A Deep Learning Approach to Dynamic Passive RTT Prediction Model for TCP

Desta Haileselassie Hagos*, Paal E. Engelstad†, Anis Yazidi‡, Carsten Griwodz§
*†University of Oslo, Department of Technology Systems, Kjeller, Norway
*†‡Oslo Metropolitan University, Department of Computer Science, Oslo, Norway
§University of Oslo, Department of Informatics, Oslo, Norway
Email: *destahh@ifi.uio.no, {*desta.hagos, †paal.engelstad, ‡anis.yazidi}@oslomet.no, §griff@ifi.uio.no

***Abstract*—The Round-Trip Time (RTT) is a property of the path between a sender and a receiver communicating with Transmission Control Protocol (TCP) over an IP network and over the public Internet. The end-to-end RTT value influences significantly the dynamics and performance of TCP, which is by far the most used communication protocol. Thus, in communication networks, RTT is an important network performance variable. By measuring the traffic at an intermediate node, a network operator or service provider can estimate the RTT and use the estimation to study and troubleshoot the per-connection characteristics and performance. This paper aims at improving the accuracy and timeliness of the RTT estimation, to help network operators improving their analysis. We propose and evaluate a novel deep learning-based model capable of dynamically predicting at real-time the RTT between the sender and receiver with high accuracy based on passive measurements collected at an intermediate node, taking advantage of the commonly used TCP *timestamps*. We validate extensively our prediction methodology in a controlled *experimental testbed* and in a *realistic* scenario on the *Google Cloud platform*. We show that our model, which is based on classical deep learning algorithms, gives reasonably effective *state-of-the-art* performance results across multiple TCP congestion control variants. We also show that the model works well for transfer learning. Even though the RTT prediction model was trained on an emulated network, it performs well also when applied to a realistic scenario setting, as demonstrated in our experimental evaluation.**

***Keywords*—TCP, RTT Prediction, LSTM, Passive Measurements**

I. Introduction and Motivation

Passive measurement techniques of TCP flows have gained much attention in the networking research community lately [3, 7, 9, 25]. The main reason is that such measurements are becoming increasingly useful for network operators and Internet Service Providers (ISPs) to evaluate the communication performance of applications and services running on their network. Monitoring the traffic at an intermediate node, allows the ISP to assess the underlying network performance, which is crucial for their operation. The RTT is one of the most important indicators of communication performance. The RTT is a TCP state variable that influences congestion control in many TCP variants, and the RTT has a huge influence on the performance of the end-to-end communication. In order for network access providers to determine and diagnose application performance issues on the public Internet, knowledge about the characteristics of the network is a very important factor. The ability to passively compute and dynamically predict the RTT is very crucial for a lot of reasons. For example, it allows network operators to measure and optimize the network performance of real-time applications and services, and it helps providers understand the responsiveness, availability of their network services, performance and predict the behavior of a TCP connection. Network operators and service providers care about RTT, and they even make RTT a Service Level Agreement (SLA) parameter in legal contracts with their customers [27]. Customers who demand better services can *passively* detect the occurrence of SLA violations and this ability would allow the network providers to quickly respond to Quality of Service (QoS) problems [27]. This is particularly important for the quality of *latency-sensitive* and *bandwidth-intensive* real-time media applications (such as video, audio, and application sharing), etc. [27]. RTT is the length of time it takes for an outgoing TCP client packet plus the minimal time spent for an acknowledgment of that segment from the server to be received by the client [23]. The RTT between the sending and receiving endpoints is typically a combination of a fixed *BaseRTT*, a fixed *propagation time*, and the amount of queueing that is experienced along the path. Thus, the changes in the RTT might give an indication of changes in queuing and the congestion in the network, and be a useful input to the TCP congestion control algorithm.

TCP is a highly reliable connection-oriented transport protocol capable of adding reliability and preventing excessive congestion on the Internet [17]. Note that congestion control in TCP was not part of the protocol initially until the first Internet congestion collapse was observed [16]. TCP controls congestion by also aiming for fair sharing of the available network resources by the competing flows, using strategies empowered by TCP [17]. TCP fairness means that if N TCP sessions share the same bottleneck link of a bandwidth B, each session should ideally get an average rate of $\frac{B}{N}$ and $\frac{1}{N}$ of the available link capacity assuming that all the active TCP connections have the same increase of rates and similar RTTs. If the multiple TCP sessions have different RTTs but share the same bottleneck link, the flows with larger RTTs usually achieve lower throughput, while the flows having smaller RTT may utilize the bandwidth more aggressively than the others [13]. Indeed, the RTT directly influences the TCP throughput according to the following equation:

$$T_i \propto \frac{1}{\sqrt{p_i}.RTT_i} \tag{1}$$

where T_i is the throughput, p_i is the probability of a packet loss rate, and RTT_i is RTT of a TCP flow i. Equation 1 shows that the throughput ratio of individual TCP connections is inversely proportional to the RTT [29]. This means that RTT is one of the most important state variables that determine the aggressiveness of a TCP flow. This also means that passively predicting RTT is useful for the deployed TCP variants to optimize for high bandwidth by leveraging the TCP *timestamps* option carried in each TCP header. Evaluating the

RTT, inflated by queueing across the network [29], may also give a more detailed view of the sender state than merely the throughput, as the RTT also influences the *Retransmission Timeout (RTO)* of an active TCP session and the *Congestion Window (cwnd)* size [19]. The *cwnd* is also one of the most important TCP per-connection state variables. The *cwnd* is a TCP per-connection state internal variable that represents the maximum amount of data a sender can potentially transmit per RTT at any given point in time based on the sender's network capacity and conditions. TCP decides the maximum number of *bytes* that can be *outstanding* without being acknowledged at any time maintained independently by the sender.

Benefits: It is very natural to ask: *why RTT prediction performed in an intermediate node from passive measurements is important*? In addition to the reasons we address above, there are myriad reasons we may want to use passive RTT measurements. Passive RTT prediction in an intermediate node is important, for example, when *(i)* We have no control over either end-host of communication so we can't launch active measurements from either host, but want to know the RTT between them. *(ii)* We want to know the RTTs of the actual communication occurring on the Internet, and not the RTT between a pair of hosts artificially picked. *(iii)* The TCP active probes used in active measurements (such as *ping* messages) are blocked by firewalls etc. For more details about the difference between *active* and *passive* measurement techniques, we refer the reader to our previous work [12].

Recurrent Neural Networks (RNN) models: In this paper, we are interested in the capabilities and potentials of *RNN* models for implementing our passive RTT prediction model for TCP using *timestamps* and *timestamp echoes* [18]. Hence, we have explored an approach to dynamically predict an end-to-end RTT for TCP from passive measurements using *Long Short-Term Memory (LSTM)*-based RNN architecture. As described in Section III, different approaches have been proposed to estimate RTT from passive measurements. However, we believe that no previous research works have applied *deep learning* models to estimate RTT in relation to TCP from passive measurements. To the best of our knowledge, this paper is the first to study the applicability of LSTM for passive RTT measurement schemes in real-time.

