Analysis and Compensation of Latencies in NTS-secured NTP Time Synchronization

Martin Langer, Kai Heine, Dieter Sibold and Rainer Bermbach

Full Citation of the original article published by IEEE:

M. Langer, K. Heine, R. Bermbach and D. Sibold, "Analysis and Compensation of Latencies in NTS-secured NTP Time Synchronization," *2020 Joint Conference of the IEEE International Frequency Control Symposium and International Symposium on Applications of Ferroelectrics (IFCS-ISAF)*, Keystone, CO, USA, 2020, pp. 1-10, doi: 10.1109/IFCS-ISAF41089.2020.9234871.

Available at:
https://doi.org/10.1109/IFCS-ISAF41089.2020.9234871

Analysis and Compensation of Latencies in NTS-secured NTP Time Synchronization

Martin Langer
Ostfalia University of Applied Sciences
Wolfenbüttel, Germany
mart.langer@ostfalia.de

Kai Heine
Ostfalia University of Applied Sciences
Wolfenbüttel, Germany
ka.heine@ostfalia.de

Rainer Bermbach
Ostfalia University of Applied Sciences
Wolfenbüttel, Germany
r.bermbach@ostfalia.de

Dieter Sibold
Physikalisch-Technische Bundesanstalt
Braunschweig, Germany
dieter.sibold@ptb.de

***Abstract*—Many time synchronization services use the Network Time Protocol (NTP), which resides in the upper OSI layers and is thus usually implemented in software. However, software-related runtimes reduce the synchronization accuracy and are further influenced by the cryptographic protection of time messages using the Network Time Security protocol (NTS). This paper examines these runtimes and shows the effects on popular NTP implementations. After separating these latencies into their components, we present different approaches for compensation by modifying the NTP timestamps. For non-correctable latencies, this paper provides mitigation strategies to improve the synchronicity.**

Keywords—NTP, NTS, time synchronization, security, analysis, latencies, runtimes, compensation, correction

I. Introduction

Today, the number of computers and machines interacting with each other is ever increasing. The interconnection of devices enables complex processes and working operations. This usually requires time synchronicity between them, for which reason time protocols are often employed. A well-known representative is the Network Time Protocol (NTP) [1] that synchronizes computer systems with a time server via packet-based networks. In conjunction with the Network Time Security protocol (NTS) [2], which is soon to be published as an RFC standard, NTP messages can additionally be protected against packet manipulation. The synchronization of a computer with a time server accessible via the Internet typically provides accuracies in the single-digit millisecond range. If the time server is located in the local network, accuracies in the lower microsecond range are also possible, depending on the topology and the network load. Even higher accuracies achieves the Precision Time Protocol (PTP) [3], which provides a synchronicity in the nanosecond range. However, this requires special hardware support and a much higher setup effort. Therefore, in the local network the choice of the time protocol depends on the achievable accuracy as well as on the specific accuracy requirements.

Nevertheless, the synchronization of the clock is affected by errors and subject to both deterministic and stochastic error sources. In particular, runtime fluctuations of NTP messages transmitted over the Internet lead to a reduction in synchronization accuracy. In the local network these are significantly lower, so that errors due to software-related delays dominate and can lead to a systematic time offset. Especially the cryptographic protection of the NTP packets has a negative effect on the accuracy, because the integrity protection of a message can only take place after time information has been written into the NTP packet.

This paper examines the software-based latency and describes the composition of the individual parts. In the course of the paper, Chapter II first gives an overview of NTP and NTS, which serve as a basis for the analysis here. Chapter III then addresses the reasons for the asymmetric runtimes and focuses on the software-based share. The latencies occurring here are then separated and described as error terms for the time offset calculation in NTP. Furthermore, this section discusses general possibilities for the modification of NTP timestamps and presents solution approaches with the respective advantages and disadvantages. Afterwards we depict various measurements in Chapter IV. These demonstrate the achievable synchronicity of uncorrected NTP implementations and the magnitude of the software latencies. In comparison, measurements with the suggested correction approaches under different hardware conditions follow. Chapter V considers non-correctable software runtimes that influence the synchronicity and provides countermeasures for mitigation. Chapter VI illustrates the magnitude and consequences of software latencies and finally discusses the effectiveness of the correction mechanisms.

II. Preliminaries

This chapter briefly describes the functionality and properties of the Network Time Protocol and the securing of the time information by using NTS.

A. Network Time Protocol (NTP)

When synchronizing the time of computer systems, packet-based time distribution protocols are typically used. One of the oldest and widely used mechanism for that is the Network Time Protocol, which was originally developed by D. L. Mills in 1985 [4]. In 2010, the Internet Engineering Taskforce (IETF) standardized the revised version 4 (NTPv4) as RFC 5905 [1], which is currently the latest version.

NTP uses the connectionless transport protocol UDP to transmit time messages on port 123. It supports both one-way time synchronization (broadcast) as well as two-way time synchronization (unicast) and is therefore very flexible. The established communication architecture bases on a hierarchical approach, whereby the individual levels are known as *strata*. The highest level, stratum 0, represents the time source and is typically an atomic clock or a suitable Global Navigation Satellite System (GNSS) receiver. Time servers on stratum 1 distribute that time to the next lower stratum 2, on which clients and also further time servers may be located. With increasing stratum number, the achievable synchronization accuracy of a client in relation to the time source on stratum 0 decreases. For unicast communication, the calculation of the time difference between client and server

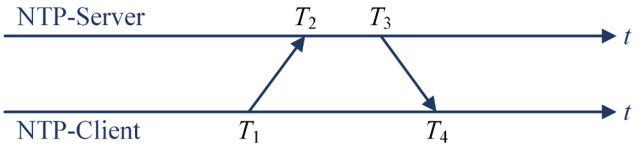

Fig. 1. Capturing of the four timestamps in NTP

bases upon four timestamps. The client sends a request to the server at time T_1, which is received there at time T_2. After processing of the message, the server sends back the modified packet as a response at time T_3 that the client receives at time T_4 (see Fig. 1). With these four timestamps, the client can now calculate the Round-Trip-Time (RTT) δ (1) of the packet, as well as the time offset θ (2) to the server and thus adjust the local clock.

$$\delta = (T_4 - T_1) - (T_3 - T_2) \quad (1)$$

$$\theta = ((T_2 - T_1) + (T_3 - T_4)) / 2 \quad (2)$$

The increasing demand for security in the recent years also comprises time information, for which reason NTP already provides two protection mechanisms. However, both of them are barely applied. The pre-shared key scheme in [1] uses symmetric encryption and is still secure. Since the keys between client and server must be manually configured upfront, this approach scales poorly. The following Autokey procedure [5] was intended to remedy this situation. It scaled significantly better, but a security analysis in 2012 [6] revealed serious design flaws. For this reason, most NTP communication is still insecure and vulnerable, today.

B. *Network Time Security (NTS)*

The malfunctioning of existing security measures for NTP led to the development of the Network Time Security protocol [2], which is about to be finalized as an RFC standard. NTS is a security extension for time protocols with a current focus on NTP. Among other features it provides authenticity, message integrity, unlinkability and key freshness. The communication structure decomposes into two main phases (see Fig. 2), which are implemented by different sub-protocols.

