Performant TCP over BLE

Jiamei lv, Wei Dong, Yi Gao, and Chun Chen
College of Computer Science, Zhejiang University,
Alibaba-Zhejiang University Joint Institute of Frontier Technologies.
Email: {lvjm, dongw}@zju.edu.cn, qhgaoyi@gmail.com, chenc@cs.zju.edu.cn

2022 IEEE 30th International Conference on Network Protocols (ICNP) | 978-1-6654-8234-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICNP55882.2022.9940406

***Abstract*—Bluetooth Low Energy (BLE) has gained large popularity as an important infrastructure of Internet of Things (IoT). Recently, researchers have integrated the TCP/IP stack with the BLE stack for interoperability, supporting more upper-layer protocols and applications. However, these works gain extremely low TCP goodput due to connection event inefficiency when TCP cooperates with BLE. How to improve the performance is an urgent problem. This paper proposes TCPle, a performant TCP-over-BLE stack that presents a novel adaption layer design bridging the gap between TCP and BLE without violating their specifications. TCPle improves the efficiency of connection events significantly by two fundamental mechanisms: connection event length adaption and connection event maintenance which correspond to the two root causes of low goodput. The connection event length adaption mechanism predicts the data size to send based on an online learning method and updates the connection event capacity adaptively. This mechanism avoids the long waiting time of ACK to back to the TCP sender. The connection event maintenance mechanism prefetches data packets to maintain the connection event. This mechanism avoids the long waiting time of data packets when out of the sender after the ACK is back. We implemented TCPle on nRF52840 DK with RIOT OS based on lwIP stack and NimBLE stack and conducted extensive experiments to evaluate its performance. Results show that TCPle 1) is lightweight and well-suited for resource-constrained IoT devices; 2) improves TCP goodput by up to 101.6% compared with other existing TCP-over-BLE stacks.**

I. Introduction

Bluetooth Low Power (BLE) is a low-power wireless technology that can be used over a short distance to enable devices to communicate. Since being defined in Bluetooth specification 4.0 [1] in 2010, BLE has gained considerable popularity and implemented in a broad set of IoT fields such as health [2], home automation [3], smart industry [4].

As an essential infrastructure of the emerging IoT, most BLE devices are still hidden behind the multi-radio gateways [5] and connect to the Internet relying on the gateway conversion. With the proposal of "Web of Things" [6], the need for communication between smart things and web-based applications behooves us to re-examine the transport question over BLE. Intuitively, TCP/IP-over-BLE stack has the following benefits: 1) Interoperability. A TCP/IP stack helps BLE to be interoperable with traditional TCP/IP networks. Using TCP critically simplifies IoT gateway design; 2) Protocol support. A number of widely-used IoT application protocols, such as MQTT [7] and ZeroMQ [8] .etc, are built upon TCP.

There already exist some attempts that build TCP/IP over BLE. Spörk et al. design BLEach [9], an open-source IPv6-over-BLE stack based on the uIP [10]. Besides, a number of embedded TCP/IP stack (e.g., uIP [10], GNRC [11], BLIP [12]) are proposed to enable diverse applications integrating embedded IoT devices. However, these works focus on the architecture design or the optimization of BLE performance itself and are failed to explore the performance of integrated stack when TCP cooperates with BLE.

We measure the TCP goodput of existing TCP-over-BLE stacks, finding that the network performance degraded dramatically compared with TCP over other wireless technologies. The TCP goodput over BLE only achieves 34.4% of the ideal TCP goodput, much lower than the ratio of TCP-over-802.11b, TCP-over-802.11ac, and TCP-over-802.15.4 stack (Detailed in Section II). After careful analysis, we attribute the poor performance to *connection event inefficiency*. BLE devices communicate with each other in non-overlapping fixed-length slots called connection events. If there are no more data to send in the BLE stack buffer or the size of data sent reaches the connection capacity, BLE devices close the radio and wait for the next connection event to start. If packets arrive at the BLE stack when the radio is off, they may wait for a long time. In the worst case, packets wait for a connection interval which is defined as the time between the start of two consecutive connection events.

Generally, there are the following two cases that lead to the connection event inefficiency and harm the network performance severely when transmitting TCP over BLE: 1) The actual connection event length is much smaller than the connection interval. When their difference is larger than the Round Trip Time (RTT) of TCP, TCP ACKs have to wait in the BLE stack buffer each time. 2) The connection event where the TCP receiver sends ACK back ends after the transmission completes, leading to the subsequent TCP data packets having to wait in the BLE stack buffer for nearly a connection interval. (Detailed in Section II).

Based on the above analysis, we propose TCPle, a novel performant TCP-over-BLE stack design that 1) enables BLE nodes to interoperable with TCP without violating the specification of BLE and TCP; 2) outperforms existing TCP-over-BLE stacks with higher TCP goodput. TCPle adds an adaption layer between the network layer of the TCP/IP stack and the L2CAP layer of the BLE stack, making the connection-based link layer design transparent to the TCP layer. TCPle can transmit TCP packets smoothly upon BLE, just like they are transmitted on IEEE 802.15.4 or Ethernet.

TCPle uses two fundamental mechanisms to achieve this:

978-1-6654-8234-9/22/$31.00 ©2022 IEEE

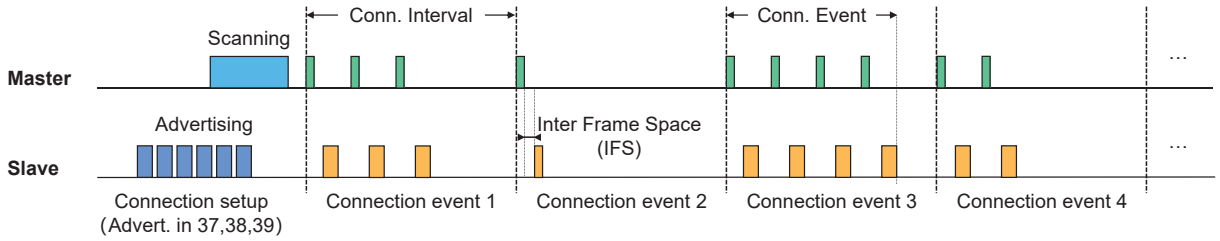

Fig. 1: BLE connection between the slave and master. 37, 38, and 39 are the indexes of the primary advertising channels.

connection event length adaption to deal with the long waiting time of TCP ACK packets when transmitted back to the TCP sender and *connection event maintenance* to deal with the long waiting time of TCP data packets when transmitted out from the TCP sender. The connection event length adaption mechanism predicts the size of data to be sent in the following connection event using an online learning method. Based on the data size, TCPle updates the connection interval adaptively, reducing the waiting time of ACK. The connection event maintenance mechanism keeps connection events. When the TCP sender is processing ACK packets, the connection event maintenance mechanism prefetches data packets to keep the connection event. Once ACK packets are processed, the TCP sender can send the data packets out quickly. Besides, we design an RTT estimation module in TCPle, which recovers the actual RTT of each TCP packet, helping TCP protocol to perceive the network conditions.

We implement TCPle on nRF52840 DK [13] with RIOT OS [14] based on lwIP [15], one of the most popular embedded TCP/IP stacks, and NimBle [16], an open-source BLE stack. We conduct extensive experiments to evaluate the performance of TCPle. Results show that: 1) TCPle is lightweight and well-suited for resource-constrained IoT devices; 2) TCPle achieves 363 kbps, which is within 75.1% of the upper bound and substantially higher than prior work by up to 101.6%.

In summary, this paper makes the following contributions:

- We systematically study the performance of TCP over BLE and point out that connection event inefficiency is the main cause of poor performance.
- We propose TCPle, a novel performant TCP-over-BLE stack design which improves the efficiency of connection events by connection event length adaption mechanism and connection event maintenance mechanism. TCPle can improve the performance of TCP over BLE significantly without violating the BLE specification.
- We integrate TCPle into RIOT OS and evaluate its performance by extensive experiments. Results show that TCPle outperforms existing TCP-over-BLE stacks by 101.6%.