RNNs are powerful neural sequence models that achieve *state-of-the-art* performance on sequential, time-dependent prediction and classification tasks. However, when the input sequence is very long, RNNs have a significant limitation of *gradient vanishing*. LSTM [15] is a special kind of RNN introduced with the purpose of overcoming this shortcoming of RNNs. LSTM has the ability to solve the *vanishing gradient* problem by dynamically controlling the information flow within the layers through its *memory blocks* and capture the long-term dependencies of the connections in a sequence effectively [15]. In an LSTM model, we consider a time-series prediction task of length n producing an output y_t at each time-step t$\in \{1, 2, 3, ..., T\}$ by mapping a temporal input feature vector sequence $x_1^n = (x_{(1)}, x_{(2)}, x_{(3)}, ..., x_{(n-1)}, x_{(n)})$ where $x_i \in \mathbb{R}^n$ to a corresponding output vector sequence $y_1^n = (y_{(1)}, y_{(2)}, y_{(3)}, ..., y_{(n-1)}, y_{(n)})$ where $y_i \in \mathbb{R}^n$ by calculating the network unit activations of a weighted sum using the Equations 2-7 iteratively from $t = 1$ to n. As it is shown in Equations 2, 3, and 5, LSTM [15] uses *three*, an *input*, *forget* and *output*, gates shared by all cells in the LSTM block in order to learn long-term dependencies and control the flow of information. The *input* gate determines the flow of input activation into the memory cell whereas the *output* gate determines the output flow of cell activation into the rest of the network. The *forget* gate determines the extent to which the current value remains in the memory cell of the LSTM unit before it gets gradually discarded when its data is no longer needed.

$$i_t = \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i) \quad (2)$$
$$f_t = \sigma(W_{fx}x_t + W_{fm}m_{t-1} + W_{fc}c_{t-1} + b_f) \quad (3)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c) \quad (4)$$
$$o_t = \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_t + b_o) \quad (5)$$
$$m_t = o_t \odot h(c_t) \quad (6)$$
$$y_t = \phi(W_{ym}m_t + b_y) \quad (7)$$

where the *i*, *f*, *c*, *o* are *input*, *forget*, *memory state*, and *output* gate activation vectors respectively at each time step *t*. σ is the logistic sigmoid function while $\odot$, *g* and *h* are element-wise product of the vectors, the cell input and output non-linearity activation functions of the entire neural network applied to each layer of the deep network respectively. *W* and *b* represents a vector of weighted recurrent connections and the bias vector. m_t is the hidden state output of the LSTM layer. Finally, ϕ is the activation function in the hidden layer applied to the network output. Figure 1 describes the basic unit of an LSTM network where the input sequence to the LSTM cell is carried over each time-step of *t-1*, *t* and *t+1*. C_t and C_{t-1} are the memory cell state activation vectors from the current and previous blocks at time *t* and *t-1* respectively.

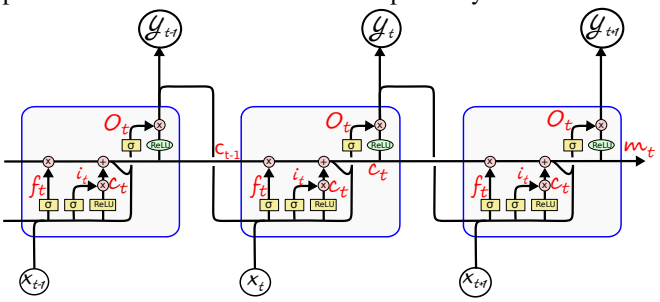

Fig. 1: LSTM Networks. For more thorough details, refer [28]

Why did we use deep learning? As explained above in Section I, both *cwnd* and RTT are TCP state variables relevant to congestion control. However, neither the value of *cwnd* nor the value of RTT is contained in the TCP header. The *cwnd* size is stored in the memory of the TCP sender, and the RTT is a product of the varying behavior of the network between the TCP sender and receiver. Therefore, trying to predict these values somewhere other than at the TCP sending node is challenging. Deep learning techniques have found a great success in multiple areas of research. In our case, let's consider a situation where a network model is trained for a specific intermediate node which has been trained for a specific bandwidth, background load, multiplexing rate, and a multitude of different router conditions, can predict well for exactly this node. Hence, we want a model that is able to train in one scenario setting and apply it as a pre-training on another setting by leveraging trained knowledge. As it is presented in Section VI, this paper proofs that it makes sense in principle to use learning algorithms for TCP state predictions.

Contributions: We summarize our main contributions below.

- We present a dynamic *deep learning*-based approach for RTT prediction in relation to TCP from passive measurements collected at an intermediate node.
- We identify the main challenges in the *passive* estimation of RTT across a broad range of network conditions.
- We show that the learned prediction model performs reasonably well by leveraging trained knowledge from the *emulated network* when it is applied and transferred on a *real-life scenario* setting.
- We demonstrate the benefits and explore the applicability of our prediction model using an *LSTM* architecture.
- We experimentally validate our prediction model extensively through several controlled experiments across an *emulated* and *realistic* settings.

II. Background

RTT measurements are used in congestion control algorithms to determine connection timeouts. *Delay-based* congestion control algorithms use the measured *RTT* as an implicit feedback to control congestion, and they adjust the *cwnd* size according to the queuing *packet delay* instead of packet loss [20]. These algorithms increase the *cwnd* size *quickly* when the queuing delay is low and decreases the *cwnd slowly* when the delay is high. Different *rate* and *delay-based* TCP stacks come with a variety of features that will violate the assumptions we might make if we only look at one or two TCP implementations. Hence, the following are a list of the most widely used TCP variants we consider in our analysis to cover the whole scope of the problem.

1) **TCP Westwood**: Westwood [10] is a sender-side modification of the traditional TCP algorithm [17]. At the time of congestion triggered in response to RTO or *triple DupACKs*, TCP Westwood [10] estimates the available end-to-end per-connection bandwidth by monitoring the flow of returning ACK rates instead of packet loss and sets the *cwnd* size equal to the measured bandwidth which helps to avoid too much reduction of the *cwnd*.
2) **TCP-Vegas**: Vegas [5], instead of packet loss, uses a measured RTT as congestion feedback and hence it attempts to accurately tune the *cwnd* by using the measured *BaseRTT* of every segment sent and reacting to changes in it by altering the *cwnd*.
3) **BBR**: BBR is an emerging TCP delay-controlled congestion control algorithm from *Google* fully deployed across all *Google* TCP services and the B4 Wide Area Network (WAN) backbone connections [6]. Unlike the traditional congestion algorithms, BBR doesn't overreact to packet loss. Instead, it reacts to actual congestion and relies on maximizing the throughput with minimal queue by sequentially probing and periodically estimating the underlying available bottleneck bandwidth and minimum path RTT in a similar fashion as TCP Vegas [5].

Roadmap: The rest of this paper is organized as follows. Next, in Section III, we summarize the related work in the literature considered as a *state-of-the-art*. In Section IV, we describe our controlled experimental setup for the evaluation. Section V gives an overview of our methodology highlighting the practical challenges and considerations. Section VI presents the validation scenario settings of our prediction model. The experimental results and discussion are presented in detail in Section VII. Finally, Section VIII concludes the paper and outlines directions of research for future extensions.

III. Related Work

Our work benefits from a wide range of existing passive measurement related research works in computer networking.