The first phase is for negotiating security parameters and the exchange of key material. The employed NTS-Key Establishment protocol (NTS-KE) [2] uses a TLSv1.3 connection [7] as a basis for the encrypted transmission of these data. Certificates allow the client to verify the authenticity of the server during the TLS handshake. After successful negotiation, the client receives a set of cookies with which the time server can later check the integrity of the NTP requests. These cookies are also encrypted by the time server and contain all the information that allow the server to work stateless. Once the security parameters and key material have been exchanged, the client and server properly close the TLS channel and thus complete the first phase. During the second phase the NTP protocol is now used, which remains unchanged in terms of communication. The client first generates an NTP request packet, adds a cookie and secures the whole message with NTS. The NTS data are embedded in NTP extension fields, so no adaptation of the NTP protocol is necessary. After receiving the NTS-secured NTP request, the server decrypts the parameters contained in the cookie and then checks the packet for manipulation. If the message is intact, the server generates a response packet, adds a fresh cookie and secures it in the same way. After receiving the response, the client also checks the message, then saves the

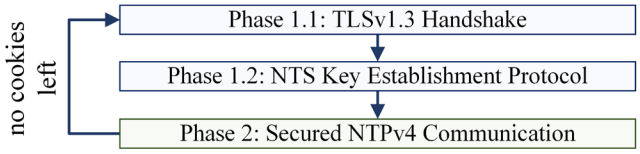

Fig. 2. Phases in NTS secured NTP communication

new cookie for future messages and uses the time information to synchronize the clock. If there are no more cookies left, e.g. due to connection errors, the first phase is repeated.

III. Asymmetries in NTP Communication

This section discusses the causes of asymmetric runtimes in NTP and the effects on the synchronization process. The focus lies on the software-based delays which, after further decomposition, are included as error terms in the time offset calculation. On this basis, approaches for correction follow.

A. *Assumptions and Reality*

The Network Time Protocol contains various mechanisms such as selection, cluster and combine algorithms to achieve the best possible synchronicity. These algorithms allow NTP to select the best source from a bundle of several time references as well as the detection and exclusion of sources with large deviations, also known as *falsetickers*. In addition, it increases redundancy, because if a time source is lost, the next best one is selected. The NTP packets of a chosen NTP time server are subsequently processed by a clock filter algorithm. This algorithm uses a sliding window of eight packets and selects the NTP message with the lowest expected error for time synchronization. These are typically packets that have the smallest delay δ (1) and are therefore most suitable for synchronizing the local clock. Using the four timestamps of a message, NTP can now determine the time difference θ between client and server (2). The subsequent synchronization process is done in conjunction with the Clock Discipline algorithm to control the speed of the clock and thus avoid leaps by hard setting the time. That algorithm uses a software implemented *phase locked loop* (PLL) or a *frequency locked loop* (FLL) to correct phase and frequency of the local clock (typically a crystal oscillator). This minimizes time and frequency errors and stabilizes the local clock, which has to be continuously adjusted due to environmental influences (e.g. temperature changes). If errors due to network fluctuations dominate in NTP communication, the PLL is a suitable solution to stabilize the clock. In contrast, the FLL delivers better results with a dominant oscillator wander [8].

In order to achieve high synchronization accuracy, NTP assumes symmetrical packet runtimes. Accordingly, the transmission times of request and response packets must be identical between client and server. Furthermore, this also means that ideal timestamps in the NTP packet must be assumed (see Fig. 3), which have to be recorded when sending or receiving the first bit of a message. PTP defines the message timestamp point specifically as the beginning of the first symbol after the *Start of Frame* delimiter of an Ethernet frame. In contrast, this is not specified concretely in NTP.

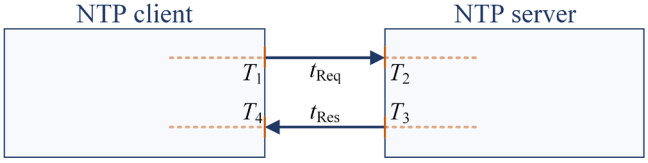

Fig. 3. Assumption of ideal timestamps in NTP

The Network Time Protocol is subject to various influences in a real environment, which lead to asymmetrical runtimes in the RTT (see Fig. 4) and therefore to errors in the time synchronization. These can be separated into three groups: hardware errors ε_{Hw}, network errors ε_{Net} and software-based errors ε_{Sw} (3).

$$\theta = \theta_{True} + \varepsilon_{Hw} + \varepsilon_{Net} + \varepsilon_{Sw} \quad (3)$$

$$\varepsilon_{Net} = (t_{Res} - t_{Req}) / 2 \quad (4)$$

$$\varepsilon_{Sw} = (t_{Sw\text{-}Server} - t_{Sw\text{-}Client}) / 2 \quad (5)$$

The hardware error ε_{Hw} is simplified and consists of a stochastic error term and the clock drift. Stochastic errors are random faults that are caused by natural processes in the hardware and appear as noise. A direct correction is not possible, but fortunately the effects on the synchronicity are rather small. The drift on the other hand describes the continuous deviation of the local clock from an ideal time source. This is primarily dependent on the oscillator built into the hardware. The amount of the error and the stability of the clock is particularly linked to the accuracy class (e.g. TCXO or OCXO), to error tolerances in manufacturing and to aging. Furthermore, this error varies continuously depending on the surrounding temperature. The effect on the offset calculation is negligible, since the time drift here only affects the short time span T_4 - T_1. Moreover, NTP continuously compensates this error with the help of PLL and FLL and adjusts the speed of the local clock of the client to its time server.

As for the second group of errors, the network error ε_{Net} dominates in most cases, where the NTP client synchronizes with a time server via the Internet. This is caused by runtime differences between request and response packet, which typically results in synchronization accuracies in the lower 1- to 2-digit millisecond range. Such asymmetric packet runtimes are caused, among other things, by network load and the forwarding of messages over different paths. Congestion control and other mechanisms in switches and routers (e.g. store-and-forward or queuing) additionally increase the runtime differences in the RTT. Since NTP client and time server have separate clocks, which differ from each other, it is not possible for the client to detect such asymmetries. This consequently leads to an error in the time offset calculation. Subsequently, the resulting time offset error corresponds to half of the runtime difference between request and response packet (4). However, if the time server is located in the local network, the RTT is subordinate and the runtime differences are significantly reduced. In this case synchronization accuracies in the microsecond range are possible, which can be interesting in local networks.

Talking about the last group of sync errors ε_{Sw}, the differences in runtime do not only occur in the network, but begin with the capturing and writing of the timestamps. The NTP protocol is normally programmed in software and usually runs on a non-real-time operating system. Delays that occur here form a software error (see Fig. 5), which is included in the offset calculation in the same way as the network error

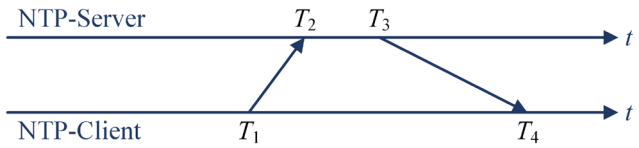

Fig. 4. Different runtimes of NTP messages lead to an asymmetry

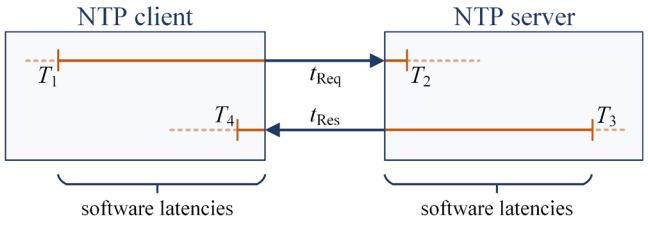

Fig. 5. Asymmetric runtimes within the devices due to the software

(5). This is caused by reading the time at software level that serve as transmit or receive timestamps. The time between the recording of the transmit timestamp and the actual start of the data transmission on the network depends on many factors. These comprise the type and configuration of the operating system, the performance of the hardware, the quality of the NTP implementation and the quality of the timestamps itself. Though the software-based latencies on a given system remain approximately constant, but can differ between client and server, this also leads to an asymmetry of the runtimes. However, these do not simply result in stronger fluctuations in synchronization, as is normally the case with network errors, but rather lead to a systematic deviation. This effect is further amplified by the cryptographic protection of the NTP packets by NTS, since the time-consuming integrity protection can only take place after the timestamping of the packets. When using NTS-secured NTP in the local network, the software error ε_{Sw} therefore dominates, as will be shown by the measurements in chapter III.D.2.

