Securing Unprotected NTP Implementations Using an NTS Daemon

Martin Langer, Thomas Behn and Rainer Bermbach

Full Citation of the original article published by IEEE:

M. Langer, T. Behn and R. Bermbach, " Securing Unprotected NTP Implementations Using an NTS Daemon," 2019 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), Portland, OR, USA, 2019, pp. 1-6.
doi: 10.1109/ISPCS.2019.8886645

Available at:
https://doi.org/10.1109/ISPCS.2019.8886645

Securing Unprotected NTP Implementations Using an NTS Daemon

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

Thomas Behn
Meinberg Funkuhren GmbH & Co. KG
Bad Pyrmont, Germany
thomas.behn@meinberg.de

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

***Abstract*—This paper presents a method to secure the time synchronization messages of various Network Time Protocol (NTP) services. It uses the Network Time Security protocol (NTS), which is now in a final, pre-RFC state, without the necessity of changes of their underlying implementations. A dedicated NTS service – the so-called NTS daemon (NTSd) – captures the standard NTP messages of the client and passes them on to an NTS server (tunneling). Supplied with the respective timestamps the secured message travels back via the NTS daemon to the NTP client, a procedure completely transparent to the NTP services. The presented research and the implementation of the method show advantages and limitations of the approach. Furthermore, it offers limited correction of NTS related time message asymmetries. Measurements provide an insight into the achievable accuracy and show the differences to NTP services with integrated NTS capability.**

Keywords—Network Time Security (NTS); Network Time Protocol (NTP); security; authentication; integrity; NTS daemon

I. Introduction

Time information is of paramount importance for computer systems and their network communication. It ensures accurate information exchange and interoperability. Typically, atomic clocks or GNSS receivers serve as a primary time source. Packet-based protocols like the Network Time Protocol (NTP) distribute the time messages using standard computer networks. Up to now, the dissemination of time information normally has taken place without any security measures, allowing precarious attacks. Existing security techniques for NTP are either unsecure (Autokey [7]) or impractical like the Symmetric Key [4] method as it needs upfront distribution of cryptographic keys via non-electronic media.

This situation motivated the development of a new kind of security measure for packet-based time distribution protocols. At the moment, the Network Time Security (NTS) protocol, mainly for securing NTP, is in a final development state and defined in an IETF draft in version 20 [1]. The release as an RFC is expected in 2019. A fully functional proof-of-concept (PoC) implementation is already available [15]. Final productive versions are still under development, because of the draft character of the specification.

Using NTS with an NTP service requires communication between the two. NTP requests have to be secured by NTS and responses need to be verified before standard NTP can use the time information. Servers have to check the incoming requests and to secure responses. In either case, a communication interface between NTP and NTS is necessary. As there is no generally defined interface up to now, integration of NTS into NTP is specific to the particular implementation. Therefore, every NTP/NTS developer designs his own version. Thus, retrofitting of NTS in existing NTP implementations surely will take time and the spreading of heavily needed securing of NTP by NTS will be hampered. In addition, special platforms run the risk of late or even no supply with NTS security.

This paper presents a solution to the problem described. The principle idea uses some kind of tunneling for NTP messages over a so-called NTS daemon (NTSd). The NTSd intercepts unsecured NTP messages, cryptographically secures the requests by use of the Network Time Security protocol and transfers them to another NTS daemon on the server side. That daemon checks the messages and passes them on to the standard NTP server. The server NTSd again catches the NTP response packages, secures and sends them to the client side. The client daemon checks the messages and transfers the trusted ones to the NTP client. The entire approach appears completely transparent to the NTP communication partners. This procedure enables arbitrary NTP implementations to use the NTS protocol without or with minimal changes to their software. In addition, this document analyzes possible asymmetries inferred by NTS and the NTSd and depicts a compensation method.

Chapter II presents basic information on NTP and NTS. Chapter III describes the tunneling concept, the communication structure and the compensation method in more detail. Chapter IV illustrates the measurement setup and actual measurements using a first implementation of an NTS daemon whereas chapter V discusses the measurement results compared to unsecured NTP and an integrated NTS/NTP implementation.

II. Preliminaries

This chapter gives a short review of both protocols, the Network Time Protocol (NTP) and its securing protocol Network Time Security (NTS). Some details can already be found e.g. in [2, 14]. More details on the current version of NTS [1] are presented in [6].

