Algorithmic Approach to Estimate Variant Software Latencies for Latency-Sensitive Networking

Henning Puttnies, Björn Konieczek,
Jakob Heller, Dirk Timmermann
University of Rostock
Institute of Applied Microelectronics and Computer Engineering
18051 Rostock, Germany
Tel./Fax: +49 (381) 498-7277 / -1187251
Email: henning.puttnies@uni-rostock.de

Peter Danielis
ACCESS Linnaeus Center, School of Electrical Engineering
KTH Royal Institute of Technology, Stockholm, Sweden
Email: pdanieli@kth.se

***Abstract*—Recently, there has been the arise of latency-sensitive IoT applications, e.g., in the field of telemedicine or Industrial Internet. Such applications have stringent latency requirements. The entire end-to-end latency between two devices is composed of several individual latencies: the software latency of the application, the software latency in the networking stack, the hardware latency on wire, and the hardware latency in switches. The hardware latencies are relatively invariant (constant). Contrary, the software latency is highly variant and thus hard to determine. However, the software-introduced latency is crucial for latency-sensitive applications, since the end-to-end latency equals the sum of all latencies between two devices. In the literature, different investigations based on simulations as well as practical test-beds using specialized hardware are proposed. In contrast, we propose a novel method using no specialized hardware that precisely estimates the latency in the networking software (precision: approx. $144\mu s$). Our approach bases on the measurement of round-trip times between several devices and solving the resulting system of equations to estimate the latency of every individual device. Compared to the state-of-the-art the proposed method has also several advantages regarding scalability and resiliency.**

***Index Terms*—Internet of Things, Industrial Internet, Real-Time Systems, Software Latency, Network Latency**

I. Introduction

In future networks, there will be new application requirements like latency-awareness. This leads to the current interest in latency-sensitive networking. The digitalization of industrial facilities [1] as well as other latency-sensitive applications (e.g., in the field of telemedicine and driving assistance) contribute to this interest.

Partly, these applications have latency requirements, which have to be met in most cases but do not always have to be fulfilled (soft real-time requirements). Typical examples for soft real-time applications are video streaming and Voice over IP communication. Contrary, for several applications a deterministic latency must be guaranteed (hard real-time) like in industry automation scenarios [2].

To be able to guarantee hard real-time, it is essential to have knowledge about the worst case end-to-end latency between the devices. The latency between two devices has variant and invariant (constant) components. Figure 1 depicts a schematic overview of the communication between two devices and the introduced latencies. The constant components arise from the hardware latency inside switches and on wires. The hardware latency can be easily determined from the specification of the switches, the length of the wires, and the maximal transmission data rates.

978-1-5090-0996-1/16/$31.00 2016 IEEE

Contrary, the software latency imposed by the processing of a packet in the software stack is harder to determine. Many parameters such as the operating system, the workload of the system, and the executed application influence the software latency. However, the software latency is important for latency-sensitive applications, as the software latency has to be known to determine the worst case latency. Hence, the determination of the software latency is essential for enabling latency-sensitive applications.

In the literature, different simulation evaluations as well as practical test-beds using specialized hardware are proposed. In order to measure the latency between two devices, most approaches use the round-trip-time (RTT). However, using one RTT it is only possible to determine the sum $t_a + t_b$, where t_a is the software latency of device A and t_b is the software latency of device B.

Instead, we propose a conceptually different approach that estimates the software latency as a probability distribution measuring multiple RTTs. Therefore, we can precisely state the mean latency as well as the minimum latency and maximum latency. Moreover, we can state the statistical accuracy of the estimated probability distribution and the accuracy of the estimated worst cast delay. Furthermore, we are able to determine the software latency of a particular device (t_a,t_b) and not only the sum of the latency of two devices ($t_a + t_b$) .

The contributions of the proposed approach are as follows:

1) As our algorithmic approach is software-based there is no need for specialized hardware. Therefore, we are able to estimate the latencies of several devices in a network in situ (in the actual application area) at any time without preparation.
2) It is robust as there is no single point of failure and

hence also suited for industrial scenarios.

3) It is scalable as we need n estimations for n different devices having unknown latencies. Furthermore, we need to estimate every latency distribution only once if the latency distribution is time-invariant.
4) It is precise as we estimate the latency of every device separately. This is easier and more accurate than the estimation of the sum of two latency distributions. In addition to the distribution of the software latency, we can state the worst case latency as well as the statistical accuracy.