TCP RTT Measurement: TCP implements a retransmission strategy by setting the *time-out* interval to ensure data delivery in the absence of any acknowledgment for a particular segment from the receiver side [30]. The timer relies on the measurement of the network latency which TCP does by periodically estimating the current RTT of every active connection in order to determine the RTO when it sends data and receiving an acknowledgment for it. Accurate measurement of RTO is crucial to TCP performance and it is determined by estimating the *mean* and a *variance* of the estimated RTT [30]. When the timer RTO expires, the segment is retransmitted. To compute the current RTO, TCP sender keeps track of the *Smoothed Round-Trip Time (SRTT)* and the *Round-Trip Time Variation (RTTVAR)* state variables. When the first RTT measurement *R* is made on the active connection, the host should compute the following *Jacobson RTO Estimation algorithm* [17].

$$\texttt{SRTT} = \texttt{R} \tag{8}$$

$$\texttt{RTTVAR} = \texttt{R}/2 \tag{9}$$

$$\texttt{RTO} = \texttt{SRTT} + \max(\texttt{G}, \texttt{K} * \texttt{RTTVAR}) \tag{10}$$

However, when a subsequent RTT measurement *RTT'* is made, the host should compute the following algorithm [17].

$$\texttt{RTTVAR} = (1 - \beta) * \texttt{RTTVAR} + \beta * |\texttt{SRTT} - \texttt{RTT'}| \tag{11}$$

$$\texttt{SRTT} = (1 - \alpha) * \texttt{SRTT} + \alpha * \texttt{RTT'} \tag{12}$$

$$\texttt{RTO} = \texttt{SRTT} + \max(\texttt{G}, \texttt{K} * \texttt{RTTVAR}) \tag{13}$$

Understanding RTO: In a typical implementation, TCP computes the RTT of an active connection using the *Exponential Weighted Moving Average (EWMA)* estimator [17]. As it is used in TCP RTT computation implementations, the *SRTT* is also updated using the *EWMA* estimator as it is shown in Equation 12 where the *smoothing* factor (α) = $\frac{1}{8}$ [30]. RTTVAR is also calculated using *EWMA* as shown in Equation 11 where the smoothing gain of the samples β (*variance factor*) = $\frac{1}{4}$. The new value of RTO is given in Equation 13 as a function of *SRTT* and *RTTVAR* where *K* is usually 4 and *G* is a clock granularity in *seconds*. The TCP sender, dynamically adjusted based on the estimated RTT, keeps a timer which activates retransmission of packets that have not been acknowledged before the RTO expires [30]. After computing the RTO, if its value is less than *1 second*, then the RTO value should be rounded up to *1 second* [30]. However, the *timeout* can expire spuriously across low-bandwidth network paths and triggers unnecessary retransmissions when no packets have been lost [11]. Modern operating systems like *Linux* have a minimum value for RTO in order to avoid unnecessary high retransmission delays of an open active connection. The potential pitfall of choosing a low RTO, however, is that it may trigger retransmission of a packet even though the segment is received and an ACK is on

its way. Setting a low value for RTO works better when there is a moderate background traffic [26]. For more technical descriptions and the rules governing the measurement of *SRTT*, *RTTVAR*, and *RTO* refer [30].

Algorithms for Avoiding Spurious Timeouts: To address this problem, a number of approaches have been proposed. For example, RTO estimators like [30] are based on the assumptions of older technologies. As described earlier, spurious timeouts lead to problems that cause several unnecessary retransmissions and congestion control back-off that affect the TCP throughput. In addition to this, estimating the RTT measurements are challenging in the presence of timeouts and packet loss in the end-to-end path. This is because on the receipt of an *ACK* after *R* retransmissions, the sender cannot tell which one of the *R+1* data sent is being acknowledged which again affects the measurement of *SRTT* shown in Equation 12. Wrongly computed *SRTT* values will eventually lead to wrong *RTO* values. If the value of *RTO* is too *small*, it will lead to unnecessary retransmission of data segments which again increases the load on the underlying network capacity. But if the value of *RTO* is too *large*, the sender waits too long before retransmitting lost segments which again increases delay and lowers the throughput for connections with packet loss. Making use of the TCP *timestamps*, the *Eifel* [11] algorithm has pointed out that it is possible to detect spurious TCP timeouts problems and recover by restoring a TCP sender's congestion control state saved before the timeout.

There are other previous research works who have examined and reported RTT estimation for TCP [2, 21, 22]. The approach presented in [22] uses a *unidirectional* flow during the TCP *handshake* of a connection to estimate RTT using the time from *SYN* to *SYN+ACK* method. The approaches proposed in [22] calculates one RTT sample per TCP connection associated either during the *three-way handshake* or during the *slow-start* phase. If we have captured the TCP *three-way handshake* as presented in [22], we can calculate the *initial RTT* (*iRTT*) by taking the time difference from the *SYN* packet to the *ACK* packet of the handshake. However, since the TCP handshake packets are processed by the kernel, the RTTs during the data transfer will probably be slightly larger than the *iRTT*. Hence, this approach may tend to underestimate the actual RTT. In addition to this, since TCP sets the initial *retransmission timeout* value to *3 seconds* [30], therefore this approach is not applicable in scenarios where the TCP connection setup takes longer which leads to long delays and packet losses introduced by the network. The study in [2] has reported a statistical characterization of RTT variability where the measurement point is closer to the sender. However, their study does not take *delayed ACKs* into account. The authors of [21] have introduced an approach for RTT measurements of TCP connections based on *bidirectional* traces captured at the monitoring point using a *Finite State Machine (FSM)* that *replicates* the TCP sender states of observed ACKs depending on the underlying TCP flavor. The authors have pointed out that the estimation of the TCP parameters (e.g., *cwnd*) may have potential errors primarily due to *over-estimation* of the RTT and incorrect window sizes of a connection [21]. Another limitation of this work, given differences of the many existing flavors of TCP stack implementations, the use of a separate state machine for each TCP variant is *unscalable* and we also believe that the constructed *replica* may not manage to reverse or backtrack the transitions taking the amount of data into consideration. In addition to this, the *replica* may also not observe the same sequence of packets as the sender and ACKs observed at the intermediate node may not also reach the sender. Our *deep learning*-based approach to passively predict the continuous RTT measurement throughout the lifetime of a TCP session builds upon these classical approaches by avoiding the limitations taking advantage of the commonly used *timestamp* option as explained in Section V.

IV. Experimental Evaluation

A. Testbed Hardware

We have carried out our experiments using a cluster of HPC machines based upon the GNU/Linux operating system running a modified version of the *4.15.0-39-generic* kernel release. The prediction model is performed on an NVIDIA Tesla K80 GPU accelerator computing with the following characteristics: Intel(R) Xeon(R) CPU E5-2670 v3 @2.30GHz, 64 CPU processors, 128 GB RAM, 12 CPU cores running under Linux 64-bit. All nodes in the cluster are connected to a low latency 56 Gbit/s Infiniband, gigabit Ethernet and have access to 600 TiB of BeeGFS parallel file system storage.