A. The Network Time Protocol (NTP)

Presented already in 1985 as RFC 958 by D. L. Mills [3] and revised in 2010 in version NTPv4 as RFC 5905 [4], the Network Time Protocol is a long standing and wide-spread protocol for distributing time information. NTP uses the connection-less UDP protocol via port 123. Its architecture operates with a hierarchically layered communication model. Receiving time information from stratum 0 sources (typically atomic clocks or GNSS receivers) stratum 1 servers distribute the time to layers below and so on. With every layer, the stratum number increases and the achievable accuracy decreases. Besides other communication models, the unicast mode (client to server, server to client) is the most prevalent modus operandi. In unicast communication, the client sends a request to the server at time T_1 (Fig. 1) and the server receives it at T_2. After processing, the server sends the response to the request at T_3 and the client receives it at T_4. As all four

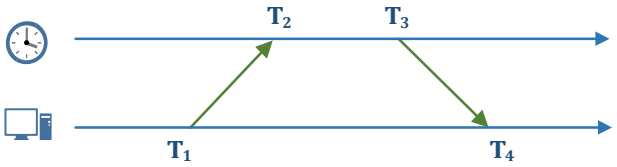

Fig. 1. Timestamps used in NTP unicast mode

timestamps are recorded the client computes the delay δ with (1) indicating the so-called round-trip time of the packet, and the time offset θ between client and server using (2). With equal propagation, delay of the packets from the client to server and vice versa the time offset quantifies the time difference between client and server. This aberration is used to control the local clock of the client. Variation of the propagation delay on the communication paths to and from the server causes an uncorrectable time offset between server and client.

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

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

B. *The Network Time Security Protocol (NTS)*

Application of NTP in Smart Grid communication in Germany led to the motivation to enhance NTP with security measures. As existing security features are unsecure or impractical, the design of the Network Time Security protocol started to solve this problem. It extends NTP communication and allows cryptographically secured time messages. NTS is still under development, but in a pre-final state [1] and the transition to RFC is expected in the near future. The NTS protocol underwent important design changes especially from [5] to the following draft versions. Previous work in [6] describes the differences and focuses on the achievable improvements in practical use.

As the unicast mode is the prevalent model, NTS concentrates on that mode. NTS in unicast mode proceeds in two consecutive phases (Fig. 2). The first step uses TLS 1.2/1.3 [8] to start the NTS Key Establishment (NTS-KE). The client sets up a TLS connection with the Key Establishment server that may be the same computer as the NTS secured NTP server or, for load balancing purposes, another dedicated server. Using that link the symmetric keys for client and server (C2S, S2C) are exchanged and a suitable AEAD (Authenticated Encryption with Associated Data, [9]) method is negotiated, defining the encryption algorithm. In addition, the NTS-KE server supplies the client with a set of cookies.

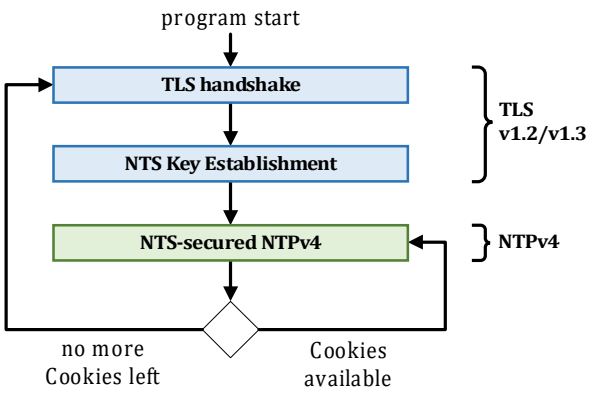

Fig. 2. Phases in NTS secured NTP communication

Amongst other things, the cookies contain encrypted information about the AEAD algorithm chosen and the generated keys. At each time request, the client resends such a cookie to the NTS server. The server derives the algorithm, the keys and other information from the cookies, which allows it to work in a stateless fashion.