II. Primer of BLE

To improve reliability, BLE nodes exchange their data based on connection. As shown in figure 1, in the time domain, a connection is split into fixed-size time slices called connection events. The interval between two consecutive events denoted as connection interval is fixed. Each connection event follows a strict packet flow. At the beginning of each connection event, the master sends a packet to the slave. After receiving the packet, the slave replies with a packet after a fixed amount of time called the inter-frame spacing (IFS). This packet exchange is done at least once in every connection event. If none of the peers has any data to transmit, they exchange packets with empty payloads to keep the connection alive (See connection event 2 in Fig. 1). If more data is ready for transfer, they transmit packets in the same connection event until the maximum connection event length (determined by the connection capacity) is reached or other BLE activities need to access the radio. The remaining data will be transmitted in the next connection event (See Connection event 3 and 4 in Fig. 1). It is worth emphasizing that the Bluetooth standard defines a set of link-layer control mechanisms that can be used to update the connection parameters (e.g., connection interval) on-the-fly after a connection is opened.

BLE provides a reliable link. The connection-based mode automatically handles packet acknowledgments (ACKs) and link-layer flow control using a 1-bit sequence number and 1-bit ACK field in the frame header. Any lost or CRC-error packet is retransmitted until a valid acknowledgment has been received.

III. Motivation and Overview

With the popularity of the concept that IoT devices should be seamless to use, many organizations, and researchers are trying to implement the ecosystem of IP-based protocols over IoT devices [9], [17], [18]. As an important part of the ecosystem, a lot of application protocols are implemented on TCP, e.g., MQTT, AMQP, etc. At the meanwhile, BLE is one of the most promising wireless protocols for IoT. Therefore, studying TCP over BLE is very significant to bring value to BLE moving forward. In this section, we present the experimental evaluation that guides our design. Specifically, we explore the upper bound of TCP over BLE on single-hop goodput first. Then we conduct a series of experiments to show the giant gap between the actual and ideal goodput. Finally, we explore the root cause of the poor performance of running TCP on BLE.

A. Upper bound on single-hop goodput

We consider TCP goodput between two nodes over the BLE link over a single hop without any border router. Here we consider a data uplink scenario, i.e, the TCP sender is the BLE slave and the TCP receiver is the BLE master. The ideal TCP packet transmission over BLE is that: in a connection event, the slave sends BLE packets carrying the transport layer data to the master continuously until the current connection event

TABLE I: The overhead sources of TCP when it upon BLE

Sources	L4 Hd	L3 Hd	L2 Hd	L4 ACKs	Radio	Link
Inverse Gput. (s/Mb)	0.064	0.040	0.028	0.776	0.375	1.00

TABLE II: Header overhead with 6LoWPAN fragmentation

Header	BLE	6LowPAN	IPv6	TCP	Total
1_{st} Frame	14 B	5B	2-28 B	20-44 B	41-91B
n_{th} Frame	14 B	5–12 B	0 B	0B	19-26 B

ends. In the next connection event, the master sends the BLE packets carrying TCP ACK to the slave.

Table I lists the various sources of overhead that limit TCP's performance when it is upon BLE, based on the ideal scenario described above. We use the inverse goodput proposed in [18] to present the overhead of different sources. Lower inverse goodput represents less overhead. In the table, link overhead refers to the link capacity. In other words, when there are no other sources of overhead, TCP throughput is 1/1=1Mbps. Radio overhead includes SPI transfer to/from the radio (i.e., packet copying), inter-frame space, and link-layer ACKs. L2 Hdr, L3 Hdr, and L4 Hdr are the header overhead whose size has been shown in Table II. Overall, we estimated a 438 kb/s upper bound on goodput.

B. Experimental study

1) Prototype and methodology: In order to measure the goodput, firstly we need a TCP-over-BLE prototype. There already exists some IPv6-over-BLE works, e.g., BLEach [9], IPv6-over-BLE example in RIOT OS [14]. However, these works are all based on the embedded TCP/IP stacks which omit some standard features of TCP. For example, uIP which BLEach is built on allows only a single outstanding (un-ACKed) TCP segment per connection, rather than a sliding window of in-flight data. This mechanism reduces the channel utilization and severely hurts the goodput. In order to ensure the results are due to the TCP protocol, not the feature omission of the TCP/IP implementation we used, we compare the feature set of different embedded TCP/IP stacks (see Table III) and finally choose lwIP [15] to ground our study. lwIP provides a full-scale TCP implementation and only occupies tens of KB RAM and around 40 KB of ROM.

As to the BLE stack, we choose Apache NimBLE [16], an open-source BLE stack that completely replaces the proprietary Soft Device on Nordic chipsets. NimBLE stack implements full-fledged BLE connections and complies with Bluetooth Core Specification 5.0 with low memory requirements (4.5 kB of RAM, 69 kB of flash).

We implement the prototype on nRF52840 DK based on RIOT OS and measure the goodput over a link path. In the experiments, the TCP and BLE parameters are all set to the default [15], [16] if not specified. The TCP sender is set to the BLE slave and the TCP receiver is set to the BLE master.

Through the measurement, we try to answer two questions:

- How is the performance of naively running TCP over BLE?
- If the performance is poor, can we improve it closer to the upper bound by directly borrowing techniques designed for other radios?

TABLE III: Comparison of core features among different embedded TCP stacks.

	uIP	GNRC	BLIP	FreeTOS	lwIP
Multi segmentation	×	×	✓	✓	✓
Flow control	×	×	✓	×	✓
Congestion control	×	✓	×	×	✓
RTT Estimation	×	✓	×	✓	✓
Keep-alive	×	×	×	✓	✓
TCP Timestamps	×	×	×	✓	✓
SACK	×	×	×	✓	✓
Delayed ACK	×	×	×	✓	✓

TABLE IV: The comprasion of TCP goodput on different wireless technologies.

	802.11b [19]	802.11ac [19]	802.15.4 [18]	BLE
Physical Rate (Mbps)	11	866.7	0.25	1
Ideal TCP Gput. (Mbps)	7	590	0.095	0.438
Actual TCP Gput. (Mbps)	6	556	0.075	0.151
Ratio	85.7%	94.2%	78.9%	**34.4%**

2) The goodput of naively running TCP over BLE: We measure the performance of naively running TCP over BLE. The offered load of the BLE end node is 600 kb/s. As a comparison, we also investigate the performance of running TCP over other wireless technologies. It is worth noting that the performance of TCP over IEEE 802.15.4 is based on TCPlp [18]. It is a state-of-art TCP/IP stack for IEEE 802.15.4 which proposes several mechanisms to resolve the poor TCP performance in 802.15.4.

The results are shown in Table IV. Compared with IEEE 802.11, IEEE 802.11b, or IEEE 802.15.4, the goodput of TCP upon BLE only achieves 34.4% of the upper bound of TCP goodput, which is much smaller than that of IEEE 802.11b (85.7%), IEEE 802.11ac (94.2%), and IEEE 802.15.4 (78.9%).

3) Goodput of running improved TCP over BLE: TCPlp proposes several performance-enhancing mechanisms (e.g., using an atypical Maximum Segment Size (MSS) equal to 5 frames to reduce the header overhead), which are also applicable to BLE. We port the TCPlp to BLE directly (called TCPlp-ble) and measure its performance to verify if these mechanisms can increase the goodput.