B. Passive RTT Monitoring Methodology and Trace Analysis

Passive measurement methodology is a technique of tracking the behavior and characteristics of packet streams where the network is not influenced by injecting extra traffic. More details on the two types of network measurement technique categories (i.e., *active* and *passive*) are briefly described in [12]. The RTT seen by a TCP segment is defined as the time a sender waits until it receives a corresponding *ACK* from the receiver before it sends more data packets. In this paper, we are interested in presenting a *deep learning-based* RTT prediction model using a packet statistics passively monitored between the sender and receiver endpoints of a network. In order to increase the RTT measurement precision, the *timestamps* option which every TCP segment carries in the header field is used in our methodology. When the server receives a data segment, it copies the *timestamps* into the ACK and this, in turn, enables the client to compute the RTT accurately for every acknowledged data segment.

Trace Analysis: To evaluate our prediction model and perform our analysis on both the *emulated* and *realistic* network conditions, we have generated our own dataset. In order to capture all sessions on the network when the client and server are sending TCP packets and measure the TCP data packets from both directions, we have used the fully controlled experimental setup shown in Figure 2. The data passively collected at an intermediate node is fed into a model that can be trained in another context, e.g., an emulated scenario as discussed in Section VI. The background traffic for all our experiments are generated using the *iperf* [8], an open source TCP streaming benchmark, traffic generator on an emulated LAN link where we run each TCP variant by adding a configurable variation of the emulation parameters *bandwidth (in Mbit/s)*, *delay (in ms)*, *jitter (in ms)* and *packet loss (%)* within a flow. The values of configuration parameters of the emulator for our samples collection are presented in Table I.

The cross-traffic variability and verification of the popular Linux-based network emulator we used, *Network Emulator (NetEm)* [14], are thoroughly addressed in [12].

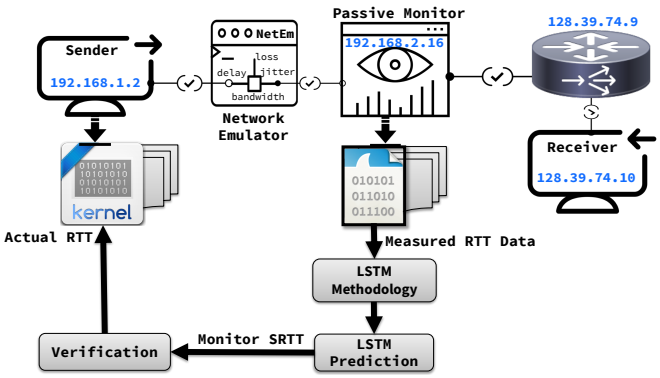

Fig. 2: Controlled Experimental Setup

TABLE I: Network *emulation* parameters

Bandwidth (*in Mbit/s*)	**Delay** (*in ms*)	**Jitter** (*in ms*)	**Packet loss** (%)
10	1	0.001	0.01
100	2	0.1	0.05
300	3	0.2	0.1
500	5	0.5	1
700	7	1	1.5
1000	10	2	2

Verification of the Emulator: Given that the software emulator is not precise, we can ask: can we trust the network emulator for all the variations of *bandwidth*, *delay*, *jitter* and *packet loss* values introduced by the emulator irrespective of the measurement we get from TCP stream? As the precision of the emulator cannot be measured from TCP streams, we set up a different experiment using *UDP* to evaluate and measure the precision where both the emulator and traffic generator create variations. We verified the raw performance by measuring the *bandwidth*, *delay*, *jitter* and *packet loss* variations created by the traffic generator and network emulator at the receiver side and we found out that each variation run by the emulator doesn't affect our results. But it is good to remember that emulator experiments are always going to have some differences compared to a real network as different networks behave completely differently.

V. Experimental Methodology

In this paper, we are exploring an approach to *dynamically* and *reliably* predict an end-to-end RTT for TCP from passive measurements using an *LSTM*-based RNN architecture. As illustrated in Figure 3, there are different techniques to estimate passive RTT values from packet arrival times at an intermediate monitoring point. The *first* and *third* RTT computation techniques shown are the *initial three-way handshake* [22] and the *termination* of a connection phase that carries the *FIN* control flag. The *three-way handshake* method uses a TCP segments association during the initial handshake phase to compute the *minimum RTT* with a small packet burst (so less affected by propagation delay through intermediate devices) but it also has limitations as described in Section III. A similar estimation technique can also be applied during the connection termination phase. However, these *two* techniques do not consider continuously estimating RTT through the course of the connection and hence they are *statically* limited to the setup and termination of the TCP connection. In our paper, however, we are interested in the *second* estimation method where we have to account for the cases where there are large packet bursts by *continuously* measuring the TCP data segments sequence and their corresponding ACK for estimating RTTs when they carry data throughout the lifetime of the connection by associating the *timestamps* and *timestamp echoes*. This helps us to dynamically estimate the RTT between the sender and receiver from the perspective of the intermediate node by measuring the streams of RTT samples which can be added to get an end-to-end RTT. Let's simplify this more with an easy to understand example. When the sender, on Figure 3, sends a TCP data segment, the receiver acknowledges the data segment with an ACK and echoes the sender's *timestamp*. The intermediate node recognizes the sender's *timestamp* in both data segments and associates the data segments with *timestamps* matching. When the sender receives an ACK, it sends more data packets by echoing the receiver *timestamps*. The intermediate node captures this data segment, it recognizes the receivers *timestamps* in both data segments and forms an association. Finally, with a *timestamp* matching of all these data segments, the intermediate node can observe a full estimated RTT. Hence, in order to reliably associate the data segments with its corresponding ACK that triggered it, compute accurate RTT and avoid the ambiguity between delayed and retransmitted segments, we have employed the TCP *timestamps* option.

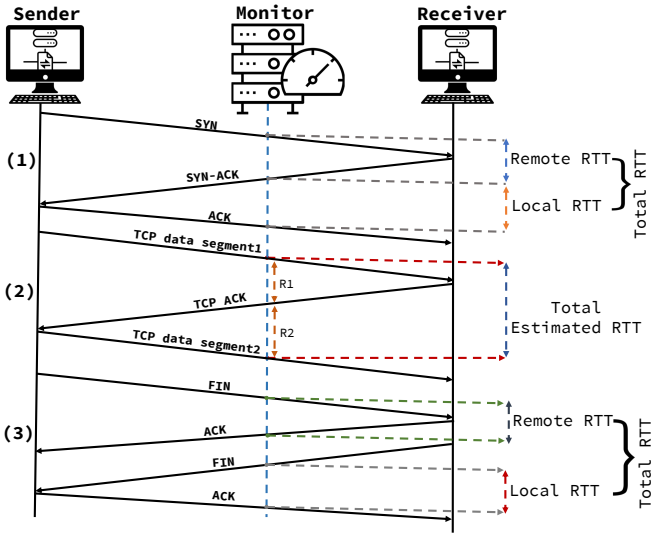

Fig. 3: Passive RTT estimation techniques

Timestamps option: The reason why we used the *timestamp* option in our evaluation is to avoid the impact of incorrect (spurious) *timeouts* and get the *accurate* RTT measurement. Anytime TCP experiences spurious timeouts unnecessarily, it significantly suffers from unnecessary retransmissions and congestion control back-off. This, in turn, triggers TCP to drop the *Slow Start Threshold (ssthresh)* to half the current *cwnd* and reduces the value of *cwnd*. This is because when the retransmission of a lost packet is as a result of the RTO value expiration, TCP cannot infer anything about the *state* of the network. Unless TCP uses the *timestamp* option while sending the retransmitted packets, it cannot correctly measure

the RTTs for those packets [23]. In addition to this, research studies like the *Eifel* mechanism have shown that the *timestamp* option substantially improves the overall TCP connection's performance over paths with a large *Bandwidth-delay product (BDP)* [11]. Since TCP is a symmetric protocol, allowing data segments to be sent and received at any given time in both directions, the *timestamp* options are also specified in a symmetrical manner [18] as they can be sent and echoed in both directions. This means the actual *Timestamp Value (TSval)* added by the sender is carried in both the ACK and data segments are echoed in *Timestamp Echo Reply (TSecr)* fields carried in the returning ACK or recently received data segments [18]. In this way, we can avoid *underestimating* the actual RTT measurement.

