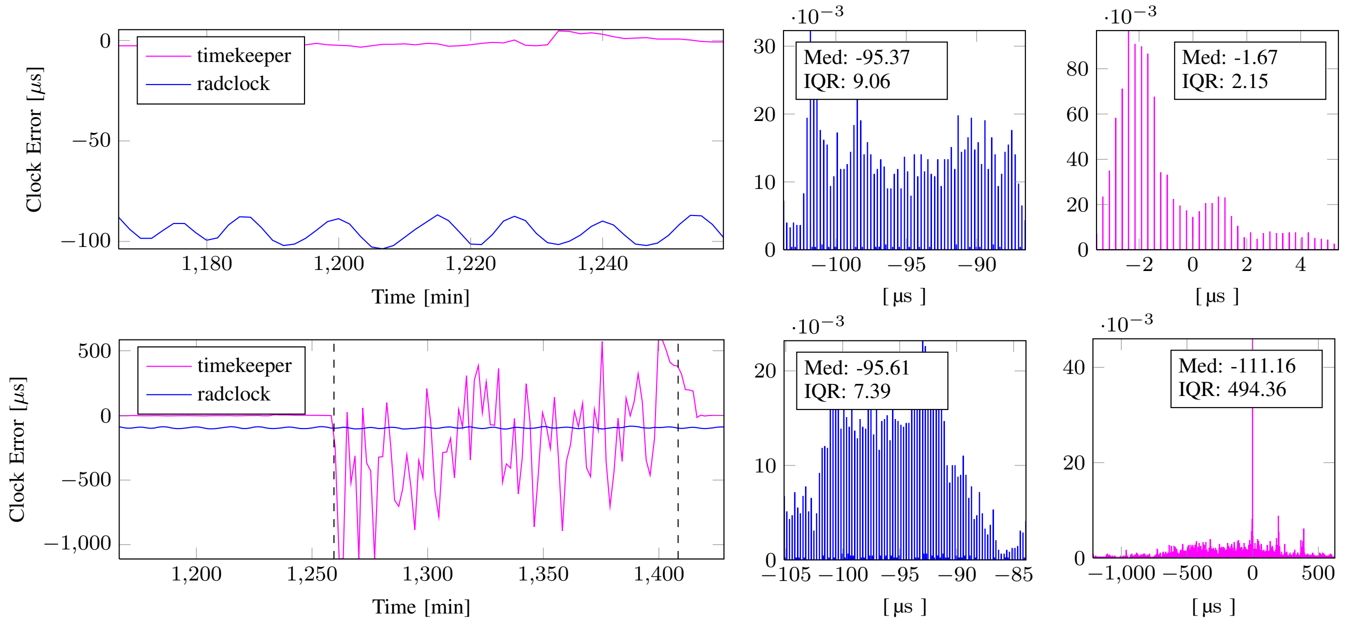

Fig. 5. *RADclock* (1588, hardware) vs. *timekeeper* (1588, hardware) on Linux. PTP master: SyncServer 350. NIC: Broadcom. Timestamping: HW.

C. *RADclock Speaks 1588 with HW timestamping*

Finally, we perform a comparison (necessarily against *timekeeper* on Linux and and the Intel I350 NIC) where *RADclock* is acting as a 1588 client with HW timestamping. Note here each client uses precisely the same hardware timestamps for the incoming UDP test packets, so the following comparison achieves perfect 'simultaneous reading' of the two clocks.

Figure 5 shows that the *RADclock* PTP implementation with hardware timestamping is working. Under nominal conditions, at 9 µs of iqr it is performing less well than *timekeeper*'s 2 µs, which is not surprising given its lack of use of Follow_Up messages and other disadvantages (see end Section II-D). Under stress its performance is essentially unaffected, whereas *timekeeper* suffers errors of the order of 1 ms. The -95 µs median error of *RADclock* corresponds closely to the network+server asymmetry of -160 µs seen by DAG ($-160/2 = -80$).

IV. Conclusion

This paper has two aims, to describe and benchmark a first implementation of IEEE 1588 for *RADclock*, and to show how, via a set of comparisons against other clients populating the dimensions of protocol (PTP, NTP), timestamping (hardware, software), platform (Linux, FreeBSD), and latency environment (nominal, stressed), that *RADclock* is capable of performing respectably in all cases. We showed in particular that its high latency robustness carries over as expected (since the algorithm is unchanged) from NTP to PTP, despite the inadequacies of the current PTP implementation, in particular the fact that Sync messages and the benefits of the PTP two-step clock (the Follow_Up message) are not as yet exploited.

The most important result was that the robustness of *RADclock* was seen in all cases to be much higher than that of the other clients, even *timekeeper* using hardware timestamping, while its performance under nominal conditions was generally comparable if not better.

For future work, we will develop feed-forward compatible approaches to exploiting the two-step clock.

RADclock packages for Linux and FreeBSD, software and papers, can be found at http://www.synclab.org/radclock/.

V. Acknowledgements

We gratefully acknowledge the donation of the 3.7GE DAG from Endace, and Symmetricom's loan of the SyncServer 350. This work was supported in part by a grant-in-aid research grant from Symmetricom Inc..

References

[1] D. L. Mills, *Computer Network Time Synchronization: The Network Time Protocol*. Boca Raton, FL, USA: CRC Press, Inc., 2006.

[2] J. Ridoux, D. Veitch, and T. Broomhead, "The Case for Feed-Forward Clock Synchronization," *IEEE/ACM Transactions on Networking*, vol. 20, no. 1, pp. 231–242, Feb. 2012.

[3] The Precision Time protocol (PTP), ptpd. http://ptpd.sourceforge.net/.

[4] P. Ohly, D. N. Lombard, and K. B. Stanton, "Hardware Assisted Precision Time Protocol. Design and case study." in *Proceedings of LCI International Conference on High-Performance Clustered Computing*. Urbana, IL, USA: Linux Cluster Institute, April 2008, pp. 121–131.

[5] Hardware assisted PTPd. http://home.mit.bme.hu/~khazy/ptpd/, 2010.

[6] J. Ridoux and D. Veitch, "The Cost of Variability," in *ISPCS'08*, Ann Arbor, Michigan, USA, Sep. 24-26 2008, pp. 29–32.

[7] Timekeeper software, FSMlabs. http://www.fsmlabs.com/.

[8] D. Veitch, J. Ridoux, and S. B. Korada, "Robust Synchronization of Absolute and Difference Clocks over Networks," *IEEE/ACM Transactions on Networking*, vol. 17, no. 2, pp. 417–430, April 2009.

[9] R. Cochran and C. Marinescu, *Design and implementation of a PTP clock infrastructure for the Linux kernel*. IEEE, 2010, p. 116-121.

[10] R. Cochran, C. Marinescu, and C. Riesch, *Synchronizing the Linux System Time to a PTP Hardware Clock*. IEEE, 2011.

[11] Endace, "Endace Measurement Systems. DAG series PCI and PCI-X cards," http://www.endace.com/networkMCards.htm.

[12] T. Broomhead, J. Ridoux, and D. Veitch, "Counter Availability and Characteristics for Feed-forward Based Synchronization," in *ISPCS'09*, Brescia, Italy, Oct. 12-16 2009, pp. 29–34.

[13] J. Ridoux and D. Veitch, "A Methodology for Clock Benchmarking," in *Proc. Tridentcom*. Orlando, FL, USA: IEEE Comp. Soc., May 21-23 2007.