Fig. 2 shows the results of transmitting 600-byte data with different connection event lengths. Fig. 3 shows the goodput when transmitting data of different size. The connection interval is set to 50 ms. Compared with running TCP over BLE naively, TCPlp-ble increases the goodput significantly. This is attributed to its large MSS setting which can decrease the amount of data transmitted per connection event. For example, 1 kB data is sent in one TCP segment with TCPlp-ble whose MSS is 1,000 and is sent in one connection event. But, with our prototype TCP/IP stack, it is divided into five TCP segments which take two connection events to send.

We also measure the maximum goodput of TCPlp-ble. We set a big enough buffer size for sender and receiver and offer 800 kb/s flow. Results show that TCPlp-ble reaches 180.2

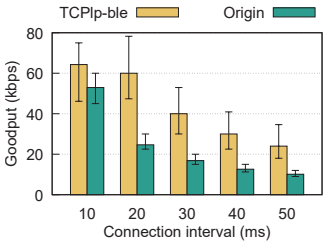

Fig. 2: The goodput when transmitting 600 bytes with different connection intervals.

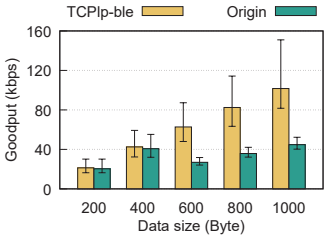

Fig. 3: The goodput when transmitting data of different size.

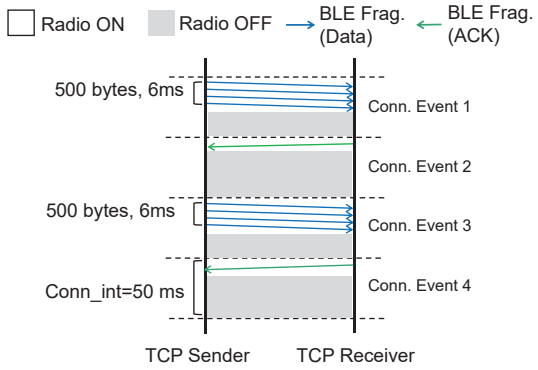

Fig. 4: The timing diagrams when TCP sender sends 1,000 bytes of data to the TCP server.

kbps, increasing the goodput by 19.3% compared with our prototype stack. The performance gain comes from the smaller header overhead. Though an improvement, it is only 41% of the upper bound. In summary, existing work can improve performance significantly when transmitting small-scale data. But its performance is still unsatisfactory compared with the upper bound goodput.

C. The main causes of the poor performance

In order to find out the root reasons for the poor performance, we depict a data transmission through a timing diagram as shown in Fig. 4 . The shaded areas indicate that two nodes turn off their radio while the white areas indicate that their radio is on. The blue arrows present the transmission of BLE fragmentation carrying data from the upper layer. The green arrows present the BLE packets carrying TCP ACKs. For simplicity, we omitted some empty BLE packets related to the maintenance of the BLE connection.

The transmission takes four connection events. In the first connection event, the TCP sender sends a TCP segment whose payload size is equal to the send window of TCP, taking about 6 ms. And then two BLE nodes turn off their radio and sleep for about 44 ms since there is no more data in BLE send buffer. During radio off, the TCP receiver processes the TCP segment received, generates the corresponding ACK, and pushes it to the send buffer. When the second connection event starts, the TCP receiver sends the prepared ACK to the sender. The sender needs to schedule the pending data according to ACK, its BLE stack buffer is still empty when it receives the ACK. Therefore, the second connection event ends with only one TCP ACK transmission. The sender sends the prepared data to the receiver in the third connection event and receives the corresponding ACK in the fourth connection event.

Overall, it takes about 152 ms from the TCP layer to send the data to receive the last ACK. Up to 92% of the time is wasted on waiting for the start of the connection event, leading to inefficient connection events and dramatically harming the performance.

Based on the timing diagram, we summarize two reasons for the connection event inefficiency:

- The mismatching between the maximum connection event length and actual connection event length (e.g., the connection event 1 and 3 in Fig. 4).
- Connection event ending prematurely due to the empty sending buffer of BLE stack (e.g., the connection event 2 and 4 in Fig. 4).

D. Overview of TCPle

Based on the above two root reasons, we propose TCPle, a novel TCP-over-BLE design, aiming to improve the goodput of BLE nodes equipped with TCP protocol.

When designing TCPle, we strive to achieve the following three goals:

- **Specification-compliant:** First of all, TCPle should follow both the specifications of TCP and BLE.
- **Lightweight:** Given that IoT devices are always resource-constrained, TCPle should have low memory usage.
- **Adaptive to network variation:** BLE devices are widely used in various scenarios. TCPle should provide effective TCP service for BLE devices adaptively.

The architecture of TCPle is shown in Fig. 5. We add an additional adaptation layer, decoupling the critical design of TCPle from the TCP/IP stack part and BLE stack. Based on this architecture, TCPle can port to various TCP designs and BLE chipsets with little overhead. TCPle uses two key mechanisms to deal with the root causes of poor performance mentioned above:

- **Connection event length adaption mechanism** for the gap between the maximum connection event length and actual connection event length.
- **Connection event maintenance mechanism** for the connection event ended prematurely.

We detail these two mechanisms in the following sections.

IV. Connection Event Length Adaptation Mechanism

In this section, we present the connection event length adaptation mechanism which deals with the mismatching between the maximum and actual connection event length. As shown in Fig. 5, this module is located at the BLE master (i.e., the gateway in most realistic scenarios). There are three reasons for such design: 1) The connection event length adaption is always initiated by the master according to the BLE

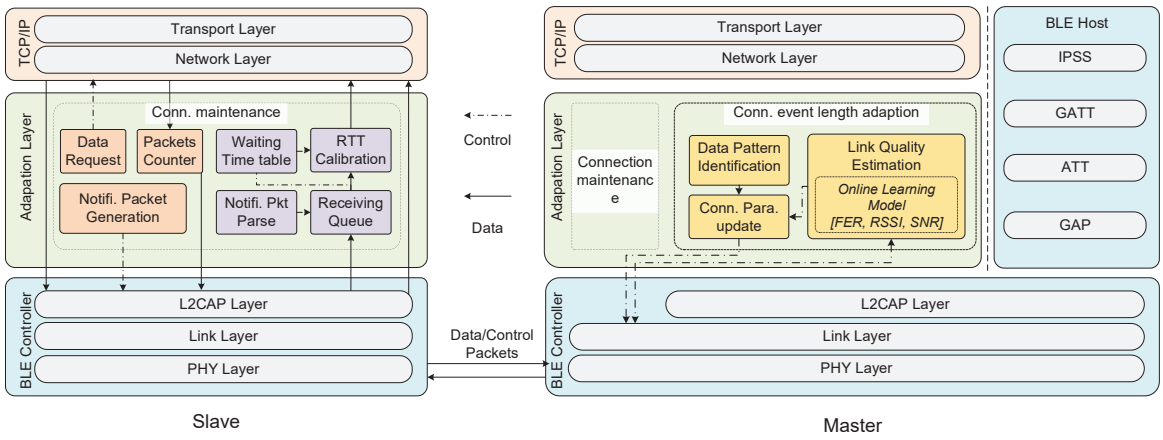

Fig. 5: Architecture of TCPle. Connection maintenance is necessary in both slave and master while connection event length adaption is only needed in master.

specification [20]; 2) Compared with the slave, the masters are usually more powerful in terms of computing, energy supply, .etc, which is capable to host the function of predicting data length (detailed in Section IV-B). 3) The masters are capable to get more data to predict the data length in multi-connection scenarios. Next, we detail the mechanism.