A. Practical Challenges

The TCP congestion control algorithms that are widely deployed today perform the most important parameters used for TCP performance evaluation such as handling the *cwnd* and RTT from the sender-side. In this paper, we are interested in passively inferring an end-to-end RTT for TCP from packet arrival times collected at an intermediate monitoring point of a network without having access to the sender. However, there are some practical factors and concerns which complicate the implementation of a passive RTT measurement from an intermediate node. Here, we describe these practical challenges and the approaches how we address them in our evaluation.

Fig. 4: RTT computation scenarios

Let's suppose two end-points are exchanging traffic in both directions as shown in Figure 4. If we measure at the intermediate point, one approach is to measure from the perspective of the sender in both directions. This is because if we measure the RTT at the receiver side, we cannot be sure when we send an ACK that it will trigger a new sent packet at the sender. In addition to this, the sender might not have any data to send, or the sender may also still have opportunities to send without receiving an ACK. If the sender sends data to the receiver, and the receiver sends a corresponding ACK, then the *monitor* could measure *monitor-to-receiver-to-monitor* part of the RTT by observing the data packet and the associated ACK. Similarly, if the receiver sends data to *sender*, and *sender* sends an ACK, then the *monitor* could measure the *monitor-to-sender-to-monitor* part of the RTT by observing the data segments and the associated ACK. Finally, these two values could be added together into an observation of an estimated RTT. However, there are a number of limitations that require consideration: *(i)* many connections send traffic mainly in one direction, rather than in both directions *(ii)* since Internet routing is not necessarily symmetric (i.e., the path from sender to receiver is not necessarily the reverse of the path from receiver to sender), the monitor might not be on the path in both directions between sender and receiver, and between receiver and sender. *(iii)* The RTT estimate is combined from two different data-ACK pairs at different times. In order to get a more reliable and accurate estimation, our passive RTT prediction model takes advantage of the TCP *Timestamps* option (see Section V).

Measuring at both endpoints: What happens if we capture the traffic at both the sender and receiver endpoints and do the RTT estimation separately? There will be a difference in the timing of the received data which will affect the RTT estimation. It means this, to get a good measure of the raw one-way RTT for each direction, would require clock synchronization between the sender and receiver endpoints. We have no way to combine these two clocks with any strong guarantees unless the two endpoints are reasonably synchronized (e.g., by using GPS signals). However, it is important to remember that there isn't a timing difference for predicting the underlying TCP variant since it simply measures the change in *cwnd* size.

Sender idle time: *How do we technically handle the idle time (delay) of the sender when the buffer is not full and the sender waits for enough data to be pushed*? As explained above, since the queue is *one-way*, the idle time in the sender when there isn't enough data to send doesn't have an impact on the *network propagation time*, but it does for an application latency. Hence, this is both application and implementation dependent as there are many applications where the sender has nothing to send. We may have a transmitting delay when there is a lost packet that triggers a *dupPACK*. In the presence of a packet loss or *out-of-order* packet, the response will come right away. Basically, if the sender is application limited, measuring RTT on the *three-way handshake* is very reliable. However, we are interested in when a segment carries data. In order to passively estimate RTT, we measure all the sequence numbers of the data segments going in both directions at the intermediate node and their corresponding ACK only if they carry data. If the monitoring point is somewhere in the path, all we observe is packets flowing back and forth, and we can't tell the difference between *network* and *application* latency. Hence, we may treat them both the same way while data is being exchanged between the sending and receiving endpoints. The inclusion of TCP *timestamps* was supposed to help in these calculations independently by observing the *timestamps* sent and echoed in both directions and provide an improved RTT estimation.

Multiple packets with the same sequence number: The fact that TCP can send multiple packets with the same sequence number is a challenge. For example, if we send a packet with a sequence number and an ACK, *how do we know when the other packets have been sent if we see another packet with the same sequence number*? This is challenging and highly depends on whether the connection is using TCP *timestamps* or not. As explained in detail above, if TCP *timestamps* are not enabled, RTT samples cannot be safely computed due to the *retransmission ambiguity*, and thus *unreliable*. However, if TCP *timestamps* are enabled, then the sender can accurately compute an RTT sample even for retransmitted data using the *Timestamp Echo Reply (TSecr)* [18]. This is one of the main reasons why we are using the *timestamps* options for our RTT measurement scheme. If more than one *Echo Reply* of a data packet is received before a reply segment is sent, our model estimates the RTT using the latest transmission time of most recently sent data packet with the oldest sequence number while ignoring the data packets with the earliest transmission time. This helps us to avoid spurious retransmissions.

B. Considerations

Here is the list of TCP mechanism we consider in our analysis.

Delayed ACKs: During our RTT passive measurement scheme, we took regular *delayed ACKs* into account by leaving the *delayed ACK* enabled since major operating systems enable it by default for TCP even though the algorithms and constants are slightly different for each operating system. Most of the widely-deployed TCP variants nowadays will have the *delayed ACK* tag switch on by default to reduce network overhead. That means the TCP implementation would have to deal with whatever ACK algorithm the receiver is using and the receiver can wait up to *500ms* (common TCP implementations delay the ACK only up to *200ms*) before it sends an ACK in the hope that it can save the packet [4, 18]. So most of the TCP variants nowadays are configured in such a way that they are allowed to send an ACK for every second full-sized segment. In order to do that they receive data and wait for up to *200ms*. If nothing else comes, they send an ACK – if something else comes, they send a cumulative ACK for both. However, it is necessary to remember that the receiver's *delayed ACK* mechanism, noisy links, and other factors may introduce bias by causing systematic overestimation of RTT derived from Data-ACK matching (especially for slow-moving TCP connections like an interactive *telnet* or *ssh* session). In our analysis, to eliminate this bias when an ACK covers multiple packets, we used only the RTT from the latest data packet that is ACKed. It can also be done by filtering out the ACKs of unacknowledged data segments whose value is less than *2*MSS* since those are quite possibly *delayed ACKs* [4]. TCP is generally supposed to delay ACKs until either: *(i)* at least 2 full MSS of data has arrived, in which case the TCP receiver should send an immediate ACK, or *(ii)* the delayed ACK timer fires. If we get an ACK that is for *>= 2*MSS* of data, then there is a very good chance that the ACK was triggered by *(i)*, in which case the latest data that arrived was probably ACKed immediately. If we get an ACK that is for less than *2*MSS* of data, it was probably triggered by *(ii)*, the delayed ACK timer, and should be filtered out if we want the RTT of the network path. This is precisely one of the reasons why we avoid packet sizes over the regular legitimate *MSS* in our experiments by disabling TCP segmentation offloading as described below.