The remainder of this paper is organized as follows: Section II explains the concept of the proposed method and considerations concerning scalbility, resiliency and uncertainty. In Section IV, we describe and evaluate the experimental results. Section V gives an overview of the related work and compares our approach to the literature. Finally, Section VI concludes this paper.

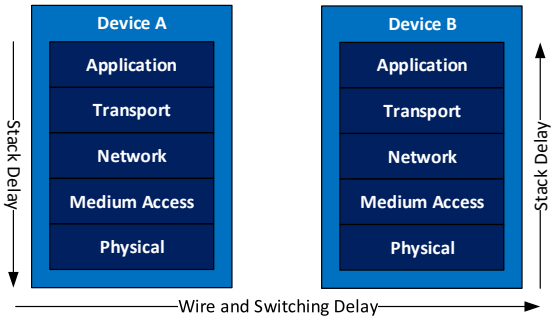

Fig. 1. latency between two devices in a network

II. Concept: Estimation of variant Software Latencies

In this chapter we will present our approach. First, we will give the mathematical background of the algorithm and afterwards explain the procedure of the algorithm.

A. Algorithm of Latency Estimation

Firstly, we want to determine the latency t_a of the device A and the latency t_b of device B (Figure 2). As there are two devices in the network, we can measure one RTT between A and B. As a result, we get the following equation:

$$RTT = 2 \cdot t_a + 2 \cdot t_h + 2 \cdot t_b \tag{1}$$

t_h denotes the hardware latency of wires and switches connecting the devices A and B. As we have one equation and two unknown parameters t_a and t_b, the system of equations is underconstrained and it is mathematically impossible to determine t_a and t_b. We have to state that it is possible to measure the RTT between A and B multiple times. However, if we assume that t_a and t_b are constant the resulting system of equations will always have $rank$ 1. If we assume t_a and t_b to be variant we can estimate the sum of both. Nevertheless, it is impossible to estimate t_a and t_b separately as multiple measurements of the same RTT do not lead to independent equations.

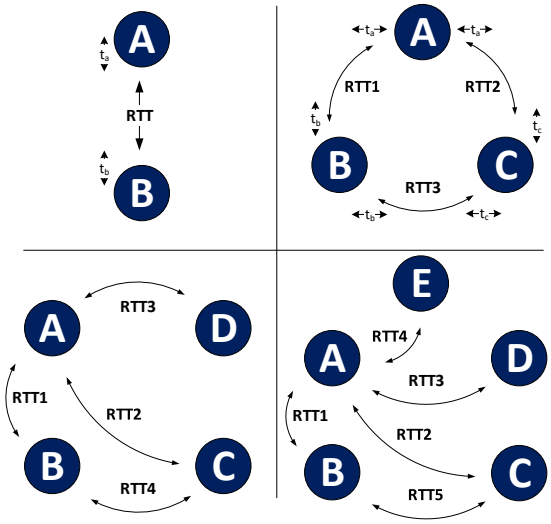

Fig. 2. The RTTs between three or more devices must be estimated to obtain a solvable system of equations.

Secondly, we assume that we want to determine the latencies t_a, t_b, and t_c of three devices: A, B, and C. As there are three devices in the network, we can measure three RTTs between A, B, and C. As a result, we obtain the following system of equations:

$$RTT_1 = 2 \cdot t_a + 2 \cdot t_{h1} + 2 \cdot t_b \tag{2}$$

$$RTT_2 = 2 \cdot t_a + 2 \cdot t_{h2} + 2 \cdot t_c \tag{3}$$

$$RTT_3 = 2 \cdot t_b + 2 \cdot t_{h3} + 2 \cdot t_c \tag{4}$$

Again, t_{hx} is the hardware latency of wires and switches between these devices. As we have three equations and three unknown parameters, the system of equations is solvable. We can also write this system of equations in matrix form:

$$\begin{bmatrix} RTT_1 \\ RTT_2 \\ RTT_3 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 \\ 2 & 0 & 2 \\ 0 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} t_a \\ t_b \\ t_c \end{bmatrix} + 2 \cdot \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}. \tag{5}$$

Finally, we assume that we want to determine the latencies of n devices in a network. As there are n devices in the network, we can measure n independent RTTs. As a result, we get the following system of equations:

$$\begin{bmatrix} RTT_0 \\ RTT_1 \\ \vdots \\ RTT_n \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & \dots & 0 \\ 2 & 0 & 2 & & \vdots \\ \vdots & \vdots & & \ddots & 0 \\ 2 & 0 & \dots & 0 & 2 \\ 0 & 2 & 2 & 0 & \dots \end{bmatrix} \cdot \begin{bmatrix} t_a \\ t_b \\ \vdots \\ t_n \end{bmatrix} + 2 \cdot \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_n \end{bmatrix}. \tag{6}$$

Again, t_{hx} is the hardware latency of wires and switches between the devices. As we have n equations and n unknown parameters, the system of equations is still solvable.