After the NTS Key Establishment, the second phase exchanges secured NTP packets by incorporating the NTS content into NTP's extension fields. To this end, it defines special extension fields for the unique identifier, the cookies, the cookie placeholders and the NTS authenticator. The unique identifier prevents replay attacks. The cookies serve for identification of the client and enable the stateless operation of the server. The client transmits cookie placeholders to get new cookies, thus accomplishing request messages and response messages of equal size, which prevents amplification attacks. The NTS authenticator extension field carries the result of the AEAD algorithm applied to the NTP packet. The communication in the second phase may last until the server changes his internal crypto parameters, the client runs out of cookies (normally only when messages get lost) or its cookies are too old. Then the communication restarts with the NTS-KE phase.

The Ostfalia University of Applied Sciences presented first PoC implementations of NTS [15] as well as NTS integrated in an own NTP service [19] starting a first NTS server back in January 2018. Since then, Ostfalia created new implementations whenever different draft versions were released. Now three NTS servers with different draft versions are offered for tests with other implementers. In addition, Ostfalia defined a standard interface for its NTS library in C or C++, which can be used by other NTP implementations [15].

III. NTP Tunneling Concept

As already mentioned, the fast dissemination of the NTS standard would be desirable, but is hindered by necessary adaptations of the various NTP implementations and non-standardized APIs of NTS libraries. A special service, the NTS daemon, which tunnels NTP packets of any implementation and secures them with NTS, presents the solution and defuses the problem. In addition, NTP clients using multiple server connections can freely decide which connections should be secured and which not (Fig. 3). This chapter describes the structure of the NTS daemon and the communication process between NTP and NTSd that also provides an opportunity to compensate for NTS-related time lags.

A. *The NTS Daemon (NTSd)*

The basic idea is not to use NTS as an integral part of NTP or to embed it as a module, but to operate it separately as a service. The daemon communicates exclusively via the

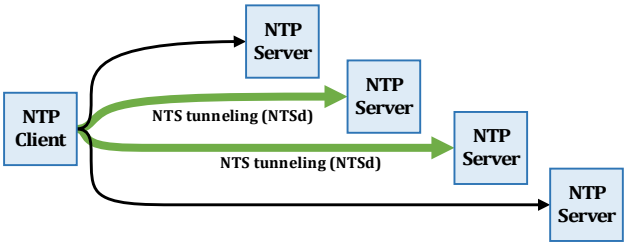

Fig. 3. The NTS daemon can be used for arbitrary NTP connections

network interface with the connected NTP or NTSd instances. This allows data exchange between NTP and NTSd without the need for changes to the NTP implementation. The daemon also manages connections to various time servers that provide an NTSd interface and controls the secure communication. This is done using the NTS embedded in NTSd (Fig. 4), that ensures the integrity of the NTP packets and the authenticity of the server.

The communication between NTP and NTSd takes place using a UDP connection. The client-side NTSd listens on the NTP standard port 123 of the localhost addresses to intercept outbound NTP request packets. A port change in the configuration of the NTP implementation is unnecessary which simplifies the application of the NTSd on existing systems. The simultaneous use of secured and unsecured NTP connections thus is possible, but not advisable. If the NTP client requires an unsecured connection to a time server, the public IP address of the server is specified as usual in the NTP configuration. However, to use a secure channel, an IPv4 address from the localhost address range (127.78.79.1 to 127.78.79.255) is used. The transmitted NTP packet is received via UDP port 123 from NTSd and processed accordingly. The assignment of the NTP packet to the secure time server is then carried out via an IP mapping within the NTS daemon. The simplest solution here is to use a configuration file for NTSd that manually assigns a secure time server to a specific localhost address. However, an idea for automated IP mapping could also be a NAT-like procedure within NTSd.

The main task of the daemon is to secure outgoing and verify incoming NTP packages secured by NTS. The integration of NTS is realized as a module that communicates via a defined programming interface. Similar to NTP, NTS also allows simultaneous communication with different time servers and manages individual states. The authentication of the time servers and the negotiation of cryptographic parameters are carried out via an NTS Key Establishment server. This can be either a separate actor in the communication chain or part of the server-side NTS daemon. If the NTS-KE server is separate, the IP must be known and set accordingly in the NTSd configuration. The transmission of information and parameters between NTSd and the NTS-KE server is independent of the NTP communication and occurs over a separate TLS channel. The client and server NTSd instances, however, communicate via a UDP channel. The port on which the server-side NTSd listens is not defined currently and thus freely configurable from the pool of available port numbers. The NTS-secured NTP packets are transmitted over this channel. If the NTSd receives such a packet, it forwards it to the NTP instance as standard NTP packet via a localhost connection after successful verification. However, corrupt or manipulated packets are discarded.