Maximum Segment Size (MSS): Our experiments are carried out over a path that is jumbo-frame clean by disabling TCP segmentation offloading in order to avoid packet sizes greater than the regular legitimate values. If we measure at a higher level and when packets are pushed down layer by layer on the protocol stack, the negotiated MSS will be violated. This means when the data size is greater than the legitimate MSS, the messages will be split into several frames with a higher chance of unnecessary retransmissions which will introduce processing delays that affect the time it takes to send the data. Therefore, in order to avoid this violation, we made sure that each TCP flow uses a standard Maximum Transmission Unit (MTU) value of *1500-byte* data packets.

C. Impact of the Underlying TCP Variants

We believe RTT is generally unaffected by the underlying TCP congestion algorithm that is being used, except *indirectly* due to ambiguous retransmissions which will probably make some RTTs seem longer when congestion is detected. TCP congestion avoidance algorithms specify: *(i)* How much should the current packets per burst be reduced when there is a packet loss, and *(ii)* How should that number be increased for each RTT. As described in Section II, RTT between two endpoints is typically a combination of a fixed *baseRTT*, the propagation time and whatever amount of queueing is experienced along the way. The propagation time is not a function of congestion control or even of TCP, it's the same for any IP packet. Therefore, we believe that there is no direct impact of the underlying TCP variant on a per-packet measured RTT.

VI. Validation Scenarios

Our model has been validated on the following settings.

Emulated setting: As illustrated in Figure 2, we used the *measured* RTT data from the intermediate node as an *input* to our methodology for an inference of the RTT prediction. Finally, we verified the predicted RTT with the actual TCP *timestamps* directly logged from the Linux kernel used only for training and generate new data for the learning model to predict on. Once we finish with the verification, we run our learning model and get the predictions. We validated our methodology using the experimental testbed shown in Figure 2 over a LAN link. In order to train and test our prediction model, we employed a single trained network that adapts to all experiments with variations of *bandwidth*, *delay*, *jitter* and *packet loss* into one learning model. We have demonstrated that our model can also be applicable in real networks.

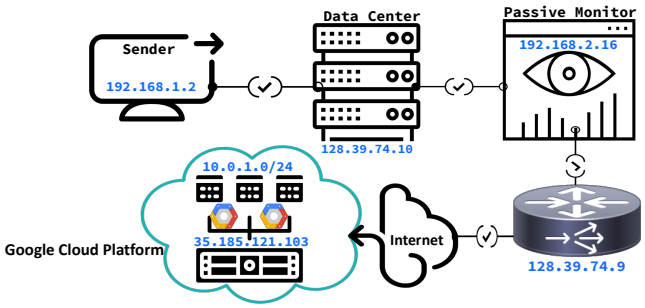

Fig. 5: Realistic Scenario Setup

Realistic setting: The ability to use embeddings of a model trained on an emulated environment to a realistic setting is a huge advantage in terms of scalability, applicability, and robustness. In this paper, we are able to train in one scenario setting and apply it as a pre-training in another scenario setting. Therefore, we are able to show that the learned passive RTT prediction model by leveraging a pre-trained knowledge of the LSTM network from the *emulated* network performs reasonably well as it is shown in the results when it is applied and transferred to a *realistic* scenario setting bearing similarity to the concept of *transfer learning* in the machine learning community [32]. Here, we rely on passive measurements of real-world TCP network trace to evaluate the effectiveness of our model. This guarantees that our LSTM-based RTT prediction model is able to discern the results to unforeseen scenarios. As shown in Figure 5, we performed a realistic experiment using Google cloud Virtual Machines (VMs) hosted across different regions. The experimental results of our realistic scenario are presented in Figure 7.

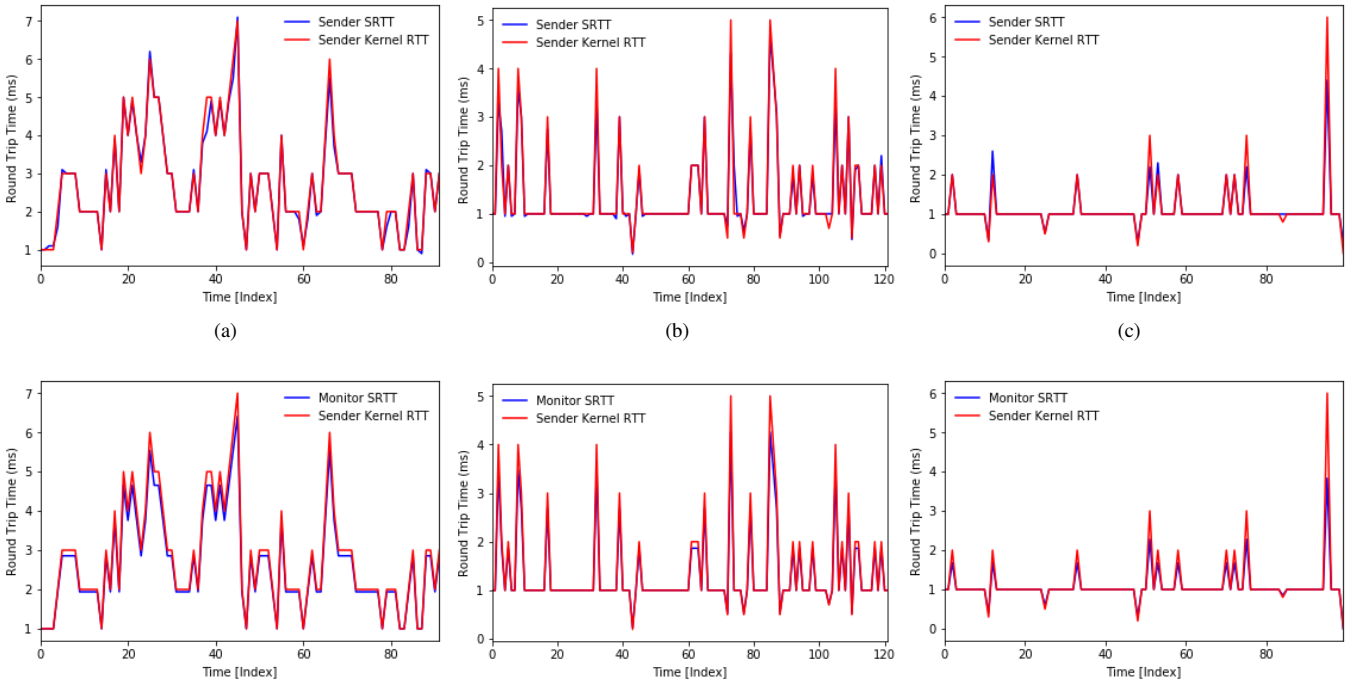

Fig. 6: RTT Prediction results comparison of an *emulated* setting between the TCP sending node and the intermediate node. **(a)-(d)**, TCP Westwood [10]. **(b)-(e)**, TCP Vegas [5]. **(c)-(f)**, TCP BBR [6].