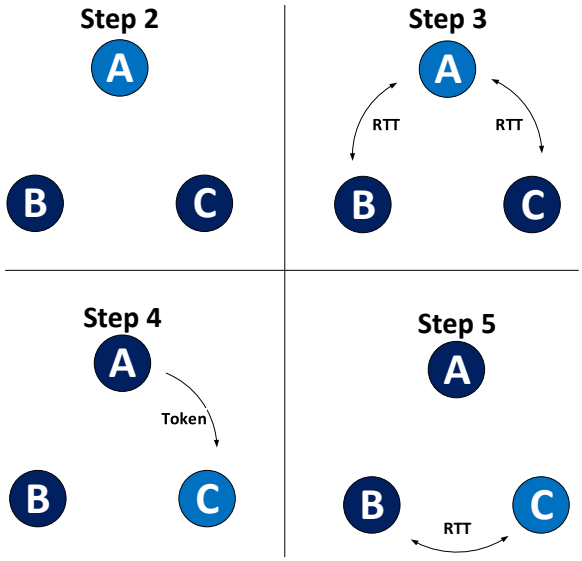

Fig. 3. Steps 2 to 5 of the proposed concept

B. Procedure of Latency Estimation

The basic idea of our approach is the estimation of the RTT as distribution as well as the estimation of several RTTs between different devices, which results into a solvable system of equations. The procedure executes several steps that are partly illustrated in Figure 3:

1) The hardware latency of the wires and switches have to be determined. We can calculate the latency from the network topology and the specifications of the network infrastructure devices (switches, wires) or introduce measurements as presented in [3].
2) An inception device (A) is selected. One way to select the inception device is introducing a sleep time of a random duration initially. The device, which awakes first, is the inception device. A second way to select the inception device is to use a best master clock algorithm similar to PTP. This algorithm determines the device whose clock has the highest precision.
3) The inception device (A) estimates the RTTs to all devices of interest (B, C) as a distribution. We denote every device whose software latency shell be estimated as device of interest.
4) The inception device (A) sends the token to one other device (e.g., B).
5) The token receiver (B) estimates yet another RTT to one arbitrary device of interest (e.g., C) as distribution, and sends this distribution back to the inception device (A).
6) The inception device (A) uses the estimated RTTs of all devices of interest, the previously calculated hardware latency, and the presented system of equations to calculate the average as well as the maximum and the minimum of every latency distribution for every device of interest.

To solve the system of equations, it is assumed that the transmission latency of a device equals the receive latency of this device. Without this simplification, it is impossible to determine all unknown variables of the latency according to [4]. However, this simplification solely introduces a minor conceptual error that can be precisely stated.

C. Consideration of the Conceptual Error

We assume the receive software latency and the transmission software latency to be equal. This leads to a conceptual error. However, we can determine the maximal conceptual error:

$$Error_{max} = (latency_t + latency_r)/2. \tag{7}$$

Thereby, $latency_t$ is the transmission software latency and $latency_r$ is the receive software latency. Furthermore, we can quantify an upper bound for the maximum (worst case) latency $latency_{worst}$ of a device. The transmission latency and the receive latency have the same upper bound:

$$latency_{worst} \leq (latency_t + latency_r). \tag{8}$$

In addition to this conceptual error, which results from our assumptions, there are several errors introduced by measurement uncertainties. These errors depend on the implementation as well as the measurement setup and will be considered later.

D. Conceptual Consideration of Resiliency

As a dynamic selection of the inception device is possible, our approach does not have a single point of failure. The dynamic selection of the inception device can be done by waiting for a random time. Alternatively, a best master clock algorithm can be utilized similar to PTP. Furthermore, there are no special requirements to hardware or software of the inception device.