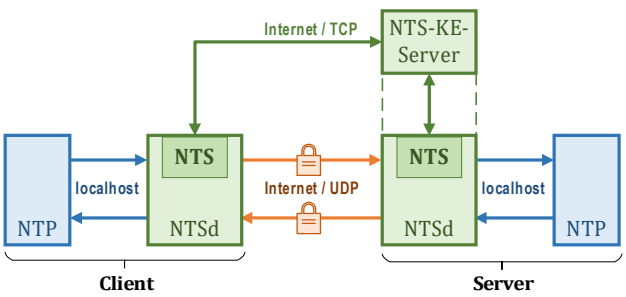

Fig. 4. NTS daemon concept overview

A precondition for the correct operation of the NTS daemon is the use of the unicast operating mode in NTP, since the support of other modes by NTS is not yet given. Some restrictions may also occur with the localhost communication between NTP and NTSd. For example, the NTP reference implementation NTPD [10] uses part of the localhost address range and locks the rest. To be able to use them again, a slight adaptation of the NTPD implementation is necessary. This may be possible with open source software, but it is not ideal. In this case, a container (for example *Docker* [11]) with an embedded NTS daemon offers a convenient alternative. The communication between NTP/NTSd then operates via a freely configured IP address. This procedure also solves possible port conflicts if, in addition to the NTP client, an NTP server application is located on the system and claims UDP port 123 for itself. The container solution allows the NTS daemon to communicate with the NTP instance even without a localhost connection. Furthermore, a later replacement of the NTP implementation with one having NTS natively integrated is easily possible and does not conflict with the NTSd. Of course then the tunneling through the daemon is no longer necessary.

B. *Communication Overview*

The NTSd-secured communication always starts on the client side by generation of an unsecured request packet in the NTP instance (Fig. 5). Normally, the NTP client writes the timestamp T_1 into this package and sends it. In addition, it temporarily stores the timestamp T_1 in order to be able to assign the subsequent response to the request. Due to the transmission of the NTP packet to a localhost address, the packet arrives at the client NTS daemon at the UDP port 123. There, the message is forwarded to the internal NTS module, which first checks whether the destination server has already been authenticated and the cryptographic information for securing the NTP packets for this time server is available. If not, a TLS connection to the NTS-KE server is automatically established in order to obtain this information. Afterwards the securing of the NTP package by NTS is possible. The protection of the NTP packet is achieved by means of NTP extension fields with integrity check of the packets. After returning of the NTS-secured packet from the internal NTS module to the client NTSd instance, it will be transmitted to the server-side NTSd over the UDP connection. There NTSd forwards the secured package directly to the NTS module, which performs the integrity check. Upon success, the server NTSd temporarily stores some NTS-related information of the request packet and removes the extension fields added by NTS. The daemon then forwards the standard NTP packet to the NTP server via localhost.

Upon reception of this packet, the NTP server immediately captures the *receive timestamp* T_2. Subsequently, the timestamp T_1 written by the client is shifted into the NTP header of the request packet and the timestamp T_2 is added. Now the server generates the *transmit timestamp* T_3 and

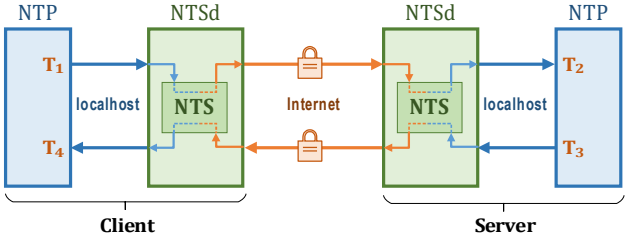

Fig. 5. NTSd-secured NTP communication

inserts it into the NTP packet. Immediately thereafter, this packet is sent back to the server NTSd via the localhost address. The daemon forwards the package directly to the NTS module and uses the pre-stored package information from the request to secure the package. Then, NTSd sends the protected NTP-packet back to the client-side NTSd. The previously stored packet information is discarded enabling a stateless mode of the server NTSd. Meanwhile, the client-side NTSd checks the secured NTP packet in the NTS module. If there is no integrity violation, the daemon removes the existing NTS extension fields from the NTP packet and forwards it as a standard NTP packet to the client NTP instance. In case of an error, NTSd discards the package. Immediately after receiving the NTP packet, the NTP client generates the *destination timestamp* T_4 and then compares the *origin timestamp* T_1 with the T_1 initially stored. If both match, the NTP client uses the packet for time synchronization.