B. Analysis of Software-Based Latencies

In this section we take a closer look at the software runtimes and separate them into logical parts. Here we assume an NTS-secured NTP connection between client and server.

1) Types of Timestamps: The point in time when the timestamps are taken is decisive for the magnitude of the software latency. In Linux environments, a distinction is made between hardware, kernel space and user space for the send and receive timestamps (see Fig. 6). The hardware timestamps T_{Hw} are captured either on layer 1 (PHY) or layer 2 (MAC) of the OSI model. These offer the highest quality, as they do not contain any distortions due to software delays and thus represent almost ideal timestamps. However, this type of timestamping must be supported by the network interface being used. Most computer systems do not offer such support, which is why this type of timestamping is rarely employed in NTP. In such cases, kernel space timestamps T_{So} are used, which the operating system generates at the driver level and are also known as (software-) socket timestamps. Unfortunately, these are affected by runtimes of the underlying network layer and distorted with delays of the operating system. Typically, NTP implementations use this type of timestamping when receiving NTP response messages, because it provides the best accuracy on a software-level. A direct use of the kernel space timestamp as a transmit timestamp is not possible in NTP, because it can only be read after sending the messages. For this reason, user space timestamps T_{Usr} are used, which unfortunately provide the least accuracy in the context of time synchronization. This type of software timestamps is taken directly in the NTP implementation and is therefore subject to many effects and delays of the operating system. Compared to ideal transmit timestamps, user space timestamps additionally contain the runtimes of the hardware, the network layers (OSI level 1-4), the NTP implementation as well as the NTS processing times.

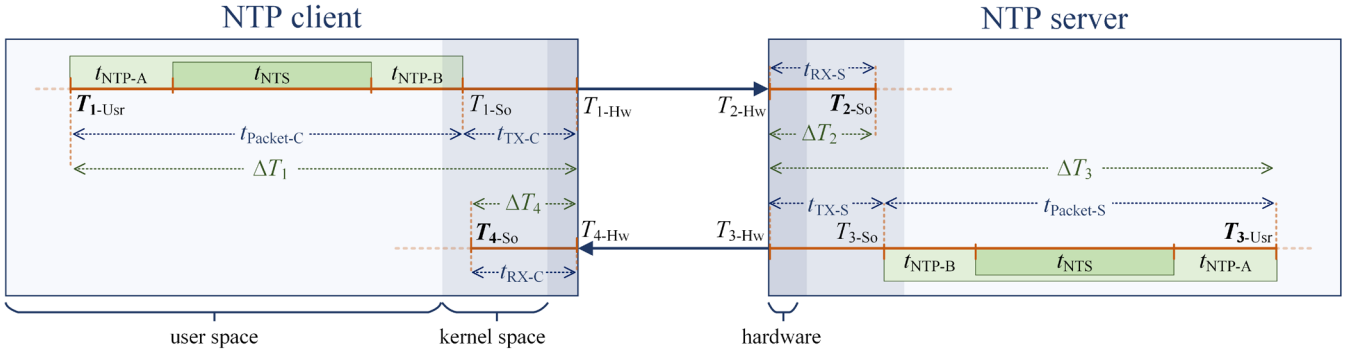

Fig. 6. Detailed composition of the software runtimes

2) NTP Latencies: The software latency begins with the readout of the clock, which serves as the transmit timestamp $T_{1\text{-Usr}}$ for the client side and $T_{3\text{-Usr}}$ for the server side. To perform this, the NTP implementation calls a system Application Programming Interface (API) function (e.g. *clock_gettime*) that already causes initial delays due to runtimes of the operation system. The returned time is then converted to the NTP timestamp format and written to the packet. The runtimes up to this point are depicted in Fig. 6 as t_{NTP-A}. If this message is to be secured, NTP executes the NTS protocol at this point, which will be discussed in the next section. Afterwards, the finalization of the NTP message now follows, where the embedding of the NTS checksum or the padding of the NTP packet takes place. The durations for these actions form the term t_{NTP-B}. Finally, the magnitude of these NTP latencies depends mainly on the implementation quality, possible additional mechanisms (e.g. logging features) as well as on the performance of the hardware. Since the procedure for the construction of NTP packets usually does not change, an almost constant delay can be assumed.

3) NTS Latencies: Further delays are caused by the Network Time Security protocol due to the cryptographic protection of the NTP packet (t_{NTS}). In addition to general delays inducted by the individual implementation of the NTS protocol, the duration of the security process depends on four substantial factors.

The first factor is the amount of data to be secured. Besides the header, each protected NTP packet contains additional NTS contents, which NTP embeds in the form of extension fields. With a stable connection between client and server, an NTP message contains exactly one NTS cookie. In case messages are lost, the client requests up to eight cookies from its server that can be transported in a single NTP message. Moreover, the size of the cookies depends on the structure of the information they contain and the cryptographic algorithm used. The amount of data to be secured by NTS is therefore in a discrete range of 188 to 1428 octets (NTP header + extension fields).

The second factor is the Authenticated-Encryption-with-Associated-Data-Algorithm (AEAD) used to protect the packets. According to the specification, NTS supports at least the AES-based AEAD_AES_SIV_CMAC_256 (AEAD-256) algorithm [9]. In addition to the 256-bit variant, the bit lengths 384 and 512 are usually also available in NTS implementations. Although a higher bit length increases security, it also requires more time to secure the data. Beyond this, the choice of crypto-library (e.g. OpenSSL) influences the resulting runtimes too.

The third factor refers to an essential property of the AEAD algorithms. In addition to providing integrity protection, these allow the optional encryption of data. While the client only generates the integrity checksum of the NTP request packet, the server additionally utilizes the encryption mechanism to transmit new cookies read-protected to the requesting client. However, the encryption needs more time, so that the processing of the same amount of data takes longer compared to the generation of a pure integrity protection. Even with equivalent hardware performance between client and server and identical size of the NTP request and response packets, this already leads to asymmetric runtimes.

The last factor is the performance of the hardware, which has a great impact on the duration of the NTS securing process. Embedded computer systems are mostly designed for low-power applications and often use less powerful processors. In contrast, desktop PCs or server systems are in the high-performance segment and mostly provide both a faster CPU and AES hardware acceleration for cryptographic operations. A difference in performance between client and server thus may increase the asymmetry in the RTT and may have a massive influence on the achievable synchronization accuracy.