E. Conceptual Consideration of Scalability

As we focus on latency-sensitive IoT scenarios of the future, scalability is an significant issue. Prospectively, there will be networks of thousands of connected devices. Concerning scalability, our approach exhibits advantages compared to the state-of-the-art. We define the execution time T that is required for the estimation of the software latencies of n devices. Therefore, in a network of n devices, we can formulate the following equation:

$$T \sim n. \tag{9}$$

As we always need n independent measurements for a system of equations of $rank\ n$ composed of n independent equations, we have to measure n RTTs. Consequently, we need to measure one RTT for every unknown parameter. Nevertheless, our approach allows a reduction of the number of unknown parameters. The number of unknown parameters is not equal to n (the number of devices in the network) but it is equal to the number of differing devices in the network. A device D_x

is defined as a pair of a hardware setup HW_y and a software setup SW_z:

$$D_x = (HW_y, SW_z). \tag{10}$$

Moreover, the hardware setup is defined as the set of all hardware elements of this device and the software setup is defined as the set of all software elements of this device.

$$HW_y = \{Processor, NIC, Memory, ...\} \tag{11}$$

$$SW_z = \{OS, Application, Drivers, ...\} \tag{12}$$

If two devices D_{x1} and D_{x2} have a different hardware setup and/or hardware setup they may introduce a different latency distribution. We denote D_{x1} and D_{x2} as differing devices. Instead, if D_{x1} and D_{x2} have the same software setup and the same hardware setup they introduce the same latency distribution. Consequently, it is sufficient to estimate the latency distribution of D_{x1} or D_{x2}. Even in IoT scenarios connecting thousands of devices, it is assumable that the number of differing devices is greatly lower compared the total number of devices. Many devices of the same model from a particular manufacturer executing identical tasks (e.g., sensors, actors, controllers) will have the same software setup and hardware setup. Therefore, they introduce the same software latency. Furthermore, if we assume that software setup and hardware setup are relatively time-invariant only one estimation for every unique pair $D_x = (HW_y, SW_z)$ is needed.

III. Implementation and Experimental Setup

To evaluate our approach, we developed a prototype implementation. A simulation would not be useful as the first step as many parameters contribute to the latency in a complex way. As these parameters are unknown we have to determine them in a real test-bed. We implemented our approach in Java to achieve high platform independence. Java is not widely used in latency-sensitive and real-time scenarios. Nonetheless, the authors of [5] show that Java implementations can even meet hard real-time requirements. Therefore, we used the Jamaica VM [6], which is a real-time-capable Java Virtual Machine. Furthermore, our approach estimates the latency as distribution and thus we do not assume the software latency to be constant. Our implementation is flexible and generic. We can change the number of examined devices, the size of the packets as well as the number of packets to estimate every RTT. Especially, the size and the number of packets are important parameters. We assume that the software latency as processing latency is dependent on the packet size. Moreover, the number of packets to estimate each RTT is important to ensure a sufficient statistical accuracy of the distribution.

As experimental setup, we used three Intel Galileo Boards [7] and a 1 GBit/s switch to connect these devices.

IV. Measurements and Evaluation

First, we conducted a measurement, which is independent of our implementation (see Figure 4). Afterwards, we can use this independent measurement for comparison with our implementation. We used the ping tool to measure the RTT between two PCs. The difference between our implementation and the ping measurements is that ping uses the ICMP protocol and thus there is no overhead introduced by UDP. Furthermore, as we executed ping natively there is no overhead introduced by the Java Virtual Machine in this measurement. The mean of the RTT varies between 3 ms and 4 ms. Furthermore, there are many outliers.

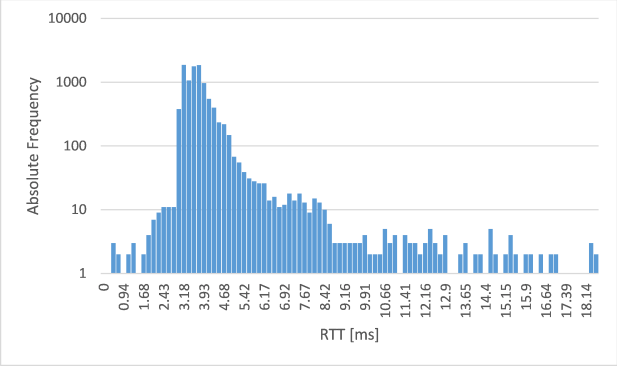

Fig. 4. Histogram of the RTT between 2 PCs: 10,000 packets (PCs: Windows7/i7-2670QM and Archlinux/i3-6100U)