C. Delay Compensation

NTP uses the timestamps T_1 to T_4 to determine the round-trip time of the packet and to calculate the time offset between NTP client and server by use of (1) and (2). However, the extension of the NTP communication path by the addition of NTSd causes an increase of the round-trip times. This in turn can lead to a reduction of the achievable synchronization accuracy. In particular, the securing process by NTS causes asymmetric runtimes and results in a systematic time offset. On the server side, protection of NTP packages always occurs after the timestamp T_3 has been written and the duration of the process varies depending on the performance of the hardware.

However, the additional time-synchronization jitter generated by NTSd can be mitigated by extending the function of the daemon. Therefore the client daemon generates four additional timestamps, which are recorded immediately when a packet is received or sent. These are stored for the duration of the current request of the NTP client and allow the segmentation of the round-trip time into their individual phases. The server-side NTSd instance also records timestamps and optionally can handle NTP server tasks. The function of an NTP server in unicast mode is essentially limited to the insertion of two timestamps into the NTP packet, whereat the operating system provides the time information via APIs. This functionality can easily be implemented in the server NTSd itself, which eliminates the need for a dedicated NTP server and the runtimes generated thereby. This leads to the principal configuration shown in Fig. 6. Regardless, the server can still use a separate NTP instance to ensure its own time synchronization.

The delay compensation uses the four timestamps T_1 to T_4 generated by NTP as a basis. The adjustment of these timestamps allows the compensation of the processing times in the NTS daemon, in order to reduce the jitter in time synchronization. Since the client NTP instance stores T_1 locally for later identification of the response, it is excluded from adaptation. In the same way, T_4 is generated in the NTP software after receiving the response and thus cannot be changed. Hence, the correction is limited to the adjustment of the two timestamps T_2 and T_3.

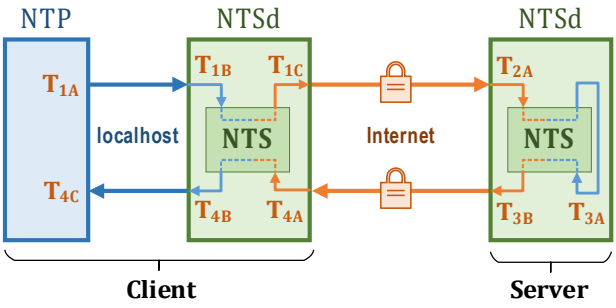

Fig. 6. Additional timestamps taken for the round-trip time (RTT) compensation; NTP server integrated into NTS daemon

T_2 is the server-side receive timestamp and is captured in this model (Fig. 6) by the server NTSd after receiving the request (T_{2A}). The pure packet transit time in the network can be determined by subtracting the process duration $T_{1C} - T_{1A}$. Since the server does not know these runtimes, this adjustment must be executed in the client daemon after receiving the response message. For this purpose, it temporarily stores the timestamps T_{1B} and T_{1C} until the corresponding reply message has been processed. The duration $T_{1C} - T_{1A}$ is composed of the localhost transmission time $T_{1B} - T_{1A}$ and the NTS processing time $T_{1C} - T_{1B}$. Both runtimes are variable and can be eliminated completely by temporary storage of the timestamps, regardless of which time is actually stored in the request packet.

T_3 is the server-side transmit timestamp of the NTP packet and is also set by the server NTSd. By adding the duration $T_{4C} - T_{4A}$ to this timestamp, a later transmission time of the server is simulated. As a result, the NTP client can again determine the pure packet transit time of the response message in the network. However, T_3 is not completely correctable in contrast to T_2. One difficulty is the server-side protection of the NTP packet by NTS at time T_{3A}, because the later dispatch time T_3 cannot be stored upfront in the NTP packet. An adjustment of the timestamp after securing is no longer possible, otherwise the integrity of the package would be violated and the client would discard it after the verification process. Although the processing time $T_{3B} - T_{3A}$ can be determined by NTS, it cannot be used directly. Adding an estimate value on T_{3A} to determine the probable transmission time T_{3B} provides one possible solution. This estimate could be e.g. a moving median of the processing time, allowing a rough correction. However, it should be noted that different cryptographic parameters of the NTP packets and server-side fluctuating workloads have an impact on the securing duration. Another problem exists on the client side when adding up the localhost runtime $T_{4C} - T_{4B}$. Since the timestamp T_{4C} cannot be determined by the client NTSd, an estimate also must be used here. Assuming that the internal localhost runtimes between NTP and NTSd vary only slightly for request and response packets, the duration $T_{1B} - T_{1A}$ can be used. Based on the model presented here, the following corrected timestamps in (3) to (6) now result:

$$T_1 = T_{1A} \tag{3}$$

$$T_2 = T_{2A} - (T_{1C} - T_{1A}) \tag{4}$$

$$T_3 = T_{3A} + \Delta t_{NTS} + (T_{4B} - T_{4A}) + \Delta t_{localhost} \tag{5}$$

with $\Delta t_{NTS} = T_{3B} - T_{3A}$ and $\Delta t_{localhost} = T_{1B} - T_{1A}$

$$T_4 = T_{4C} \tag{6}$$

However, still a problem may occur with the timestamp T_1. NTP implementations using the data minimization draft

[12] to improve the privacy protection transmit a random number (a so-called nonce) in the NTP packet instead of the timestamp T_1. Some implementations like NTPsec [13], Chrony [17] or OpenNTPD [18] apply this method. As a result, the duration $T_{1B} - T_{1A}$ cannot be determined. One solution is a localhost delay measurement of the client NTSd to itself. Here, once or periodically, an NTP packet is sent out by NTSd and addressed to itself. The round-trip time is measured and used as an approximation for $T_{1B} - T_{1A}$. A simple check of the received timestamp T_1 with the current time allows NTSd to determine if it is a random number or a timestamp.

It should be noted at this point that the use of filter mechanisms or the phase-locked loop (PLL) is determined exclusively by the NTP implementation used. NTSd does not use any other filter mechanisms in addition to the described timestamp correction.

IV. Measurement Setup And Configuration

Based on the NTS daemon described above, a PoC implementation has been developed to provide insight into the effects of the presented concept on the synchronization accuracy. This allows an initial assessment of the practicality and provides further information that is advantageous for optimizing the process. The daemon written in the programming language C uses an embedded NTS library developed in C++ to secure the NTP packages. Both software components are available as open-source software [15, 16]. The data exchange between NTSd and the embedded NTS takes place via an interface provided by NTS. The daemon implemented in the test setup is designed for Linux environments, but can also be used for other operating systems after some adaptations. The choice for the NTP itself fell on the reference implementation NTPD [10] due to its wide spread. This required small adjustments to ensure localhost communication between NTP and NTSd. These were limited to the adaption of three lines of code in NTPD to release a part of the localhost address range. After final configuration of the services by setting the ports and IP addresses, NTPD and NTSd were operational and ready for tests.

A practical implementation of the test setup was carried out with the support of Meinberg by providing suitable measuring equipment. A *LANTIME M500* (Fig. 7) forms the hardware basis for the client as well as for the server. These are equipped with a *Qseven* board carrying an *Intel Atom E3805* processor (2 x 1.33 GHz) as well as 2 GB RAM. These devices run *LANTIME OS v7* (LTOS), a Linux-based system on which NTPD and the developed NTSd are executable. During the measurement, client and server communicate over Ethernet via a *LANCOM GS-1224* switch with Gigabit connection.

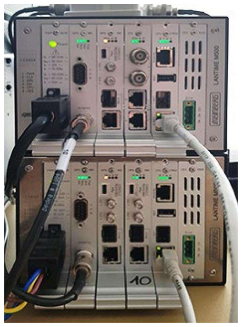

Fig. 7. Measurement setup with two LANTIME M500

Furthermore, both devices are equipped with a GPS receiver (*GPS180*) which are used to synchronize the system time. By contrast, the timestamps the client receives from the server are not used for synchronization, but to measure the time offset and delay, which NTPD continuously writes into statistics files. This enables recording and comparing measurement series with different configurations.