First, we model the ACK wait time in the BLE buffer. According to the description in Section II, if the ACK is not pushed into the link layer buffer before the end of the current event, it can only be transmitted on the next connection event. Its waiting time T_{ack} in the link layer buffer can be presented as:

$$T_{\text{wait}}^{\text{ack}} = \left(\left\lfloor \frac{T_{\text{trans}}^{\text{data}} + T_{\text{rtt}}}{T_{\text{ci}}} \right\rfloor + 1 \right) \cdot T_{\text{ci}} - T_{\text{trans}}^{\text{data}} - T_{\text{rtt}}, \tag{1}$$

where $T_{\text{trans}}^{\text{data}}$ is the transmission time of data in link layer, T_{rtt} is the round-trip time from the BLE gateway to the server, and T_{ci} is connection interval. In an ideal scenario, the ACK packets are just ready when the next connection event starts. In other words, connection interval should approach $(T_{\text{trans}}^{\text{data}} + T_{\text{rtt}})/n$ where $n \in \mathbb{N}^+$. Since a smaller connection interval introduces more control overhead, we set $n = 1$ in this paper. Now, the problem has changed to how to get $T_{\text{trans}}^{\text{data}}$ and T_{rtt}. Generally, $T_{\text{trans}}^{\text{data}}$ is determined by the size of data sent in the current connection event and can be given by $\frac{P_s + P_r}{R}$. P_{s} is the size of data packets in the BLE stack buffer, P_{r} is the size of retransmitted packets due to the bad link quality and R is the payload size transmitted per second. P_{s} is related to the size of the send window, the capacity of the BLE buffer, .etc, and P_{r} is related to the link quality between the BLE slave and master. We detail the approach of get P_{s} and P_{r} in next section.

Compared with $T_{\text{trans}}^{\text{data}}$ which is correlated to highly dynamic link quality, T_{rtt} is relatively static and can be estimated from

$$T_{\text{rtt}} = T_{\text{RTT}} - T_{\text{trans}}^{\text{data}} - T_{\text{trans}}^{\text{ack}}. \tag{2}$$

T_{RTT} is the round trip time of data packets in terms of BLE senders. We introduce how to estimate RTT over BLE in Section V-A. $T_{\text{trans}}^{\text{data}}$ and $T_{\text{trans}}^{\text{ack}}$ are the transmission time of data and ACK packets respectively.

It is worth noting that when connection interval is smaller than actual sum of $T_{\text{trans}}^{\text{data}}$ and T_{rtt}, the ACK packet can not be sent in next connection event. In the worst case, the ACK packet has to wait for a connection interval. In order to avoid a long waiting time due to short connection interval caused by estimation bias, TCPle set the connection interval as $\alpha(T_{\text{trans}}^{\text{data}} + T_{\text{rtt}})$, after estimating $T_{\text{trans}}^{\text{data}}$ and T_{rtt}. α is a coefficient that is larger than 1 (1.25 in this paper).

TCPle uses connection parameter update which is naturally supported in BLE specification to update the connection interval. The master periodically estimates $T_{\text{trans}}^{\text{data}}$ and T_{rtt} for each slave. Since too small or too large connection intervals all lead to long wait time, master updates the connection interval if $\frac{T_{trans}^{\text{data}} + T_{\text{rtt}}}{T'_{\text{itval}}} < 0.8$ or $\frac{T_{\text{trans}}^{\text{data}} + T_{\text{rtt}}}{T'_{\text{itval}}} > 0.9$, where T'_{itval} is the current connection interval.

Next, we introduce how to estimate P_{s} and P_{r}.

A. Size of data to be sent

The traffic pattern, the sending window size of TCP S_{snd}, Bluetooth software stack that can be used S_{ble}, .etc, together determine the amount of data to be sent P_s in the next connection event. Mathematically, P_s can be denoted as

$$P_{\text{s}} = \min(S_{\text{data}}, S_{\text{snd}}, S_{\text{ble}}), \tag{3}$$

where S_{data} is the data remaining to send which is directly related to data patterns.

For periodic data packets with a small payload (e.g., MQTT packet carrying light sensor data), the size of data to be sent is mainly limited by S_{data}. For continuously streaming data, the bottleneck is the sending window of TCP and the Bluetooth software stack. For some older devices which only support one packet per connection interval due to small stack size, S_{ble} is the bottleneck.

The TCP sending window size is dynamic to some extent, compared with the traffic pattern and BLE software stack size. Take TCP Reno as an example. The sending window size of the next round trip may be double (in the slow start phase) or half (when detecting network congestion) of the current window size. TCPle estimates a maximum sending window

considering the phase of TCP protocol to avoid the maximum connection event length being too small to send all data out.

B. Size of the retransmission data

BLE provides reliable transmission in link layer. Given the frame error rate (FER) e and the bytes of data waiting to be transmitted P_s, the BLE device has to transmit $\frac{P_s}{1-e}$ bytes to successfully transmit all data. In other words, P_r can be denoted as $\frac{e}{1-e}P_s$. Since P_s represents the total amount of data in a round, e here represents the quality of the link over a short period of time, rather than the loss rate of individual packets. The problem is how to predict e in BLE.

As the physical layer information is directly related to the channel's quality, TCPle estimates link quality based on PHY information intuitively. Since the dynamics of the BLE link are hard to be captured by a single rigid model, TCPle uses a stochastic gradient descent (SGD) online learning algorithm [21] to train a logistic regression classifier (LR) such that the model can adapt to the changing link conditions.

The input of the model is the historical information available from past M packets ($M = 10$ in this paper). The information includes the Received Signal Strength Indicator (RSSI), Signal to Noise Ratio (SNR) and frame error rate (FER) when the packet is received. FER is calculated from the WMEWMA output. RSSI is captured in every BLE packet. SNR is obtained by the method in [22]. In order to simplify the algorithm, the output of the model is the probability that the link quality is worse than the threshold θ. In this paper, θ is set to 22.5% (i.e., 25% $\times$ 0.9) according to the fault tolerance ratio mentioned before.

Formally speaking, assume $X =< X1...X_{|M|} >$ represents the input vector where $X_i = [FER_i, RSSI_i, SNR_i]$, Y is the binary variable denoting whether $FER > \theta$, the logistic regression classifier can be expressed as:

$$P(Y = 1|X) = \frac{1}{1+\exp(-f(X))} \tag{4}$$

$$P(Y = 0|X) = \frac{\exp(-f(X))}{1+\exp(-f(X))} \tag{5}$$

where $f(X) = \beta_0 + \sum_{i=0}^{n} \beta_i X_i$, β is a vector of the weight parameters to be estimated.

Given a training set of N samples, $(X^1, Y^1)..., (X^N, Y^N)$, TCPle trains the logistic regression classifier by maximizing the log of the conditional likelihood, which is the sum of the log-likelihood for each training example:

$$\begin{aligned} l(\beta) = Y^l \sum_{l=1}^{N} \log P(Y = 1|X^l, \beta) \\ +(1-Y^l)\log(P(Y^l = 0|X^l, \beta)) \end{aligned} \tag{6}$$

To maximize the log likelihood, TCPle uses the gradient, which is the partial derivative of the log conditional likelihood. The ith component of the gradient vector is

$$\frac{\partial}{\partial \beta_i} l(\beta) = \sum_{l=1}^{N} (Y^l - \widehat{P}(Y^l = 1|X_i^l, \beta))X_i^l \tag{7}$$

where $P(Y^l = 1|X_i^l, \beta)$ is the logistic regression prediction using Equation (4), (5) and the wight β.

Instead of batch training which optimizes the cost function defined on all the training samples, TCPle uses SGD, an online algorithm that operates by repetitively drawing a fresh random sample and adjusting the weights based on this single sample only. It performs weight updates on the basis of the gradient of a single sample X^l,Y^l:

$$\beta_i \leftarrow \beta_i + \lambda \triangle \beta_i^l \tag{8}$$

where λ is the learning rate that determines the step size and how fast the gradient descent converges. TCPle uses a classical adaptive learning rate algorithm s-ALAP [23] whose metalearning rate is set to 0.8 to update the learning rate. $\triangle \beta_i^l$ is the gradient of the l_{th} sample:

$$\triangle \beta_i^l = (Y^l - \widehat{P}(Y^l = 1|X_i^l, \beta))X_i^l \tag{9}$$

The model is running on the master (an RPI in our evaluation) and estimates the size of retransmission data periodically (two connection events in our paper). It is worth mentioning that in practice, TCPle has pre-trained the model based on the information of 1,000 BLE packets (including control packets and data packets whose payload size is 100 bytes) offline so that the model converges faster as it online works. In order that the data for pre-training covers a wide range of situations, we change the distance between the slave and master when collecting packets.