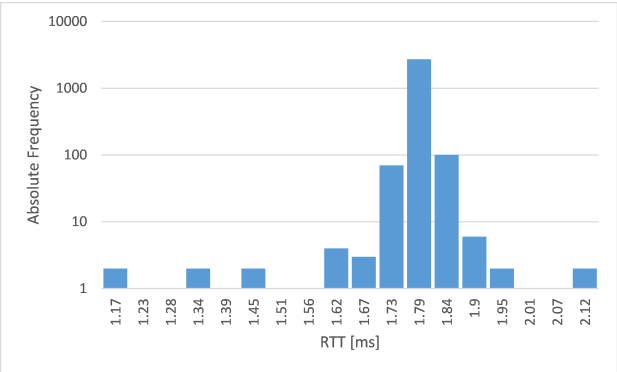

Fig. 5. Histogram of the RTT between 2 Galileo Boards: 3,000 packets

Secondly, we used ping to measure the RTT between two Galileo boards (see Figure 5). The mean of this RTT distribution lays between 1.5 ms and 2 ms, the RTT is normally distributed, and there are no outliers. In this measurement, we observed the RTT between two platforms running a real-time OS. Thus, we can assume a predictable execution time and hence measured no outliers. From the first two measured distributions, we can conclude that the RTT as well as the software latency is a distribution and thus should not be assumed to be constant. We have to state that even on a real-time platform where the software latency has a predictable maximum value the latency is still variant and not constant.

Thirdly, we measured the RTT using our Java implementation (see Figure 6) using the Oracle JVM. We see that the distribution has a mean value of 2.30 ms. This additional latency compared to the ping measurement was introduced by the overhead of UDP as well the processing overhead of the JVM. Again, we observed a few outliers.

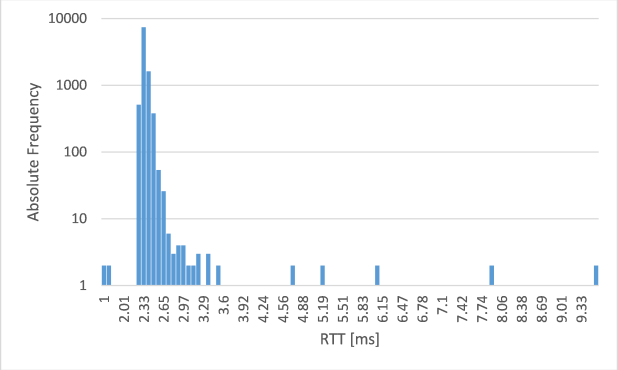

Fig. 6. Histogram of the RTT between 2 Galileo Boards measured with our Java Implementation using the Oracle JVM (10,000 packets; min = 1.69ms; mean = 2.30ms; max = 9.56ms)

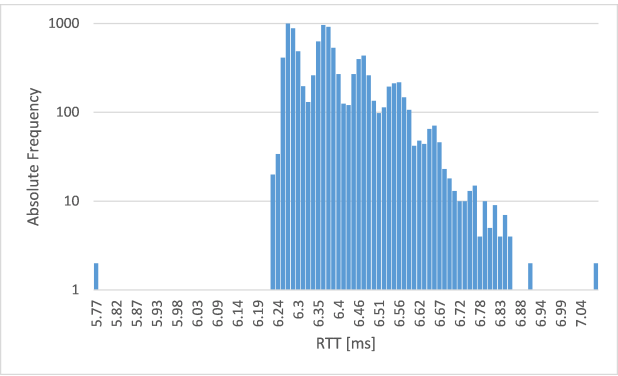

Fig. 7. Histogram of the RTT between 2 Galileo Boards measured with our Java Implementation using the Jamaica VM (10,000 packets; min = 5.76ms; mean = 6.38ms; max = 7.08ms)

Additionally, we investigate our Java prototype implementation using the Jamaica real-time JVM and the corresponding Java real-time threads (see Figure 7). We see that the distribution has a mean value of 6.38 m. The additional latency compared to the Oracle JVM is introduced by the large processing overhead of the real-time JVM. Nevertheless, the behavior of the implementation is much more predictable. All observed RTTs lay within a range of 1.38 ms whereas the RTTs exhibit a range of 7.87 ms using the Oracle JVM.