V. Measurement Results

With the setup described in Chapter IV, two measurement series were carried out to enable comparison between unsecured and secured NTP communication. For both series, the client performed several measurements simultaneously, which the NTP implementation recorded. One measurement series consists of three time sources (Fig. 8). The first source "*GENERIC*" serves as reference time, which is provided by the GPS180 module and has a high stability. The client synchronizes itself with it in order to be able to evaluate the delay and offset of the other channels. The second source (127.78.79.1) is an NTP connection tunneled through the NTS daemon to the connected time server. This can be seen from the localhost address that is specified here as the server. The third source (172.27.38.6) provides an unsecured NTP connection, as it is usual with NTPD and is indicated by a public IP. This one gets the time from the same time server, but does not use the NTS tunneling. Both measurement series in principle use the same sources, but differ in the NTS configuration. While the NTSd in the first measurement series exclusively tunneled the NTP packets, the compensation method described in Chapter III was used in the second measurement series. For all measurements, the console outputs of all implementations had been deactivated in order to avoid measurement distortion. Furthermore, both measurements use a fixed poll of 4 (message generation every 16 seconds) for a duration of 3 hours each.

The results show that NTS tunneling of NTP packets works well and that manipulated packets, unlike an unsecured NTP connection, are discarded. The comparison of the results reveals clear differences between the three measurements. Fig. 9 presents the round-trip time (delay δ) calculated by

```
Every 1.0s: ntpq -p                                    Wed Mar 20 13:02:47 2019

     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
oGENERIC(0)      .MRS.           0 l    6    8  377    0.000    0.000   0.001
 127.78.79.1     .MRS.           1 u    6   16  377    0.136    0.031   0.008
 172.27.38.6     .MRS.           1 u    3   16  377    0.109   -0.003   0.007
```

Fig. 8. Statistics output of NTPD. First line shows data of the synchronization source, whereas the second gives data of the NTSd secured communication. The last line presents data of a standard NTSd connection.

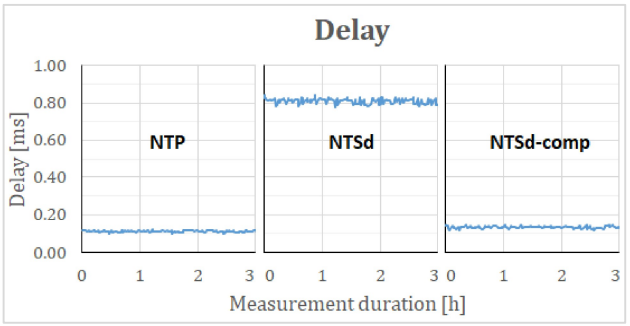

Fig. 9. Delay δ measured for standard NTP (left), normal NTSd tunneling (middle) and compensated NTSd commmunication. Sampling time: 16s

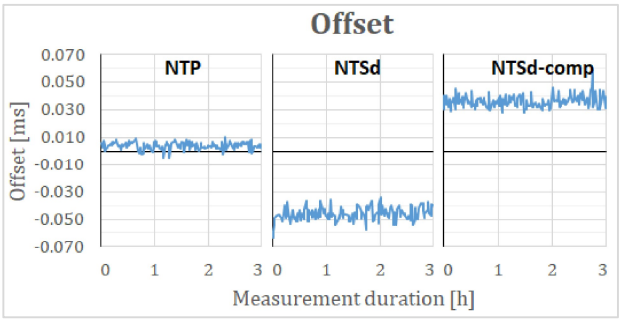

Fig. 10. Offset θ measured for standard NTP (left), normal NTSd tunneling (middle) and compensated NTSd commmunication

TABLE I. RESULTS OF THE MEASUREMENT SERIES

	Delay	Offset	Offset
	mean value[a]	*mean value*[a]	*std deviation*[a]
NTP	0.112 ms	3.991 µs	2.692 µs
NTSd	0.811 ms	-46.149 µs	4.857 µs
NTSd compensated	0.135 ms	35.683 µs	4.184 µs

[a] number of samples: 675

NTPD for the respective connections. Fig. 10 shows the time deviation (offset θ) of the time sources compared to the client synchronized with GPS and Table 1 indicates the statistical values of all connections. These results were calculated from the measured values obtained by NTPd.

Due to the short transmission distance of the hardware, the round-trip time of standard NTP is visibly short. The uncompensated NTSd tunneling, on the other hand, has a significantly higher value, since all runtimes caused by the daemon are included in this value. With activated latency compensation, NTSd is almost completely able to achieve the same value as with the unsecured NTP connection. On the other hand, the results for time offset are a bit surprising. The unsecured NTP shows only a very small deviation of 3.9 µs, while the uncompensated NTS tunneling causes a systematic offset of -46 µs. Due to uncorrected processing times and resulting asymmetries in the round-trip time, this result is as expected. However, an increased offset is also visible in NTS tunneling with active compensation.