V. Connection maintenance mechanism

In order to save energy, BLE devices turn off their radio when they have no data to exchange in the BLE stack buffer. In other words, the BLE controller is ignorant of the data waiting in the TCP layer unless they are pushed into the BLE software stack. Therefore, an intuitive idea to keep the current BLE connection event is to let the TCP sender radio on and keep sending some packets while processing the ACK received.

In order to further improve the goodput of TCP, TCPle transmits the data waiting in the TCP layer queue rather than meaningless packets to maintain the event. The main procedure is as follows.

In TCP sender, when waiting for the ACK packets, TCPle generates a data request to prefetch data packets from the TCP layer and pushes them to the BLE link layer buffer. As soon as receiving an ACK in the next connection event, the BLE controller sends these packets to maintain the connection. According to our experimental measurements, it usually takes about 2-3 ms for the TCP sender to process the received ACK packet. If the remaining data is not enough for the sender to send for 2-3 ms, TCPle generates some empty packets, which will be discarded in the adaptation layer of the receiver. The sender keeps a counter that records the number of packets sent to the receiver in advance. Suppose the number is larger than the upper bound of sending window of the next round trip (e.g., $2 \cdot snd$ when the slow start of TCP Reno), TCPle will send all empty packets to maintain the connection event. The purpose is to relieve the storage pressure of the TCP receiver.

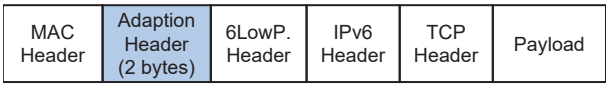

Fig. 6: The new structure of ACK packet. The blue filed is the new added adaptation layer header, containing the waiting time of the packet.

TABLE V: Waiting time table recording the time that the data packet waits in the buffer of the adaptation layer

IP Address	Seqno	Waiting Time (ms)	Time of entry
2001:250:...:4310	79	23	36734

When the ACK is processed, if data sent in advance is greater than or equal to the new sending window snd', TCPle stops prefetching data. If it is less than snd', TCPle continues to send data until the amount of data sent reaches snd' or no more data to send. When finishing sending data, the TCP sender sends a notification packet containing snd' to the TCP receiver. Meanwhile, the counter minus snd'.

In TCP receiver, TCPle stores the data packets in the receiving buffer in the adaptation layer. When receiving the notification packet, TCPle sends snd' packets from the buffer to the upper layer and forwards them to the destination.

A. RTT calibration

Round trip time (RTT) is an essential indicator of the network condition, having been widely used in various TCP schemes [24]–[27]. However, when TCP works upon BLE, the actual RTT is "hidden" due to the connection-based communication of BLE and data prefetching. Specifically, the ACK packets are likely to wait for a while at the TCP receiver's link layer for the start of the connection event. The waiting time is random and unpredictable. Besides, some packets prefetched may wait for at least one RTT in the receiver's receiving buffer located at the adaption layer if the prefetched data size is larger than the new sending window size. Thus TCP sender is hard to judge the root cause of the RTT increment— lousy network condition or long waiting time at peer's link layer or data packets transmitted in advance— when TCP over BLE.

In order to solve the above problem, TCPle adds a submodule "RTT calibration". In the TCP receiver, TCPle builds a waiting time table (as Table V) to record the duration T_{wdata} of each packet from entering the buffer to leaving. For the outgoing packets, TCPle first judges whether it is an ACK packet using cross-layer information. If it is, TCPle estimates the waiting time T_{wack} based on the state of its BLE controller. Then TCPle traverses the waiting time table and finds out the waiting time of the data packet corresponding to the ACK according to the "ACK" fields in the packet's header. Once found, TCPle gets the data and deletes the item from the table to save the resource. Those items that stay in the table for longer than the threshold ($2.5 \cdot RTO$ in this paper) will be deleted as well. TCPle calculates the total waiting time T_{wtotal} by $T_{\text{wdata}} + T_{\text{wack}}$ and fills it into the adaption layer header. The new data packet structure is shown in Figure 6.

At the TCP sender, the adaption layer extracts the total waiting time for each incoming ACK packet and then passes it to the upper layer with the data packet. The TCP layer obtains the real RTT RTT_r by:

$$RTT_{\text{r}} = T_{\text{s}} - T_{\text{r}} - T_{\text{wtotal}} \tag{10}$$

where T_{s} is the timestamp that the data packet was sent and T_{r} is the timestamp the ACK is received.

VI. Evaluation

A. Implementation

Hardware and OS. We implement TCPle on nRF52840 DK, which features an ARM Cortex TM-M4 with 512KB RAM and 1MB flash. We also build a BLE gateway comprising of a Raspberry Pi (RPI) 3 connected with the nRF52840 DK. The adaption layer and the above layer are running on the RPI while the BLE stack is running on the nRF52840 DK. We originally implement TCPle on RIOT OS which provides basic integration of NimBLE and lwIP. We modify Nimble to get the PHY and MAC layer information and modify lwIP to enable the adaption layer to prefetch data and get the sending window size.

Maximum segment size (MSS) setting. In traditional networks, it is customary to set the Maximum Segment Size (MSS) to the link MTU (or path MTU) minus the size of the TCP/IP headers. However, BLE frames (251 bytes) are much smaller than frames in traditional networks (1500 bytes). The TCP/IP headers consume nearly half of the frame's available MTU, incurring large header overhead. Like TCPlp, we also choose an MSS larger than the link MTU admits, relying on fragmentation at the 6LowPAN layers to decrease header overhead. However, using an excessively large MSS decreases reliability because the loss of one fragment results in the loss of an entire packet. Existing work [28] has identified this trade-off. We choose an MSS of about five frames in the evaluation, which is proved effective in most scenarios.

Setup. We compare TCPle with BLEach [9], TCPlp [18] (See Section III-B1) and Connected Home over IP (CHIP, now changed to Matter) [17]. CHIP is a new standard for cross-vendor networking of devices in the smart home, which is jointly proposed by Amazon, Apple, Google, Comcast, and the ZigBee Alliance at the end of 2019. The specification supports BLE device communication based on IP. It is built on lwIP as well. Several parameters may influence results, e.g., the initial size of the TCP sending window, the capacity of the packet queue, etc. When not specified, we set the common settings according to default values in specifications or stacks. Specifically, the initial size of the TCP sending window is set to $2 \cdot$ MSS. The capacity of the packet queue is $8 \cdot$ MSS. The congestion control algorithm is Reno. We disable the delayed ACK in TCP/IP stack. The size of the BLE stack buffer is 1,280 bytes. We record packet transmission and reception events with their timestamps for later analysis.

B. Overhead

1) Memory footprint: We quantify the memory footprint of TCPle on slave and master devices in terms of RAM and ROM usage. Table VI shows TCPle's ROM and RAM

TABLE VI: ROM and RAM usage (kB) of TCPle. “RAM_S” means static RAM usage and “RAM_D” means dynamic RAM usage.