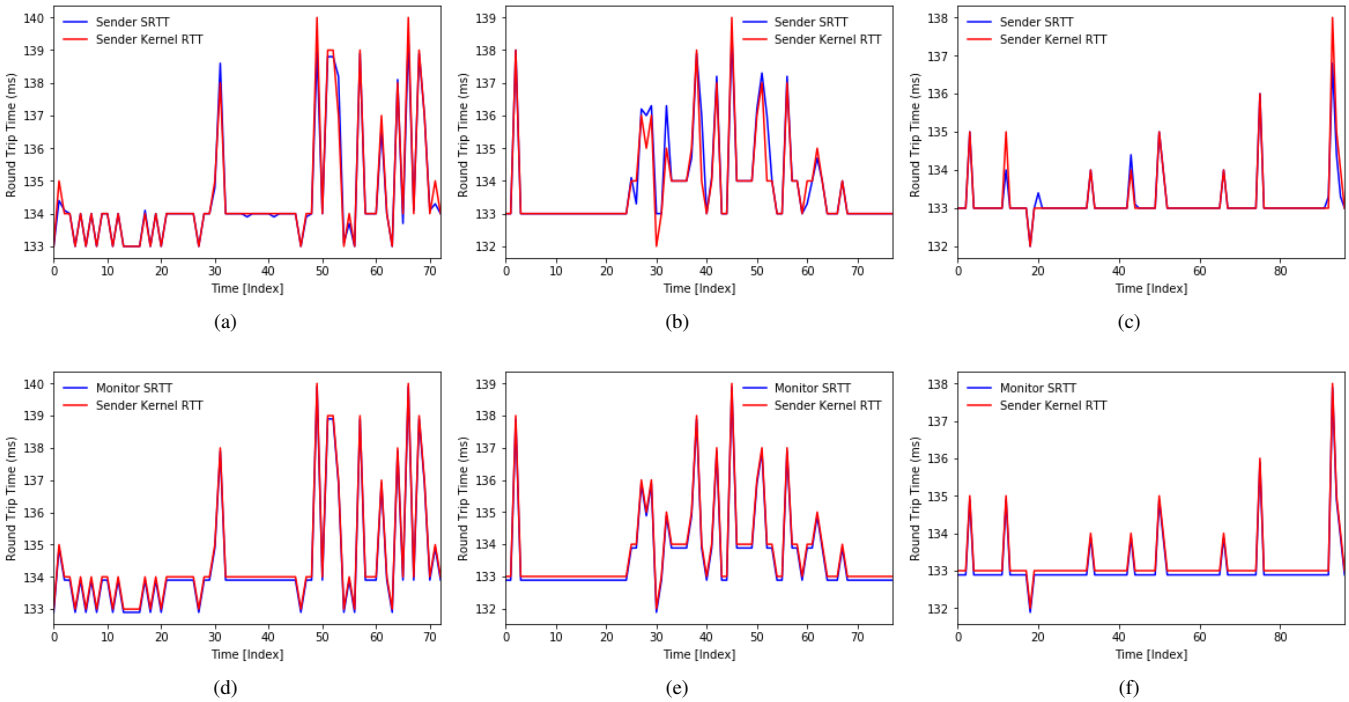

Fig. 7: RTT Prediction results comparison of a *realistic* setting between the TCP sending node and the intermediate node. **(a)-(d)**, TCP Westwood [10]. **(b)-(e)**, TCP Vegas [5]. **(c)-(f)**, TCP BBR [6].

VII. Experimental Results and Discussion

On Section I, we have justified the choice of deep learning-based approaches in our paper. In this section, we will explain how the key features of deep learning and their implementations are being exploited.

Implementation details: We implemented our RTT prediction model in *Python* using the *Keras* deep learning framework with *Google's TensorFlow* backend [1] running on NVIDIA Tesla K80 GPU where we apply an LSTM-based architecture to estimate the *RTT* trained over multiple epochs by taking the RTT samples as values in time-series. As shown in Figure 1 at each time-step of *t*, as a learning process, the LSTM model takes an entire array of the Data-ACK matching based on *timestamps* captured on the monitoring point between the sender and receiver as an input feature vector (x) indexed by *timestamps* obtained from the kernel. We propagate the input to the model through a multilayer LSTM cell followed by a dense layer of 15-dimensional hidden states with *Rectified Linear Unit (ReLU)* activation function for the different layers that generates an output of a sequence dimensional vector of predicted *RTT* (y_t) of the same size indexed by *timestamps*. Our LSTM network is trained using the *Truncated Back Propagation Through Time* (*TBPTT*) training algorithm for modern RNNs applied to sequence prediction problems [31]. We used this training algorithm to minimize LSTM's total prediction error between the expected output and the predicted output for a given input of the measured RTT time-series. We trained our LSTM-based learning algorithm without the knowledge of the input features from the TCP sender-side during the learning phase. We learn the model from the training data and then finally predict the test labels from the testing instances on all variations of the emulation parameters. In order to train our prediction model more quickly, and get a more stable and robust to changes RTT estimation model, we have applied one of the most effective optimization algorithms in the deep learning community, the *Adam* stochastic algorithm [24] with an initial *learning rate* of *0.001* and *exponential decay rates* of the first (β_1) and second (β_2) moments set to 0.9 and 0.999 respectively. Totally, all of our configurations were trained for a maximum of 100 epochs with the mini-batch size of 256 samples. We further optimize a wide range of important hyperparameters related to the neural network topology to improve the performance of our prediction model. In order to train and test our prediction model, we employed every experiment with a ratio of *60%* training, *40%* testing split and a 5-fold cross-validation on all variations of *bandwidth*, *delay*, *jitter* and segment *loss* into one learning model.

Evaluation metrics: In order to evaluate and measure how well our LSTM-based prediction model performs in terms of capturing the time-series *RTT* patterns under different network conditions, all the neural networks are trained, as it is shown in Tables II and III, by employing both the *Root Mean Square Error (RMSE)* and *Mean Absolute Percentage Error (MAPE)* performance metrics. The *RMSE* measures the root average squared error between the predicted and actual value, while MAPE measures the absolute deviation between the predicted and actual value as a percentage. The well-known *RMSE* and *MAPE* metrics are both means of estimating the point-wise errors in predictions and it is good to remember that these metrics don't depend on RTT sample sizes. Hence, the metrics values do not change for different numbers of samples in the output of the neural network. This is because the sum increases with the number of summed elements, but *mean* is sum divided by the number of elements, so it's "per element".