4) Network Driver Latencies: Additional delays occur when sending (t_{TX}) and receiving (t_{RX}) NTP messages, which correspond to the time span between kernel space T_{So} and hardware timestamp T_{Hw}. The magnitude of these latencies is also highly dependent on the device driver. Here, the driver captures the transmission timestamps $T_{1\text{-So}}$ and $T_{3\text{-So}}$ in the kernel space immediately before the data is transferred to the hardware. At this point, the NTP packet is already equipped with the UDP and IP header. On the other side, the driver also takes the receive timestamps $T_{2\text{-So}}$ and $T_{4\text{-So}}$ after the registration of incoming data, which the hardware usually announces to the operating system via an interrupt request (IRQ).

C. Error Characterization in the Time Offset Calculation

Having determined the different pieces of the software runtimes, we now can extend the time offset calculation from (2). As we said in the beginning of III.B we can omit ε_{Hw} and concentrate on the communication in the local network disregarding ε_{Net}. So (3) simplifies to (6) containing only the software error ε_{Sw}.

$$\theta = \theta_{True} + \varepsilon_{Sw} \tag{6}$$

We first separate the software error ε_{Sw} from the time offset calculation, since NTP is unable to detect asymmetric runtimes and thus assumes ideal timestamps. The timestamps

of the NTP messages T_1, T_2, T_3 and T_4 must therefore be extended by the software runtimes ΔT_1, ΔT_2, ΔT_3 and ΔT_4 to represent the actual transmit and receive timestamps T_{True}. According to Fig. 6, the software delays must be subtracted from the transmit timestamps and added to the receive ones:

$$T_1 = T_{1\text{-True}} - \Delta T_1 \tag{7}$$

$$T_2 = T_{2\text{-True}} + \Delta T_2 \tag{8}$$

$$T_3 = T_{3\text{-True}} - \Delta T_3 \tag{9}$$

$$T_4 = T_{4\text{-True}} + \Delta T_4 \tag{10}$$

By using (7) to (10) in the time offset calculation (2) and separating the timestamps from the latencies, the terms known from (6) are obtained (11):

$$\theta = \underbrace{\frac{(T_{2\text{-True}} - T_{1\text{-True}}) + (T_{3\text{-True}} - T_{4\text{-True}})}{2}}_{\theta_{\text{True}}} + \underbrace{\frac{(\Delta T_2 + \Delta T_1) - (\Delta T_3 + \Delta T_4)}{2}}_{\varepsilon_{\text{Sw}}} \tag{11}$$

Next we consider ε_{Sw}, which can be further divided into correctable and non-correctable error terms (12):

$$\varepsilon_{\text{Sw}} = \varepsilon_{\text{Packet}} + \varepsilon_{\text{Socket}} \tag{12}$$

The correctable error term $\varepsilon_{\text{Packet}}$ covers the error introduced by the time spans at client and server between user space timestamp T_{Usr} and socket timestamp T_{So}, which are measurable by NTP. It contains all NTP and NTS latencies that occur after timestamping to secure and finalize the NTP packet and results in the respective total runtime t_{Packet} (13):

$$t_{\text{Packet}} = t_{\text{NTP-A}} + t_{\text{NTS}} + t_{\text{NTP-B}} \tag{13}$$

The non-correctable error term $\varepsilon_{\text{Socket}}$ comprises the error introduced by the time spans at client and server between socket timestamp T_{So} and hardware timestamp T_{Hw} and contains the latencies of the lower network layers. This primarily covers the delays caused by the network driver and the operating system when sending (t_{TX}) or receiving (t_{RX}) NTP packets. The missing correction possibility is due to the fact that most systems have no support of hardware timestamping and therefore the magnitude of the error term is unknown. Of course, if hardware timestamping is available, this value can be corrected.

In the following step, we now replace the generic software delays ΔT_1, ΔT_2, ΔT_3 and ΔT_4 with the specific runtime components. According to Fig. 6, the transmission delays ΔT_1 and ΔT_3 comprise the terms t_{Packet} and t_{TX}, while the reception delays ΔT_2 and ΔT_4 only consist of t_{RX}. The indices '-C' and '-S' represent the client and server side respectively:

$$\Delta T_1 = t_{\text{Packet-C}} + t_{\text{TX-C}} \tag{14}$$

$$\Delta T_2 = t_{\text{RX-S}} \tag{15}$$

$$\Delta T_3 = t_{\text{Packet-S}} + t_{\text{TX-S}} \tag{16}$$

$$\Delta T_4 = t_{\text{RX-C}} \tag{17}$$

After using (14) to (17) in (11) and sorting the terms, the software error ε_{Sw} in (12) expands as follows (18):

$$\varepsilon_{\text{Sw}} = \underbrace{\frac{t_{\text{Packet-C}} - t_{\text{Packet-S}}}{2}}_{\varepsilon_{\text{Packet}}} + \underbrace{\frac{(t_{\text{RX-S}} - t_{\text{RX-C}}) + (t_{\text{TX-C}} - t_{\text{TX-S}})}{2}}_{\varepsilon_{\text{Socket}}} \tag{18}$$

Now we have completely determined the error terms so that the time offset calculation can be described like this (19):

$$\theta = \theta_{\text{True}} + \varepsilon_{\text{Packet}} + \varepsilon_{\text{Socket}} \tag{19}$$

The true offset θ_{True} is only achievable if the error terms cancel each other out or are reduced to zero. This is possible with the term $\varepsilon_{\text{Packet}}$, as it is measurable and can be compensated by the usage of a suitable correction method. Since $\varepsilon_{\text{Socket}}$ is not determinable in most cases, the hardware and software conditions of client and server must be identical in order to eliminate this error. With identical hardware, the socket runtimes cancel each other out, so that only low jitter is to be expected. In conjunction with the $\varepsilon_{\text{Packet}}$ correction an almost flawless offset should be achievable. However, if the hardware and software of the client and server differ, then $\varepsilon_{\text{Socket}}$ will dominate the error.

D. Suggestions for Correction

In this section we describe different approaches to compensate the error term $\varepsilon_{\text{Packet}}$ defined in III.C. In order to do this, we start with the general possibilities the client and the server have. Subsequently, concrete solutions will follow, whose advantages and disadvantages will be compared.

1) General approaches for client and server: The correction of the time offset calculation can be handled in NTP by the direct modification of the NTP message timestamps. Under the assumption that hardware timestamps are not available, the correction is limited to the runtime t_{Packet} (13), which only occurs in the transmission paths on client and server side. Consequently, the NTP timestamps T_1 and T_3 must be corrected by the respective runtime t_{Packet} to achieve the accuracy of $T_{1\text{-So}}$ and $T_{3\text{-So}}$ (see Fig. 6).

The transmission latencies of the client can be corrected by three methods. In the first one, the client measures the time span $T_{1\text{-So}}$ - $T_{1\text{-Usr}}$, which equals the runtime $t_{\text{Packet-C}}$. $T_{1\text{-Usr}}$ is the NTP message timestamp T_1, while $T_{1\text{-So}}$ can be queried by the operating system after the packet has been sent. The client stores this duration and can correct the packet after receiving the response by adding $t_{\text{Packet-C}}$ to T_1 (20):

$$T_{1\text{-Corrected}} = T_1 + t_{\text{Packet-C}} \tag{20}$$

The second method is similar to the first one, except that the message timestamp T_2 is adjusted instead of T_1. In order to calculate the same time offset, only $t_{\text{Packet-C}}$ has to be subtracted from T_2, while T_1 remains unchanged. This can be useful if NTS should correct the message timestamps instead of NTP. Since NTP implementations can apply the Data Minimization [10], the T_1 timestamp in the NTP packet could contain random data so that only NTP knows the transmit timestamp T_1. Therefore, the T_2 correction is the only possibility here. The third and recommended method is the direct use of $T_{1\text{-So}}$, which is taken instead of T_1 (= $T_{1\text{-Usr}}$) after receiving the NTP response.