		TCP/IP	Adap.	BLE	Total
TCP sender & BLE slave	ROM	40.766	9.703	75.278	125.747
	RAM_S	8.061	7.334	3.009	18.404
	RAM_D	6.296	2.214	2.019	10.529
TCP receiver & BLE master	ROM	40.766	15.692	75.278	131.736
	RAM_S	8.061	8.172	3.009	19.782
	RAM_D	4.016	4.022	3.215	11.253

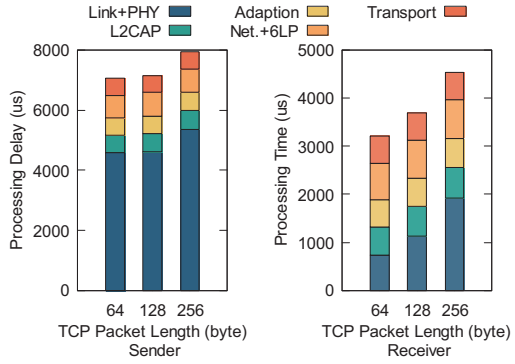

Fig. 7: Breakdown of TCPle’s processing time per layer when serving TCP transmissions of varying payload length.

usage. The ROM usage of TCPle on the slave and master is 125.747 KB 131.736 KB while the maximum RAM usage is 28.933 kB and 31.035 kB. The difference in memory usage between master and slave comes from the connection event length adaption mechanism. Compared to the original prototype stack, TCPle increases up to 15.692 kB ROM and 12.194 kB RAM. Although increased memory usage, TCPle is still very lightweight and well-suited for many resource-constrained embedded IoT devices.

2) Processing overhead: To evaluate the processing overhead, we measure the duration of TCP transmissions with different payload sizes from the slave to the master using and break down the time spent in each network layer. We average the values from 100 experiments. The results are shown in Fig. 7. The largest fraction of time is spent in the BLE controller performing the actual data transmission in the TCP sender. For example, the BLE link and physical layer take up to 64.9% of the total time when transmitting 64-byte packets. When transmitting 256 bytes TCP packets, due to the limitation of MTU, the packets are fragmented to send and the proportion increases to 67.6%. The main cause of the long processing time is that the packet waits for the connection event to start in the BLE stack buffer for a duration of time. According to the connection event length adaption algorithm, TCPle set the connection interval to 7.5 ms, the minimum value defined in the BLE specification. Since the data arrive at the BLE stack buffer randomly, they have to wait about $7.5/2 = 3.75$ ms inevitably. It is worth noting the new modules only introduce about 8% processing time. In the TCP receiver, the time spent in the adaption layer increases compared with the sender because of the overhead of the connection event length adaption mechanism which is only deployed in the BLE master. However, for the complete transfer process, the newly introduced processing time is insignificant and worthwhile, which will be proved in the below evaluation.

C. TCPle performance over a single hop

In this subsection, we evaluate TCPle’s performance over a single hop. The BLE slave acts as the TCP sender while the BLE master acts as the TCP receiver.

1) Maximum goodput: We increase the offered load from the TCP sender to reach the maximum goodput. We set a big enough buffer size to prevent that TCP exhibits “stop-and-wait” behavior due to the small flow window. The TCP sender sends data in a 50-seconds interval. We get the average goodput across all intervals.

The results are shown in Fig. 8. For a small offered load, the average goodput increases linearly. Once the offered flow exceeds a critical value, the average goodput no longer increases, in which case we believe we have reached the maximum goodput of running TCP over BLE on a single hop. The maximum goodput of TCPle is 363 kbps, reaching 75.1% of the upper bound (introduced in Section II). The difference from the upper bound is likely due to the network stack processing, fault-tolerance of TCPle, and other real-world inefficiencies.

Compared with TCPlp-BLE, CHIP, and BLEach, TCPle reaches about 2.0×, 2.3×, and 14.5× goodput improvement, respectively. The reasons are three-fold: 1) TCPle, CHIP, and TCPlp-ble support multiple segments and pipeline data packets, increasing the throughput; 2) When transmitting data, TCPle estimates the connection event length and decreases the connection interval to 7.5 ms, which is the minimum, reducing the average waiting time to 3.75 ms. The connection interval of TCPle and BLEach is always 50 ms, introducing an average waiting time of 25 ms; 3) BLEach, TCPlp-ble, and CHIP close their radio for nearly 50 ms once receiving ACK back. On the contrary, TCPle makes the sender send data, improving connection event efficiency.

2) Resilience to wireless loss: In this subsection, we evaluate TCPle in terms of dynamic link quality. One BLE slave sends 1,000 bytes data to the master per 100 ms. We inject uniformly random BLE package loss at the master and calculate the average goodput of every transmission.

The result is shown in Fig. 9. As the loss rate increases, the goodput of TCPle drops up to 20.5% while the goodput of BLEach, TCPlp-ble, and CHIP drops 0.4%, 5.1%, and 4.5%. The relatively big decrease of TCPle comes from the connection event length adaption mechanism. If the BLE packets are lost or in error, the slave retransmits them. In this case, TCPle increases the connection interval to provide a bigger connection event capacity, which introduces more waiting time. As to TCPlp-ble and BLEach, since their connection event capacity is redundant initially, their goodput drops are relatively small. Though a greater decline, TCPle’s performance is always better than BLEach and TCPlp-ble due to the existence of the connection maintenance mechanism which fastens the transmission.

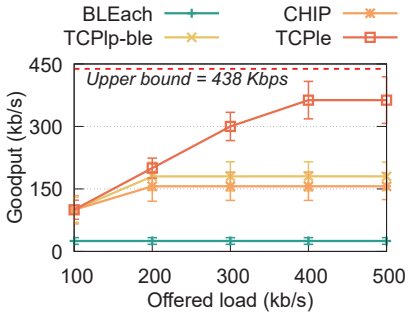

Fig. 8: The impact of offered load on the goodput.

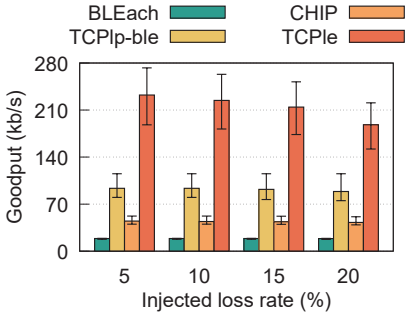

Fig. 9: The impact of injected wireless loss rate on the goodput.

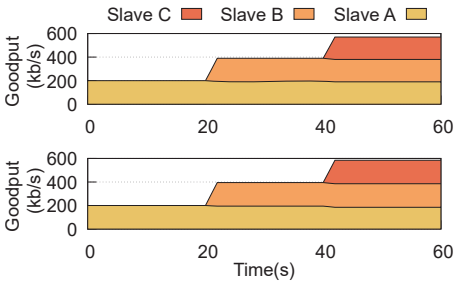

Fig. 10: The goodput when slaves join the BLE network.

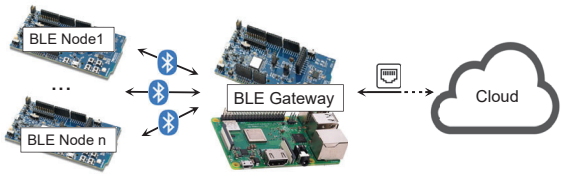

Fig. 11: A testbed of TCPle for real IoT cases.

3) Fair channel sharing: BLE is a star topology network. The master device "manages" the connection, and can be connected to multiple slaves. We run an exemplary BLE system consisting of a master and three slaves to evaluate TCPle's performance with a different number of slaves joining the network. We have two experimental setups. In Setup 1, three slaves are all implemented with TCPle. In Setup 2, Slave A runs TCPle, and Slave B and Slave C run naive BLE stack (i.e., NimBLE). Slave B and Slave C connect to the master in the 20s and 40s respectively. The offered flow of each slave is 200 kb/s. We calculate the goodput of each slave over time.