Next, we tested whether our implementation is able to determine software latencies. First, we estimate the latencies of all devices as described in Section II and stated the latency in ns:

$$Latency_{normal} = \begin{bmatrix} 1599130 \\ 1585960 \\ 1604775 \end{bmatrix}. \quad (13)$$

In a second measurement, we introduced an additional latency of 2 ms to the sending and receiving latency of one device. Finally, we executed the complete estimation again. As the result, we got the vector of the software latencies in ns:

$$Latency_{delayed} = \begin{bmatrix} 1602646 \\ 1580520 \\ 3724501 \end{bmatrix}. \quad (14)$$

We can see that the implementation can detect the introduced latency. Thus, we assume the measured latency of the modified device to be 2 ms higher than the latency of the unmodified devices. Actually, the latency of the modified device is 2.144 ms higher than the latency of the unmodified device, which amounts to 1.605 ms. Therefore, we can conclude that our approach can state the software latency with an precision of approx. $144\mu s$.

Concerning the error, we have not subtracted the hardware latency introduced by wires and the switch. However, the focus of our prototype implementation was the functional validation of our concept. Furthermore, Schweissguth et al. proposed an easy method to measure the latency of a switch in [3], where the author state that the hardware latency of layer 2 switches is approx. $30\mu s$. The half of this hardware latency is added to every measured latency value. Therefore, we can state the error considering the mean of the measured software latency of 1.595 ms:

$$\frac{15\mu s}{1.595ms} \approx 0.0094. \quad (15)$$

Thus, the error introduced by not considering the hardware latency is approximately 1 %.

V. State of the Art and Related Work

NTP and PTP are the established standard synchronization protocols. Both use the RTT between two devices to determine the software latency. Nevertheless, only the sum $t_a + t_b$ can be determined in this way, where t_a is the software latency of device A and t_b is the software latency of device B.

The standard for the conduction of latency measurements is defined in RFC 2544 [8]. The authors state that measurements should be carried out using a device under test and a tester. The latency l is defined as $l = t_t - t_r$ where the tester takes the timestamp t_t after the data is fully transmitted and the timestamp t_r after the data was fully received again. Basically, this latency definition equals the RTT and thus it is impossible to distinguish between hardware latency and software latency as well as between the latency t_a introduced by the tester and the latency t_b introduced by the unit under test. Furthermore, they utilized a specific testing device.

In [9], the authors present an NS3-based simulation as well as practical measurements of the software latency. Especially, the network interface controller (NIC) and the New API (NAPI) are examined. The measurements are done using a specialized load generator and the device under test.

Emmerich et al. examine the network latency of game servers in [10]. They state that the main part of the latency is introduced by the communication over the Internet and that buffer bloat can lead to latencies in this scenario. They consider the latencies of the network stack and propose user space stacks to optimize latency and throughput. The authors

conducted the measurements using a 10 GBit/s load generator and a game server. Moreover, they used PTP timestamps of incoming and outgoing packets.

Rotsos et al. present a framework for the evaluation of switching hardware in [11]. They focus on latency measurements of OpenFlow-enabled switches for software-defined networks. They present a generic software framework but use dedicated FPGA-based hardware implementations for the measurements rather than a measurement based on standard hardware.

In [12], Inoue et al. examine software latencies and present a hardware/software implementation of a low-latency network stack. They state that context switches and buffer copies introduce latency and co-processing can reduce this latency. Furthermore, they state that multi-core processing and parallelization has a positive impact on the throughput but a negative impact on the latency. This confirms that many parameters contribute to the the software latency. Thus, it can not be assumed to be constant. Their system achieves high performance through direct access to the hardware from the user space as well as specialized software. Hence, their approach is very platform dependent.

In [13], the end-to-end latency in TCP/IP networks is analyzed. The authors state interrupt management, head-of-queue effects, system resource contention, context switches, high processor load, and high traffic load as sources of latency variations. For latency-sensitive scenarios the authors propose core affinity as well as an adaptive change between interrupts form the NIC and active polling by the kernel of the operating system. Moreover, they propose adaptive interrupts: for low-latency requirements the interrupt shall be triggered instantly. Contrary, there shall be one interrupt for many packets to achieve a high throughput. This confirms that the software latency is highly depended on the application and implementation.