On the server side, corrections are more difficult to implement. The transmit timestamp $T_{3\text{-So}}$ of the NTP response is of course only available after sending and can therefore not be written into the NTP packet beforehand. The latency $t_{\text{Packet-S}}$ caused by NTP and NTS is also only known after the securing process. The message timestamp T_3 can therefore not be corrected by the measured runtimes, otherwise the integrity protection would be lost. The following sections describe in detail how the server can proceed.

2) Direct Timestamp Correction on Server Side: With this method, the client and server modify their respective timestamps independently of each other. The NTP server tries to achieve the best possible compensation by correcting the message timestamp T_3 for server side latencies ($t_{\text{Packet-S}}$). Since this can only be done before the message securing by NTS, the value must be estimated. However, a simple average over the processing time of NTP packets is not sufficient, since the message sizes and the AEAD algorithms used can vary for each NTP packet. Additionally, outliers affect the average value, too. Because the server can operate with a large number of clients, load differences can also lead to different latencies and thus additionally distort a simple average value.

A better alternative is the use of a look-up table, in which the averaged latencies for different configurations are stored (see Table I). A simple table can thus be created on the basis of the specific packet size and the AEAD algorithm used. In NTS, secured NTP messages typically have up to eight packet sizes, depending on the number of cookies they contain. If we use the three possible bit lengths of the AEAD algorithm defined in NTS, the table thus contains 24 cells for secured messages and one additional cell for unsecured NTP messages. The size of this table can be extended to cover additional cases. Several tables can also be deployed depending on other factors (e.g. system load or temperature). The value of each cell is calculated by a moving median that fits the corresponding configuration (packet size and AEAD algorithm). Furthermore, the window size should depend on a defined time interval, since the number of NTP requests per second can strongly vary. The window size should be sufficiently small to be able to react quickly to runtime changes and also large enough to be robust against individual outliers. To minimize excessive loads, servers with a large number of clients should limit table updates. For this reason, the respective cells should not be updated more frequently than e.g. 10 times per second (depending on table size and the specific application). Servers should also save these tables periodically in order to reuse these values when the NTP service is restarted. If the values are not available, the latency of generated dummy packets might be measured at the program start-up to obtain initial values.

With this correction method, the server remains stateless and is able to apply correction values for different NTP packets while keeping the memory usage low. These packets are not changed in size and thus do not constitute an attack potential for a possible amplification Denial of Service (DoS). However, the disadvantage is that the client is not informed of a possible T_3 modification. If the client does not perform a T_1 correction, the asymmetries can even increase, because $t_{\text{Packet-S}}$ equals zero and $\varepsilon_{\text{Packet}}$ becomes larger (see (18)). Moreover, even if a client is aware of these corrections, it cannot reverse them if it is unable to correct its T_1 due to local limitations.

A possible remedy is to transmit a correction information flag. With NTS-secured NTP, client and server can negotiate the activation of the T_3 correction during the NTS-KE phase. This information can then be encoded in the cookies so that the server can decide individually for each client on the use of the correction values. Alternatively, the server can at least publish whether it corrects or not.

TABLE I. EXAMPLE OF A LOOK-UP TABLE

Packet Size	AEAD-256	AEAD-384	AEAD-512
188 octets	20.4 µs	22.5 µs	…
292 octets	24.1 µs	27.3 µs	…
…	…	…	…

3) Sending a Correction Value via EF: A modified variant is the use of NTP extension fields. The process is mostly identical to the direct adaptation from III.D.2. However, the server does not change the timestamp T_3 directly, but transmits this correction value to the client via an additional NTP extension field. In order for this value to be protected, the extension field must be embedded in NTP before NTS performs its integrity protection. The advantage of this method is that the timestamp correction can be applied by the client. It can decide for itself whether a correction should be applied or not. A negotiation between client and server is therefore not necessary. However, a disadvantage is the slightly higher NTP packet size of about 20 octets for the additional data. To prevent amplification DoS attacks, the client must also send such an extension field with empty content to the server so that the request and response packets have the same size. If such an extension field is not contained in the request, the server can assume that the client does not need correction data and does not insert any correction data in the response message.

4) Follow-up / Interleaved Model: A complete correction on the server side is only possible if it communicates the actual software latency $t_{\text{Packet-S}}$ or the transmission timestamp $T_{3\text{-So}}$ to the client. However, this is only feasible if the designed NTP communication structure is slightly modified. The first possibility for this would be the establishment of a follow-up message, which is sent immediately after an outgoing NTP packet. NTS-secured NTP messages lead to further changes to avoid the need to apply integrity protection to both messages (NTP and follow-up). A possible solution for this is the unsecured transmission of the NTP response from the server to the client, which contains the distorted T_3 by the $t_{\text{Packet-S}}$ runtimes. The subsequent follow-up message contains a copy of the previously sent NTP packet and its socket timestamp $T_{3\text{-So}}$. In addition, the follow-up message must be secured with NTS to prevent packet manipulation. Upon receipt of both messages, the client can check the follow-up message for integrity and then compare the contents of the NTP packets. If they match, the NTP packet and the separately transmitted $T_{3\text{-So}}$ timestamp can be used for the time offset calculation instead of T_3. If one of the messages is missing or the contents do not match, the client discards these packets. With this method, vulnerability through amplification attacks must also be taken into account. To counteract this, the request packets of the client must be enlarged accordingly. This again results in the fact that clients may request fewer cookies in case of message loss, in order not to exceed the maximum transmission unit (MTU) size of the network.

Another approach similar to the follow-up principle is the *interleaved mode*. This was originally defined in RFC 5905 [1] as symmetric mode to keep servers on the same stratum synchronous. An IETF draft document [11] also proposes this principle for the NTP unicast mode in order to transmit the $T_{3\text{-So}}$ timestamp. With this approach, some timestamp fields in NTP are handled differently. However, compatibility with NTP clients and servers without *interleaved mode* is maintained. An NTP client in interleaved mode additionally transfers the faulty T_3 timestamp from a previous response packet in the request message. Using this T_3 timestamp in combination with the IP address of the client, the server can retrieve the stored $T_{3\text{-So}}$ timestamp from the previous message and also embed it in the NTP packet. The NTP response packet therefore contains the distorted T_3 timestamp of the current message and the $T_{3\text{-So}}$ of the previous packet. This procedure does not require follow-up messages or extension

fields and is also compatible with NTS. However, the server must be stateful for this to work. With a large number of clients, this could place an additional load on the server as the memory requirement increases. Depending on the request interval, the clock drift may also have to be taken into account.

IV. Measurements

In this section, the presented correction mechanisms are now applied to measure their effects on the synchronization accuracy. For this purpose, uncompensated NTP devices were first used as a reference in a measurement setup and then extended by the correction mechanisms.