Fig. 10 shows the results. The upper figure is the result of the first setup while the lower is the second setup. Since the connection events of different slaves may conflict, the goodput is smaller than the offered flow when several BLE slaves coexist. As Fig. 10 illustrated, TCPle provides fairness between different flows. Specifically, Jain's Fairness Index (JFI) [1] of Setup 1 is 0.99 while JFI of Setup 2 is 0.98. This is because the delay caused by connection event conflict in the master is transparent to TCPle. TCPle will not adapt its connection parameters in this case and therefore promise its fairness to some extent when TCPle nodes coexist with legacy BLE nodes.

D. Real IoT use cases

In this section, we evaluate TCPle with two real IoT use cases. We build a testbed as shown in Fig. 11. Several nRF52840 DKs communicate with the BLE gateway, which is comprised of Raspberry Pi and nRf5840 DK through BLE. The gateway routes the TCP packets to the cloud by Ethernet.

1) Sense-and-send application scenario: We begin with a common sense-and-send paradigm, in which devices periodically collect sensor readings and send them upstream. For concreteness, we model our experiments on the deployment of noise sensors in the building. Noise sensors collect measurements, generate one 56-byte reading and send it to the cloud server with TCP per second. We record the RTT (Round Trip Time) of each sense-and-send transmission event. The results are shown in Fig. 12. By choosing an appropriate connection interval, TCPle reduces the waiting time significantly.

Lifetime is important for IoT devices. Therefore, in addition to RTT, we also measure the energy consumption of each slave in a data upload event. We modify NimBLE to record the fractions of time the radio is in receiving, transmitting, and idle mode and compute the energy consumption based on the current and voltage from the nRF52840 data sheet [29]. Table VII shows the energy consumption of the BLE end device per sense-and-send event. TCPle consumes about 16.1%, 13.8%, and 14.2% more energy than BLEach, TCPlp-ble, and CHIP. There are two reasons for the increased energy consumption: 1) TCPle set a smaller connection interval (i.e., 7.5 ms which is the minimum defined in BLE specification) than these mechanisms, which introduces higher control overhead, which is the main reason for higher energy consumption. 2) Longer transmission time due to longer packet length.

Obviously, there exists a trade-off between energy consumption and throughput. In general, a smaller connection interval means shorter waiting time while introducing more control overhead. We build a model to mathematically measure the trade-off and find that the throughput is more sensitive to the reduction of the connection interval compared with energy consumption. For example, compared with 5.2 ms, the energy consumption increases about 12.3% while the throughput decreases about 85.7% when the connection interval is 52.5 ms. For those applications that acquire low RTT/high throughput, it can benefit from TCPle at a relatively small energy cost. For those devices whose energy consumption is the bottleneck, fortunately, the BLE specification provides "slave latency" which that allows a slave not to transmit for several connection intervals and still maintain the connection. By setting the "slave latency", the energy consumption can be significantly reduced (about 14.3%) with little performance degradation.

2) OTA update: In the context of IoT applications, OTA enables users to communicate, update, reconfigure and manage devices without having to physically access them. It is common to perform a firmware update to a microcontroller

[1] $JFI = \frac{(\sum x_i)^2}{n\sum (x_i)^2}$, where n is the number of elements. The worst fairness is $1/n$, and the best is 1.

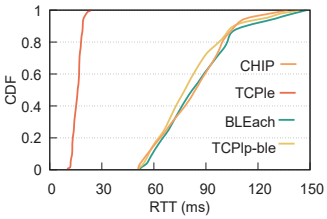

Fig. 12: The RTT of packets in sence-and-send application scenario.

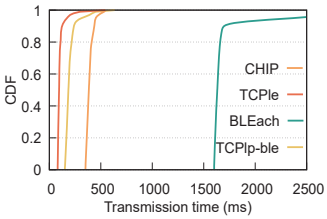

Fig. 13: The RTT of packets in OTA update application scenario.

TABLE VII: The average energy consumption of the BLE end node per sense-and-send event.

	Avg. energy consumption (uj)
BLEach	200.65±0.28
TCPlp-ble	210.22±0.23
CHIP	210.12±0.25
TCPle	240.62±0.25

device over BLE via an MQTT proxy [30]. In this section, we perform our evaluation with OTA update events. It's worth noting that we only focus on the OTA file transmission part. The BLE end node acts as the MQTT subscriber and the broker and publisher are all on the cloud. The publisher generates a 3k-byte OTA file and then pushes it to the broker. The broker sends it downstream to the BLE end node. We measure the transmission time of one OTA file transmission.

Fig. 13 shows the CDF of the transmission time. Since TCPle and TCPlp-ble adopt an unconventional large MSS (i.e., 5 frames), the OTA file is segmented into only three TCP segments. As a comparison, there are sixteen TCP segments when implemented with BLEach. Fewer TCP segments mean fewer connection events are required. Therefore, TCPle and TCPlp-ble are significantly performant compared to BLEach in this case. TCPle quickly transmits the second segment after receiving the ACK for the first TCP segment. So its transmission time is further shortened.

VII. Related work

BLE stacks. There have existed many proprietary BLE stacks lacking transport layer support. Opensource BLE support in TinyOS and Contiki is entirely missing or limited. Contiki only features transmissions of advertisement packets for the TI CC2650 radio and a closed-source BLE radio and L2CAP slave implementation for the nRF52 that does not support fragmentation of TCP packets. Zephyr [31] comes with stacks implementing full-fledged BLE connections. However, it supports TCP over BLE only on slave devices and cannot fragment large TCP packets, making it unsuitable for constrained IoT devices. Spork et al. propose BLEach [9] which is an open-source stack with support for IPv6 over BLE. It is built based on uIP without considering the impact of parameters of transport layers on transmission performance.

Unlike these work, TCPle is a TCP/IP stack supporting TCP over BLE. Considering the characteristics of BLE, TCPle adds an adaption layer to make the TCP/IP stacks and BLE stacks cooperate well.

Other relevant BLE research. In terms of BLE runtime adaptability, Gomez et al. [32] show that connection interval and slave latency impact BLE performance, suggesting that these parameters could be tuned to meet given application requirements. Similarly, Lee et al. [33] report on experiments showing that the connection interval affects the packet delivery rate. and they propose to tackle the BLE connection maintenance and energy consumption problems by controlling connection interval [34]. Kindt et al. [35] adapt the connection interval to traffic load for energy efficiency. Spork et al. [36] blacklist poor channels and select a physical mode in order to sustain high link-layer reliability, based on the link quality of the BLE connection. In other work [37], they show how BLE nodes can estimate and mitigate network loss and delay by dynamically adapting their BLE parameters. Park et al. [38] investigate a run-time scheduling system for BLE, and aim to guarantee the quality of service (QoS) of multiple connections through anchor point adjustment. Besides, some work focus on designing battery-driven [39] or energy-harvesting [40] BLE platforms. Petersen et al. [41] present a software platform and experiments to analyze multi-hop BLE network behavior. Some exploit BLE's connection-less mode for other services, such as neighbor discovery [42], localization [43], [44], and locality-based authorization [45].

These works focus on the BLE performance in different scenarios and applications. However, TCPle focuses on performance optimization when TCP is upon BLE, which is an unstudied area. We believe that TCPle, which improves the performance of TCP over BLE significantly, takes a solid step forward to achieving "Internet of Everything".

VIII. Conclusion

In this paper, we systematically study the performance when running TCP over BLE and point out that connection event inefficiency is the main cause of bad performance. Based on the analysis, we further propose TCPle, a novel TCP/IP stack for BLE which significantly improves the goodput of TCP when it is upon BLE. TCPle uses a connection event length adaption mechanism to solve mismatching between the maximum and actual connection event length and a connection event maintenance mechanism to solve the problem of connection event ending prematurely. We conduct extensive experiments to evaluate TCPle and results show that TCPle improves TCP goodput by up to 101.6% compared with other existing TCP-over-BLE stacks.