Presumably, the kind of obtaining the timestamps from the system clock makes the difference. NTSd currently uses so-called userspace timestamps, which the operating system provides by high-level functions. NTPD, on the other hand, uses socket-timestamps, which are directly obtained from the Linux kernel and provide more accurate values. In addition, compared to a classic unsecured NTP connection, the client and server of an NTSd connection generate seven extra timestamps.

Further differences can be found in the send and receive functions, which can have variable latency times. Depending on the alignment and sum of the individual latencies, this leads to an asymmetry in the calculated round-trip time, which is reflected as a systematic offset. Though further investigation is necessary, it can be seen that the NTS daemon has no strong influence on the synchronization accuracy for typical NTP applications.

VI. CONCLUSIONS AND FURTHER WORK

We proposed a solution to enhance standard NTP services by NTS secured communication with no or rather minor modifications to the normal NTP implementation. The approach using an NTS daemon for securing and tunneling standard NTP packets proved to perform well. Achievable values of time offsets are more than sufficient for typical NTP applications. The loss of accuracy is minimal and can be minimized by further analysis and optimization, especially on the method how to obtain the timestamps. Intensive testing under realistic conditions is another point for future work to create better comparison possibilities.

REFERENCES

[1] D. Franke, D. Sibold, K. Teichel, M. Dansarie, R. Sundblad, "Network Time Security for the Network Time Protocol," Internet Draft, draft-ietf-ntp-using-nts-for-ntp-20 July 2019.

[2] M. Langer, K. Teichel, D. Sibold and R. Bermbach, "Time Synchronization Performance Using the Network Time Security Protocol," in 2018 European Frequency and Time Forum (EFTF), doi 10.1109/EFTF.2018.8409017, Turin, Italy, 2018.

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

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

[5] D. Sibold, S. Roettger, K. Teichel, "Using the Network Time Security Specification to Secure the Network Time Protocol," Internet Draft, draft-ietf-ntp-using-nts-for-ntp-06, Sep 2016.

[6] M. Langer, K. Teichel, D. Sibold, R. Bermbach, "Performance Comparison Between Network Time Security Protocol Drafts," IEEE International Frequency Control Symposium and European Frequency and Time Forum (IFCS-EFTF 2019), Orlando, USA, 2019, in press.

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

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

[9] D. McGrew, "An Interface and Algorithms for Authenticated Encryption," RFC 5116, doi 10.17487/RFC5116, Jan. 2008.

[10] Network Time Foundation, "NTPD," [Online] Available (03/26/2019): https://github.com/ntp-project/ntp

[11] https://www.docker.com, [Online] Available (03/26/2019)

[12] D. Franke, A. Malhotra, "NTP Client Data Minimization," Internet Draft, draft-ietf-ntp-data-minimization-04, March 2019 (Work in Progress).

[13] E. Raymond, G. Miller, M. Selsky, H. Murray, et al. "NTPsec," [Online] Available (03/26/2019): https://gitlab.com/NTPsec/ntpsec

[14] K. O'Donoghue, D. Sibold, S. Fries, "New security mechanisms for network time synchronization protocols," IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS 2017), Monterey, CA, 2017.

[15] M. Langer, "Network Time Security v0.9.0," [Online] Available (03/26/2019): https://gitlab.com/MLanger/nts/tags/v0.9.0

[16] T. Behn, "NTSd - Network Time Security Protocol Daemon," [Online] Available (03/26/2019): https://sourceforge.net/projects/ntsd/

[17] M. Lichvar, V. Blut C. Christianson, "Chrony," [Online] Available (03/26/2019): https://git.tuxfamily.org/chrony/chrony.git/

[18] B. Cook, et al., "OpenNTPD," [Online] Available (03/26/2019): https://github.com/openntpd-portable

[19] S. Häußler, C. Jütte, T. Kompa, S. König, T. Tuschik, M. Langer, "Network Time Protocol PoC v0.6.0," [Online] Available (03/26/2019): https://gitlab.com/MLanger/ntp/tags/v0.6.0