A. Uncorrected NTS-secured NTP Synchronization

This first part of the measurements without any timestamp correction examines the synchronicity of NTS-secured NTP communications in an ideal network. We measured the NTS runtimes t_{NTS}, the software latencies t_{Packet} as well as the systematic time offset between client and server.

1) Securing Duration of the AEAD Algorithms: The NTS latencies t_{NTS} consist almost completely of the runtimes that arise during the cryptographic protection of an NTP packet. Therefore, the first series of measurements focused on the AEAD processing times including the necessary allocation and freeing of memory. For this purpose, each NTP message was equipped with one to a maximum of eight cookies and then processed with the respective AEAD algorithm. The number of cookies is defined by the client with its requests. Since the cookie size and the associated NTP message size depend on the applied AEAD algorithm, there are typically 24 possible packet sizes as described in III.D.2. The measurements with a free AEAD library [12] based on OpenSSL 1.1.1b [13] were run on several Linux machines. The hardware platforms varied from a weak Raspberry Pi 1 to a powerful desktop PC. As typical for NTS, the request packets only provided integrity protection, while the response NTP packets also contained encrypted message parts.

Table II shows examples of the duration of the securing process of the NTP packets both at the client and at the server. The configurations are based on the standard case where the AEAD-256 algorithm is used and NTP packets contain one cookie. The amount of data to be secured thus amounts to 188 octets (NTP headers + NTS content). Due to the additional encryption, the duration of the process on the server side increases by about 15% in comparison to the client. The performance of the hardware has a strong influence here, as Table III illustrates. It shows the minimum and maximum values on different devices for the typical bit lengths of the AEAD algorithm supported in NTS. Minimum values refer to the client side, in which 188 octets are to be secured and correspond to NTP messages with a single cookie. Maximum values, on the other hand, are on the server side and contain a data set of 1428 octets that corresponds to eight cookies. The duration of the NTS protection is proportional to the amount of data and the AEAD algorithm used, as shown in the measurement series on the Meinberg microSync^{RX} in Fig. 7. However, the durations for a fixed AEAD algorithm and a fixed number of cookies are almost constant for a given computer system.

TABLE II. Typical Duration of the Securing Process Using NTS

	Client (µs)	Server (µs)
Raspberry Pi 1	87	101
Raspberry Pi 3B	20	24
Meinberg microSync^{RX}	27	31
Desktop PC (i7-6700) w/o AES-NI	1.8	2.4
Desktop PC (i7-6700) w/ AES-NI	1.0	1.1

TABLE III. Duration of the Securing Process (Min/Max Values)

	AEAD-256 (µs)	AEAD-384 (µs)	AEAD-512 (µs)
Raspberry Pi 1	87 / 224	95 / 294	101 / 360
Raspberry Pi 3B	20 / 64	22 / 86	25 / 111
Meinberg microSync^{RX}	27 / 82	29 / 112	33 / 149
Desktop PC w/o AES-NI	1.8 / 8.3	2.2 / 11.7	2.6 / 16.0
Desktop PC w/ AES-NI	1.0 / 1.8	1.1 / 2.2	1.1 / 2.8

2) Correctable Software-based Latencies: The measuring of the software latency t_{Packet} is simple and starts with the acquisition of the transmit timestamp in the NTP implementation. On the client side, this corresponds to the timestamp $T_{1\text{-}Usr}$, which NTP queries via an API function in the user space and then writes into the NTP request message. The actual transmission process of the message defines the end of the measurement. Since hardware timestamps are not available in most cases, socket timestamps are usually used. Chapter V describes the consequences of this action in more details. The resulting software latency is now the difference between the TX socket timestamp $T_{1\text{-}So}$ and $T_{1\text{-}Usr}$.

Fig. 8 shows the results of an NTP server on a Raspberry Pi 3B. This server communicates with several clients and thus allows the observation of latency under different load conditions. For the measurement, the NTP clients and the time server were using an NTS-capable NTP implementation of the Ostfalia University (NTP-O). All clients communicated NTS-secured with the AEAD-256 algorithm and exactly one cookie per NTP message. The request frequency of the clients was set to 8 seconds. The results show an average software latency of 97 µs on the server when communicating with a single client. Each point represents the measured latency of an NTP response message. Due to background processes of the operating system, the measured values fluctuate by about 9 µs. If the server is loaded with several clients, the software latency decreases and settles down to 62 µs for 20 or more clients. In Fig. 8 this effect is visible after 3000s, where the number of clients was increased to 100 and later reset to 1. The results in a histogram (see Fig. 9) show a significantly lower dispersion when the server is loaded with 100 clients. However, this

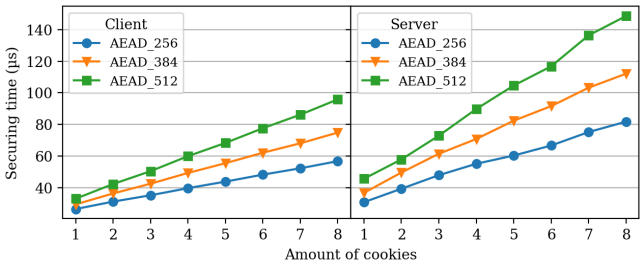

Fig. 7. Securing time of NTP packets (t_{NTS}) based on the number of cookies included and the AEAD bit length.

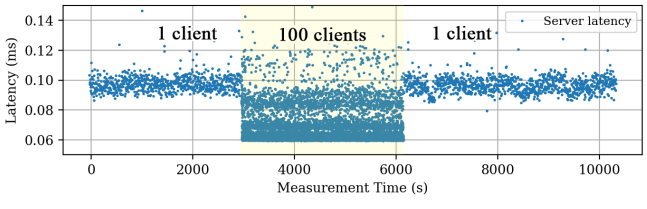

Fig. 8. Software latency t_{Packet} on a Raspberry Pi 3 server at low and high workload

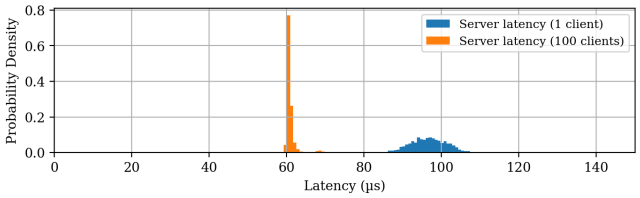

Fig. 9. Histogram of the software latency t_{Packet} of the server (Raspberry Pi 3B) at low and high workload

behavior only occurred with Raspberry Pis during the measurements. The reason for this is an increased process priority. Further measurements on other hardware platforms resulted in mean latencies of 170µs on a Meinberg microSync^{RX} and 53µs on a desktop PC (i7-6700).

3) Systematic Time Offset: The influence of the complete software latencies including t_{TX}/t_{RX} on synchronization accuracy has been measured for the most popular NTP implementations. These include NTPd, NTPsec, Chrony and NTP-O, most of them with NTS support. During the measurement series, all implementations were tested against each other, both unsecured and NTS secured (if possible). Furthermore, all tests were performed on different hardware platforms and varied among each other. Like the measurement of the software latencies, Raspberry Pi 3B devices, desktop PCs (i7-6700) and Meinberg microSync^{RX} devices were used. Furthermore, the devices were connected directly via Ethernet during the measurements to ensure ideal network conditions. The measurement of the actual time offset between client and server can no longer be reliably determined by a pure software solution. For this reason, a hardware-controlled measuring system was applied, which enabled the simultaneous time measurement of the connected devices and limited the measurement error. This made it possible to determine the actual time deviation between the devices, with an uncertainty of 1µs.