The authors in [14] present an analysis of the current Linux network stack. Especially, they focus on investigating latency and throughput. They state that the NAPI offers a trade-off between latency and throughput. Nevertheless, a joint optimization towards low latency and high throughput is not supported. However, low-latency polling can be used, e.g. utilizing the Intel ixgbe-Driver, which leads to a latency improvement of 30% according to Intel.

Compared to the state-of-the-art, our proposed method has several enhancements:

- It provides the possibility to distinguish between the latencies t_a and t_b. Therefore the software latency of a single device can be estimated, which is impossible if we use only one RTT.
- The latency is not assumed to be constant. Instead, a probability distribution is measured. As a result, we can state statistical parameters like mean, standard deviation and statistical accuracy.
- Furthermore, we can investigate the worst case latency and thus evaluate the real-time capability of a device.
- Our algorithmic approach introduces a software-based estimation. Thus, it is platform independent and no specialized hardware is needed. Moreover, measurements are possible in the real field of application.
- The proposed approach offers a good scalability: if time-invariance of the latency distribution can be assumed, we only need one estimation for every device. We note that if time-invariance cannot be assumed, a determination of the latency is impossible.

VI. Conclusion and Future Work

In this paper, we present a novel method to estimate the latency of networking software. The proposed method is platform independent and hence, there is no need for specialized hardware. Therefore, the proposed method can be used to determine the software latency for a clock synchronization algorithm or to automatically evaluate the real-time capability of a device in its real field of application.

Furthermore, the approach has a good scalability. In a network with n nodes having m different platforms (platform = hardware + software) the formula $T \sim m$ applies where T is the time needed to determine all software latencies. We need only m measurements to determine the m unknown latencies, which is the mathematical minimum. We always need m different measurements to get m independent equations, which compose a system of equations of $rank\ m$.

Prospectively, we will examine the utilization of our approach for a clock synchronization algorithm.

Acknowledgment

The authors would like to thank the German Research Foundation (DFG) (research fellowship, GZ: DA 1687/2-1) for their financial support.

References

[1] P. C. Evans and M. Annunziata, "Industrial internet: Pushing the boundaries of minds and machines," *General Electric. November*, vol. 26, 2012.

[2] M. Felser, "Real time ethernet: Standardization and implementations," in *2010 IEEE International Symposium on Industrial Electronics (ISIE 2010)*, pp. 3766–3771.

[3] E. Schweissguth, P. Danielis, C. Niemann, and D. Timmermann, "Application-aware industrial ethernet based on an sdn-supported tdma approach," Aveiro, Portugal.

[4] N. M. Freris, S. R. Graham, and P. R. Kumar, "Fundamental limits on synchronizing clocks over networks," *IEEE Transactions on Automatic Control*, vol. 56, no. 6, pp. 1352–1364, 2011.

[5] B. Konieczek, M. Rethfeldt, F. Golatowski, and D. Timmermann, "Real-time communication for the internet of things using jcoap," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 134–141.

[6] "Jamaica virtual machine." [Online]. Available: http://www.aicas.com/cms/en/JamaicaVM

[7] "Intel galileo board." [Online]. Available: https://www.arduino.cc/en/ArduinoCertified/IntelGalileo

[8] S. Bradner and J. McQuaid, "Benchmarking methodology for network interconnect devices."

[9] A. Beifus, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle, "A study of networking software induced latency," in *2015 International Conference and Workshops on Networked Systems (NetSys)*, pp. 1–8.

[10] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "A study of network stack latency for game servers," in *2014 13th Annual Workshop on Network and Systems Support for Games (NetGames)*, pp. 1–6.

[11] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation," in *Passive and Active Measurement*, ser. Lecture Notes in Computer Science, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, N. Taft, and F. Ricciato, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7192, pp. 85–95.

[12] K. Inoue, D. Pasetto, K. Lynch, M. Meneghin, K. Muller, and J. Sheehan, "Low-latency and high bandwidth tcp/ip protocol processing through an integrated hw/sw approach," in *IEEE INFOCOM 2013 - IEEE Conference on Computer Communications*, pp. 2967–2975.

[13] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni, "Architectural breakdown of end-to-end latency in a tcp/ip network," *International Journal of Parallel Programming*, vol. 37, no. 6, pp. 556–571, 2009.

[14] L. M. Märdian, "What's new in the linux network stack?" *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, vol. 2, p. 6, 2014.