TABLE II: Prediction accuracy of an *emulated* setting

	Kernel RTT vs. Sender SRTT		*Monitor SRTT vs. Kernel RTT*	
TCP Algorithms	**RMSE**	**MAPE (%)**	**RMSE**	**MAPE (%)**
Westwood [10]	0.1587	0.8632	1.4916	1.7391
Vegas [5]	0.1854	0.6341	0.7289	0.6581
BBR [6]	0.2103	1.0217	0.5733	1.2812

TABLE III: Prediction accuracy of a *realistic* setting

	Kernel RTT vs. Sender SRTT		*Monitor SRTT vs. Kernel RTT*	
TCP Algorithms	**RMSE**	**MAPE (%)**	**RMSE**	**MAPE (%)**
Westwood [10]	0.2504	1.3185	1.5277	1.8479
Vegas [5]	0.4155	2.3097	1.8327	2.5103
BBR [6]	0.2714	0.8730	1.7942	2.0715

Discussion: We start by exploring in detail the practical challenges in the dynamic inference of RTT for TCP connections in IP networks from passively monitored traffic. The plots presented in Figures 6 and 7 show the RTT prediction as a function of the elapsed time since a packet is sent until a corresponding ACK is received at the sender (*y-axis*) and index of *time* in *seconds* (*x-axis*) of various TCP variants under a wide range of network conditions and validation scenario settings. The general sawtooth patterns of the time-series RTT prediction plots we presented in Figures 6 and 7 are consistent with the behavior of each TCP variant considered. The minimum RTT during a given time window begins at around *1ms* in the *emulated* and *133ms* in the *realistic* setting but slowly ramp up to the maximum. We believe that this happens because the packets are being queued somewhere. When the queue gets filled, packets begin to be dropped and therefore, the RTTs level off. They level out because we see RTTs for packets that have been at the end of the queue. However, with TCP BBR as it is shown in Figure 6 (**c**), (**f**) and Figure 7 (**c**), and (**f**), the RTT can go down because when BBR notices the queue, it decides to send slower to drain it even if there are no packet drops [6]. Our measurement results show that we achieve high accuracy of the RTT pattern across different settings. We performed several experiments that illustrate our main approach under multiple scenarios settings and different configurations. However, due to lack of space, the experimental results presented in Figures 6 and 7 are a subset of the configurations for a proof of concept to show that our prediction model is applicable both in an *emulated* and *real-world* settings.

The experimental comparisons on both scenarios presented on Tables II and III are between the actual RTT values we obtained from Linux kernel of the TCP sending node against the SRTT of RTT samples collected on the sender. The second column on both tables compares the estimated SRTT of the intermediate node (passive monitor) against the actual RTT value of RTT obtained from Linux kernel of the TCP sending node. For a more detailed explanation and definition of SRTT, we refer the reader to Section III of this paper. On Tables II and III, *Kernel RTT* is the actual RTT value used by the Linux kernel of the TCP sending node. Whereas, *monitor*, as shown

in Figure 2, is the *intermediate* node between the sender and the receiver. Tables II and III show the performance of our model in an *emulated* and *realistic* scenarios, respectively, and we observe that our prediction model performs comparably well in both validation settings.

Optimality: The experimental results show that our deep learning-based RTT prediction model performs with high accuracy across different validation scenarios.

VIII. Conclusion and Future Work

This work demonstrates how methods from the field of Artificial Intelligence (AI) can in principle aid in solving network-related complex problems. Under different variants of TCP, RTT is a property of the path between the sender and receiver whose value influences the dynamics of TCP. Hence, an accurate and dynamic estimation of RTT is crucial for TCP to maximize fair-share of the network resources. It provides useful information for network operators in investigating the critical factors that limit a flow rate and cause a congestive collapse in their networks. In this paper, we have proposed and evaluated a novel LSTM-based prediction model capable of dynamically predicting at real-time the RTT between the sender and receiver with high accuracy based on passive measurements collected at an intermediate node, taking advantage of the commonly used TCP timestamps. We explored in detail a set of practical methodological challenges and considerations involved in performing inference of RTT dynamically and reliably from passive measurements. The primary contribution of our work is building a prediction model that works well for *transfer learning*. We have demonstrated the efficiency of our model through extensive experiments both on a controlled *experimental testbed* network and in a *realistic* scenario setting on the *Google Cloud platform*. As future work, we would like to explore extensions in greater detail to the model we have presented across a broad range of different network conditions and multiple simultaneous TCP connections. By design, unlike *loss-based* algorithms, the *back-off* parameter of *delay-based* congestion control algorithms is not fixed which makes it fundamentally challenging to predict the TCP variant from passive traffic when there is variability in *delay*. Hence, now that we are able to predict RTT with a high accuracy, we believe extending our work in developing a *delay-based* pattern mining methodology that identifies the underlying *delay-based* TCP flavors from passive traffic and real measurements over the Internet using the RTT prediction model as an input vector is a promising direction for future research.

Acknowledgment

We would like to sincerely thank Professor Øivind Kure for his comments on the final draft of our paper.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *ACM SIGCOMM*, 2003.

[3] P. Benko and A. Veres. A passive method for estimating end-to-end TCP packet loss. In *GLOBECOM'02*. IEEE, 2002.

[4] R. Braden. Requirements for Internet hosts-communication layers. Technical report, 1989.

[5] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[6] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. 2016.

[7] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM*, 2000.

[8] ESnet. iperf3. https://iperf.fr/iperf-servers.php, 2017.

[9] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. Springer, 2001.

[10] M. Gerla, M. Y. Sanadidi, R. Wang, A. Zanella, C. Casetti, and S. Mascolo. TCP Westwood: Congestion window control using bandwidth estimation. In *GLOBECOM*. IEEE, 2001.

[11] A. Gurtov and R. Ludwig. Responding to spurious timeouts in TCP. In *INFOCOM*. IEEE, 2003.

[12] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. General TCP State Inference Model from Passive Measurements Using Machine Learning Techniques. *IEEE Access*, 2018.

[13] M. Hanai, S. Yamaguchi, and A. Kobayashi. TCP Fairness Evaluation with Modified Controlled Delay in the Practical Networks. ACM, 2018.

[14] S. Hemminger et al. Network emulation with NetEm. 2005.

[15] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*. ACM, 1988.

[17] V. Jacobson. Congestion avoidance and control. *ACM*, 1995.

[18] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, 1992.

[19] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Transactions on Networking*, 2003.

[20] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM SIGCOMM Computer Communication Review*, 1989.

[21] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring TCP connection characteristics through passive measurements. In *INFOCOM*. IEEE, 2004.

[22] H. Jiang and C. Dovrolis. Passive estimation of TCP round-trip times. *ACM SIGCOMM*, 2002.

[23] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM SIGCOMM*, 1995.

[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] M. Kühlewind, S. Neuner, and B. Trammell. On the state of ECN and TCP options on the Internet. In *International Conference on Passive and Active Network Measurement*. Springer, 2013.

[26] A. Loukili, A. L. Wijesinha, R. K. Karne, and A. K. Tsetse. TCP's Retransmission Timer and the Minimum RTO. 2012.

[27] J. Martin and A. Nilsson. On service level agreements for IP networks. In *INFOCOM*. IEEE, 2002.

[28] C. Olah. Understanding LSTM Networks, 2015.

[29] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review*, 1998.

[30] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's retransmission timer. Technical report, 2011.

[31] H. Tang and J. Glass. On Training Recurrent Networks with Truncated Backpropagation Through Time in Speech Recognition. pages 48–55. IEEE, 2018.

[32] K. Weiss, T. M. Khoshgoftaar, and D. Wang. A survey of transfer learning. *Journal of Big Data*, 2016.