The evaluation of the measurement data revealed systematic deviations in all NTP implementations, which amounted to up to 85 µs depending on the constellation. Especially the communication of different NTP services, the use of NTS-secured NTP or large differences in hardware performance between client and server had a negative effect on the synchronization accuracy. But even in configurations with identical hardware and software, systematic deviations of up to 50µs occurred. Fig. 10 shows an example of a measurement recording taken by a client that synchronized itself NTS-secured with a time server. Both client and server used an adapted NTP-O on Raspberry Pi devices and were connected directly via Ethernet, like in the other tests. The packet offset (grey line) in the diagram shows the time offset θ_{NTP} of all received NTP messages calculated by the NTP. After processing these data by the clock filter algorithm, the

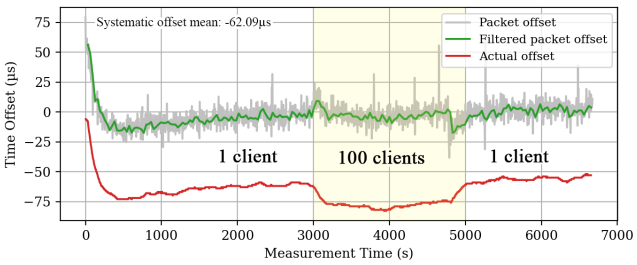

Fig. 10. Systematic time offset (θ_{True} - θ_{NTP}) of a synchronized NTS-secured NTP-O client with changing server load.

NTP messages with the least error are used for time synchronization (green line). Based on the calculated time offset θ_{NTP}, NTP adjusts the local clock, whereby the course slowly approaches zero. However, the actual time offset θ_{True} (red line) measured by the hardware shows a systematic offset of about 62 µs (= *actual offset - filtered packet offset*), which NTP cannot detect due to asymmetric software runtimes. The asymmetry in this measurement was caused by slightly higher runtimes in the client implementation. The additional load of the server with a total amount of 100 clients further decreases the synchronization accuracy. Based on the measured values on different hardware platforms and NTP implementations, it can be confirmed that NTS-secured NTP synchronization has barely any influence on synchronization accuracy when the time server is reached via the Internet and therefore the network influences dominate. However, in the local network these software latencies have a significant influence.

B. *Applying the Correction Mechanisms*

In the following measurements, the correction methods were applied to NTS-secured NTP connections. For this purpose, NTP-O was extended by two correction methods. The first one is the transmission of the correction value from the server to the client (see III.D.3), the second one is the *interleaved mode* (see III.D.4).

1) Identical Hardware on Client and Server: If both the hardware and software conditions on the client and server side are identical, then the full compensation of the software latency is almost possible. In this case the error term ε_{Socket} becomes zero (see III.C). The hardware setup used here corresponds to the procedure from IV.A.3 and uses Raspberry Pis 3B on both sides. While the NTS-secured time synchronization without a compensation mechanism shows a systematic time offset of 62µs (see Fig. 10), the usage of the correction value approach reduces it to about 2µs (see Fig. 11). The use of the interleaved approach achieves similarly low values of 1µs. Both compensation methods also stabilized the calculation of the time offset in NTP. An additional load on the server side with a total number of 100 clients had no impact on synchronicity despite fluctuating server latencies.

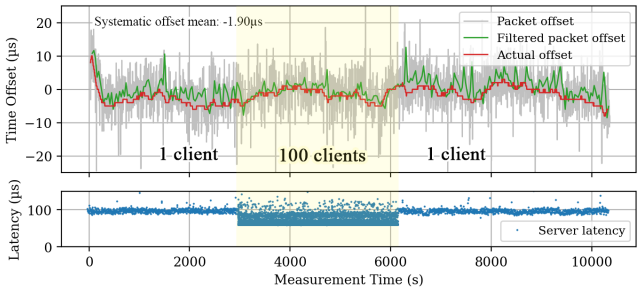

Fig. 11. Systematic time offset of an NTS-secured NTP-O client with changing server load and activated t_{Packet} compensation (see III.D.3)

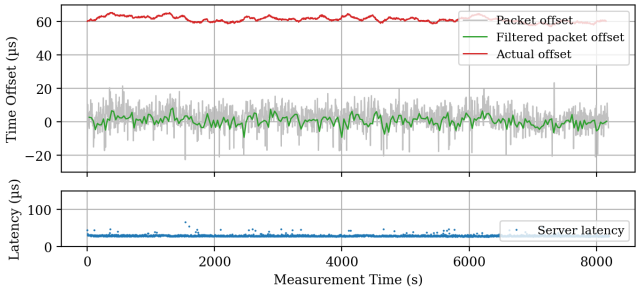

Fig. 12. Effects of hardware differences between client and server on synchronicity with active compensation.

2) Different Hardware on Client and Server: If the hardware platforms vary between client and server, the compensation of the t_{Packet} runtimes in most cases does not lead to a significant reduction of the systematic time deviation and may even increase it. Especially the performance differences of the hardware have a significant influence on the amount of the systematic deviation. Fig. 12 demonstrates the effect where the Raspberry Pi server used in IV.B.1 was replaced by a desktop PC. Despite ε_{Packet} compensation, the deviation for both correction approaches is 61µs for NTS-secured connections and 51µs for unsecured NTP connections. Without a correction mechanism, these values are 19µs for NTS-secured NTP and 41µs for unsecured NTP. The reason for this are the software runtimes between socket timestamp and hardware timestamp that now differ for client and server. In this measurement, the compensation of ε_{Packet} led to an increase in asymmetry, since the error terms described in III.C, which have different signs here, do not cancel each other out anymore (see (18)). The difference of 10µs between unsecured and NTS-secured NTP with active compensation is solely due to the different packet size of 140 octets. As a consequence of the larger messages, the runtimes in t_{TX} and t_{RX} increase. In contrast, the variation of the t_{Packet} latencies have no longer a negative influence on the synchronization accuracy.

V. Mitigation of Socket Runtimes

As could be shown, the latencies in the kernel and driver functions have a significant effect on synchronicity and cannot be easily corrected. The magnitude of these latencies varies not only due to the hardware, but also due to the operating system and the drivers installed on it. However, the handling of the paths for TX and RX are different, which favors an asymmetry in the runtimes. For the TX path, these runtimes are usually shorter, since the transmission process is started by calling the corresponding API function within NTP. This means that the data is processed immediately in the individual layers and then sent out to the network (see Fig. 13). The transmit timestamp is recorded directly before the data is written to the TX buffer. On the other hand, the RX path is usually subject to higher runtimes due to the event that signals the operating system about received data. This can be done both traditionally via interrupt requests and via New API (NAPI). With NAPI no interrupts are used, but the kernel periodically checks for incoming packets (polling) without being interrupted. While this eliminates the overhead of interrupt processing and thus reduces CPU load, it also increases the t_{RX} latency. In this path, the RX socket timestamping takes place during IRQ handling or after the polling process via NAPI.