Acknowledgements

We sincerely thank our shepherd, John C. S. Lui, and anonymous reviewers for their insightful comments. This work is supported by NSFC under grant no. 62072396, Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under grant no. LR19F020001, and the Fundamental Research Funds for the Central Universities (no. 226-2022-00087). Wei Dong is the corresponding author.

REFERENCES

[1] B. S. I. Group, "Bluetooth Specification Version 4.0," Tech. Rep., 2010.

[2] D. Rajamohanan, B. Hariharan, and K. U. Menon, "Survey on smart health management using BLE and BLE beacons," in *Proceedings of IEEE ISED*, 2019.

[3] L. F. Del Carpio, P. Di Marco, P. Skillermark, R. Chirikov, K. Lagergren, and P. Amin, "Comparison of 802.11 ah and BLE for a home automation use case," in *Proceedings of IEEE PIMRC*, 2016.

[4] R. Tei, H. Yamazawa, and T. Shimizu, "BLE power consumption estimation and its applications to smart manufacturing," in *Proceedings of IEEE SICE*, 2015.

[5] M. Aly, F. Khomh, Y.-G. Guéhéneuc, H. Washizaki, and S. Yacout, "Is Fragmentation a Threat to the Success of the Internet of Things?" *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 472–487, 2018.

[6] D. Zeng, S. Guo, and Z. Cheng, "The Web of Things: A survey," *Journal of Communication*, vol. 6, no. 6, pp. 424–438, 2011.

[7] "MQTT Community. MQTT." http://mqtt:org.

[8] "ZeroMQ." http://zeromq:org/.

[9] M. Spörk, C. A. Boano, M. Zimmerling, and K. Römer, "BLEach: Exploiting the full potential of IPv6 over BLE in constrained embedded IoT devices," in *Proceedings of the ACM SenSys*, 2017.

[10] A. Dunkels, "uIP-A free small TCP/IP stack," *UIP*, vol. 1, pp. 1–17, 2002.

[11] "Generic (GNRC) network stack." https://doc.riot-os.org/group_net_gnrc.html.

[12] "BLIP 2.0," http://tinyos.stanford.edu/tinyos-wiki/index.php/BLIP_2.0.

[13] "nRF52840." https://www.nordicsemi.com/Products/nRF52840.

[14] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proceedings of IEEE INFOCOM WKSHPS*, 2013.

[15] Dunkels, Adam, "Design and Implementation of the lwIP TCP/IP Stack," *Swedish Institute of Computer Science*, vol. 2, no. 77, 2001.

[16] A. MyNewt, "NimBLE Introduction." http://mynewt.apache.org/ network/ble/ble_intro/.

[17] "CHIP," https://www.wespeakiot.com/chip/#What_is_CHIP.

[18] S. Kumar, M. P. Andersen, H.-S. Kim, and D. E. Culler, "Performant TCP for Low-Power Wireless Networks," in *Proceeding of USENIX NSDI*, 2020.

[19] T. Li, K. Zheng, K. Xu, R. A. Jadhav, T. Xiong, K. Winstein, , and K. Tan, "TACK: Improving Wireless Transport Performance by Taming Acknowledgments," in *Proceedings of ACM SIGCOMM*. ACM, 2020.

[20] B. S. I. Group, "Bluetooth Specification Version 4.2," Tech. Rep., 2014.

[21] T. Liu and A. E. Cerpa, "Temporal adaptive link quality prediction with online learning," *ACM Transactions on Sensor Networks*, vol. 10, no. 3, may 2014.

[22] M. Spörk, J. Classen, C. A. Boano, M. Hollick, and K. Römer, "Improving the Reliability of Bluetooth Low Energy Connections," in *EWSN*, 2020.

[23] J. D. A. Almeida, T. Langlois and A. Plakhov, "Parameter adaptation in stochastic optimization," *Cambridge University Press*, pp. 111–134, 1998.

[24] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: congestion-based congestion control," *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[25] D. Rossi, C. Testa, S. Valenti, and L. Muscariello, "LEDBAT: the new BitTorrent congestion control protocol," in *Proceedings of IEEE ICCCN*, 2010.

[26] S. Abbasloo, Y. Xu, and H. J. Chao, "C2TCP: A flexible cellular TCP to meet stringent delay requirements," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 4, pp. 918–932, 2019.

[27] Sangtae Ha, Injong Rhee, and Lisong Xu, "CUBIC: a new TCP-friendly high-speed TCP variant." *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[28] A. Ayadi, P. Maillé, and D. Ros, "TCP over low-power and lossy networks: tuning the segment size to minimize energy consumption," in *Proceedings of IEEE NTMS*. IEEE, 2011, pp. 1–5.

[29] "Datasheet of nRF52840," https://www.nordicsemi.com/Products/nRF52840.

[30] "AWS: Perform Over the Air Updates using Bluetooth Low Energy." https://docs.aws.amazon.com/freertos/latest/portingguide/ota-updates-ble.html.

[31] "The Zephyr Project." https://www.zephyrproject.org/.

[32] C. Gomez, J. Oller, and J. Paradells, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-power Wireless Technology," *Sensors*, vol. 12, no. 9, pp. 11 734–11 753, 2012.

[33] T. Lee, M.-S. Lee, H.-S. Kim, and S. Bahk, "A synergistic architecture for RPL over BLE," in *Proceedings of the IEEE SECON*, 2016.

[34] T. Lee, J. Han, M.-S. Lee, H.-S. Kim, and S. Bahk, "CABLE: Connection Interval Adaptation for BLE in Dynamic Wireless Environments," in *Proceedings of the IEEE SECON*, 2017, pp. 1–9.

[35] P. Kindt, D. Yunge, M. Gopp, and S. Chakraborty, "Adaptive Online Power-management for Bluetooth Low Energy," in *Proceedings of IEEE INFOCOM*, 2015.

[36] M. Spörk, C. A. Boano, and K. Römer, "Improving the Timeliness of Bluetooth Low Energy in Noisy RF Environments." in *EWSN*, 2019, pp. 23–34.

[37] M. Spörk, M. Schuß, C. A. Boano, and K. Römer, "Ensuring End-to-End Dependability Requirements in Cloud-based Bluetooth Low Energy Applications," in *EWSN*, 2020.

[38] E. Park, H.-S. Kim, and S. Bahk, "BLEX: Flexible Multi-Connection Scheduling for Bluetooth Low Energy," in *Proceedings of ACM/IEEE IPSN*, 2021.

[39] S. Raza, P. Misra, Z. He, and T. Voigt, "Building the Internet of Things with Bluetooth Smart," *Ad Hoc Networks*, vol. 57, pp. 19–31, 2017.

[40] B. Campbell, J. Adkins, and P. Dutta, "Cinamin: A Perpetual and Nearly Invisible BLE Beacon." in *EWSN*, 2016, pp. 331–332.

[41] H. Petersen, T. C. Schmidt, and M. Wählisch, "Mind the Gap: Multi-hop IPv6 over BLE in the IoT," in *Proceedings of ACM CoNEXT*, 2021.

[42] C. Julien, C. Liu, A. L. Murphy, and G. P. Picco, "BLEnd: practical continuous neighbor discovery for bluetooth low energy," in *Proceedings of ACM/IEEE IPSN*, 2017.

[43] K. E. Jeon, J. She, P. Soonsawad, and P. C. Ng, "BLE Beacons for Internet of Things Applications: Survey, Challenges, and Opportunities," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 811–828, 2018.

[44] R. Faragher and R. Harle, "Location Fingerprinting with Bluetooth Low Energy Beacons," *IEEE journal on Selected Areas in Communications*, vol. 33, no. 11, pp. 2418–2428, 2015.

[45] J. Fürst, K. Chen, M. Aljarrah, and P. Bonnet, "Leveraging Physical Locality to Integrate Smart Appliances in Non-residential Buildings with Ultrasound and Bluetooth Low Energy," in *Proceedings of IEEE IoTDI*, 2016.