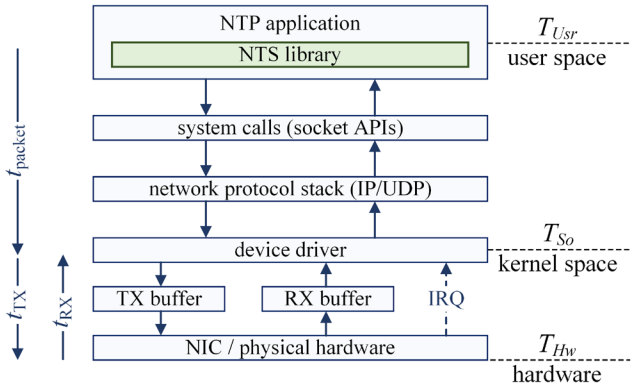

Fig. 13. Transmission (TX) and reception (RX) paths of the operating system for network communication.

Fig. 14 shows the t_{TX} and t_{RX} latencies on the desktop PC used in IV.B.1, which supports hardware timestamping and thus allows the measurement of the delays between socket timestamp T_{So} and hardware timestamp T_{Hw}. With 45µs for t_{TX} and 237µs for t_{RX}, the values here are significantly higher than the delay times due to NTP and NTS (t_{Packet}). Since these amounts usually differ between client and server depending on the hardware, this leads to the systematic time deviation measured in IV.B.2.

On Linux there are several ways to affect the t_{TX}/t_{RX} delays. The first option is the power management setting. The CPU frequency scaling can be governed, among others, to run the processor at its maximum frequency (*performance*), at its minimum frequency (*powersave*) or to dynamically adjust its frequency depending on the load (*ondemand*). For identical operations (e.g. packet securing with NTS) this can lead to small runtime fluctuations and also influence the socket timestamps. To minimize the jitter, the CPU can be run in performance mode. The second way is the activation of the *SO_BUSY_POLL* (busy-poll) option in the socket settings. To achieve a low latency this method significantly reduces the polling interval for new data in the RX buffer. However, as this process increases the load on the CPU, NTP clients should define a timeout for this method. Since the NTP response usually arrives at the client in less than 200ms after sending the request, a suitable timeout can be specified to deactivate the busy-poll option after the timeout has elapsed.

The measurements show that the busy-poll activation greatly reduces the t_{RX} latency and now achieves 25µs (see Fig. 15). An additional activation of performance mode did not result in further improvements and even increased the t_{RX} latency to 290µs when busy-poll was not activated. The t_{TX} latency remained constant at 45µs for all measurements and configurations. With activated busy-poll and interleaved compensation on client (Raspberry Pi) and server (desktop PC), the systematic time offset decreased slightly. For unprotected NTP it was now 43µs and 50µs for NTS-protected NTP.

Further correction possibilities are hardly feasible and require increased effort or additional hardware features. One idea to mitigate the t_{TX} latency, would be the use of a Network Interface Card (NIC) with *launch time* support (e.g. Intel i210). This allows the socket functions to explicitly define a transmission time. However, the successful transmission at the defined point in time highly depends on the network load. Furthermore, the RX runtimes remain untouched, so that a

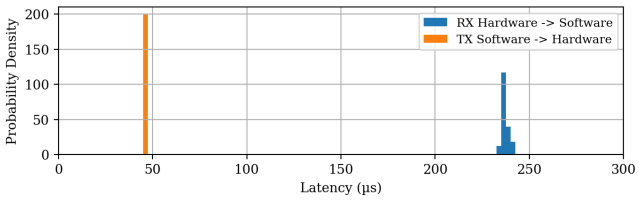

Fig. 14. Socket latencies (t_{TX}, t_{RX}) on a desktop PC without optimizations

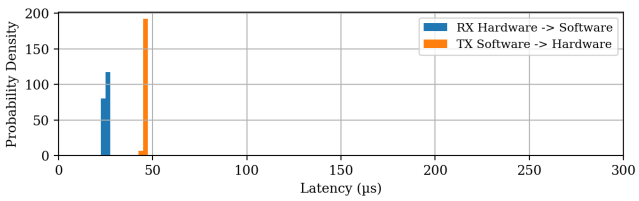

Fig. 15. Socket latencies (t_{TX}, t_{RX}) on a desktop PC with active busy-poll

slight reduction in asymmetry is only possible if client and server use this method. In order to neutralize the runtimes t_{RX} and t_{TX}, hardware timestamps at both the client and the server are therefore absolutely necessary. However, a one-sided use of the hardware timestamps on the server side leads to a greater asymmetry, especially with low-performance clients, since the socket runtimes no longer cancel each other out (see (18)).

VI. Conclusions

The result of the correction approaches presented here illustrates that an improvement of the synchronicity between client and server is quite possible. Under identical hardware and software conditions, systematic deviations of up to 85µs in local network context could be reduced to almost zero, more or less independent of the compensation method used. The introduced compensation method of direct timestamp correction on the server side or the use of an NTP extension field, both using values from a table of median values fitting the message properties leaves the server stateless giving it a vital advantage over the other approaches. The client easily can compensate its share by saving the socket timestamp when the packet is handed over to the driver buffer and using this value as sending timestamp in its offset calculation.

However, the measurements also confirmed the statement given in the analysis, that in the case of considerable performance differences between client and server, the directly non-measurable runtimes in the lower network layers dominate in a local network. So, perfect synchronicity is only achievable with hardware timestamps. Nevertheless, these must be supported and used by client and server in order to eliminate software latencies completely. Even in that case, one of the compensation methods is necessary with NTS secured NTP as the timestamps have to be packed into the NTP message before any cryptographic operation takes place.

In any constellation with or without hardware timestamping, any offset degradation caused by NTS securing can easily be compensated, thus annihilating any arguments against using NTS, today.

References

[1] D. L. Mills, U. Delaware, J. Martin, J. Burbank and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification," RFC 5905, doi 10.17487/rfc5905, 2010.

[2] D. Franke, D. Sibold, K. Teichel, M. Dansarie and R. Sundblad, "Network Time Security for the Network Time Protocol," Internet Draft, draft-ietf-ntp-using-nts-for-ntp-28, March 2020.

[3] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008), Nov 2019.

[4] D. L. Mills, "Network Time Protocol (NTP)," RFC 958, doi 10.17487/RFC0958, September 1985.

[5] D. L. Mills and B. Haberman, Ed., "Network Time Protocol Version 4: Autokey Specification," RFC 5906, doi 10.17487/rfc5906, June 2010.

[6] S. Röttger, "Analysis of the NTP Autokey Procedures," Project Thesis, Technische Universität Braunschweig, Institute of Theoretical Computer Science, Braunschweig, 2012.

[7] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, doi 10.17487/rfc8446, Aug. 2018.

[8] D. L. Mills, "Computer Network Time Synchronization - the Network Time Protocol," CRC Press, Boca Raton, 2006.

[9] D. Harkins, "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)," RFC 5297, doi 10.17487/rfc5297, 2008.

[10] D. F. Franke and A. Malhotra, "NTP Client Data Minimization," Internet Draft, draft-ietf-ntp-data-minimization-04, March 2019.

[11] M. Lichvar and A. Malhotra, "NTP Interleaved Modes," Internet Draft, draft-ietf-ntp-interleaved-modes-03, Feb 2020.

[12] D. F. Franke, "libaes_siv," GitLab Repository, [Online] available: https://github.com/dfoxfranke/libaes_siv, 2019.

[13] OpenSSL.org, "OpenSSL libraries v1.1.1b," [Online] available: https://www.openssl.org/source/old/1.1.1/, 2019.