BLEdge: Edge-centric Programming for BLE Applications with Multi-connection Optimization

YEMING LI, College of computer science, Zhejiang University, Hangzhou, China
BORUI LI, College of Computer Science, Southeast University, Hangzhou, China
JIAMEI LV, College of Computer Science, Zhejiang University, Hangzhou, China
WEI DONG, College of Computer Science, Zhejiang University, Hangzhou, China

Recent years have witnessed the rapid growth of IoT (Internet of Things). Bluetooth Low Energy (BLE) is one of the most popular wireless protocols to implement IoT applications because of its energy efficiency and low-cost properties. However, the development of BLE applications is time-consuming and exhausting. Users are required to write programs for both sides of a BLE connection using complicated low-level APIs. Moreover, it needs much expertise for developers to set appropriate parameters in accordance to different application requirements, especially when there exist multiple concurrent BLE connections. To address these problems, we propose *BLEdge*, an edge-centric programming approach for BLE applications with multi-connection optimization. First, we propose a wireless bus abstraction for BLE programming. With this, users can write BLE applications in an edge-centric way, as if the BLE-connected peripherals are physically attached to the edge node. Second, we advocate an optimization approach for BLE connection parameters. This optimization approach considers the time slot collision problem under a multi-connection scenario. We conduct extensive experiments with the nRF52840DK platform. Experiment results show that *BLEdge* can reduce 62.50% to 90.55% LOC (Lines of Code) when developing BLE applications. Furthermore, our parameter optimization approach can reduce up to 42.23% energy consumption.

CCS Concepts: • **Networks** → **Link-layer protocols**; **Programming interfaces**; **Application layer protocols;**

Additional Key Words and Phrases: Bluetooth low energy, wireless bus, connection optimization

ACM Reference Format:
Yeming Li, Borui Li, Jiamei Lv, and Wei Dong. 2024. BLEdge: Edge-centric Programming for BLE Applications with Multi-connection Optimization. *ACM Trans. Sensor Netw.* 20, 6, Article 126 (November 2024), 25 pages.
https://doi.org/10.1145/3698201

This work is supported by the National Natural Science Foundation of China under grant No. 62072396, the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under grant No. 2023C01033, and the National Youth Talent Support Program.
Authors' Contact Information: Yeming Li, College of Computer Science, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: 12021068@zju.edu.cn; Borui Li, College of Computer Science, Southeast University, Hangzhou, Zhejiang, China; e-mail: librchn@gmail.com; Jiamei Lv (Corresponding author), College of Computer Science, Zhejiang University, Hangzhou, China; e-mail: lvjm@zju.edu.cn; Wei Dong (Corresponding author), College of Computer Science, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: dongw@zju.edu.cn.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1550-4859/2024/11-ART126
https://doi.org/10.1145/3698201

1 Introduction

Recent years have witnessed the rapid growth of IoT applications, such as air quality sensing [11, 16], infrastructure monitoring [12], medical [28], and so on. In general, an IoT application contains several peripheral devices (peripheral-device in the following paper to facilitate the distinction) for sensing and actuating, and a mainboard for data processing or forwarding sensing data to the cloud for further analysis. Traditionally, the peripheral-devices are physically attached to the mainboard [11, 16]. Nevertheless, there is a strong requirement for wireless sensors due to installation, flexibility, and repair considerations. Among numerous wireless technologies, **Bluetooth Low Energy (BLE)** is an attractive choice due to its low-cost, energy-efficient, and wide adoption features. Taking structural health monitoring as an example, we could use *BLE-based sensors* which are scattered in all key monitoring locations, and several *edge nodes* which aggregate the sensor readings of their nearby sensors and transmit to the cloud via network.

Typically, to build an IoT application with BLE, developers have two major steps: (i) write application code to read and write data through BLE; and (ii) set appropriate BLE connection parameters to ensure the application requirements. Following the traditional application programming model, however, has two major problems from the system perspective. First, it is tedious to write the program on the edge node for remote sensors/actuators data accessing since users have to achieve numerous basic functions, including device discovery, connecting, and data transmission. Take the widely-used NimBLE protocol stack [4] as an example, users must first write three callback functions to handle scan, **GAP (Generic Access Profile)**, and **L2CAP (Logical Link Control and Adaptation Layer Protocol)** events which are needed to achieve connecting, data transmission, and data receiving. Second, it is hard for users to set the optimal connection parameters to meet the requirements of latency and energy consumption since the connection collisions in the case of multiple connections will complicate the estimation of BLE communication performance.

To address the above two issues in the BLE application development, we propose *BLEdge*, an edge-centered BLE programming system to simplify the BLE-based IoT application development. There are two major challenges when designing *BLEdge*. First, how to propose a universal programming model that offers convenient access to remote sensors/actuators with unified APIs? From the aspect of data transmission, the existing BLE framework is event-driven and users have to handle several connection-related tasks and events, which is tedious and error-prone. How to organize numerous tasks and events to effectively support various network traffic is challenging. From the aspect of user interaction, sensors and actuators are diverse and existing drivers provide different APIs for users. How to include them in a unified programming model is challenging. Second, it is hard to choose the optimal connection parameters for the BLE multi-connection scenario. When an edge device simultaneously connects with multiple remote nodes, collisions between different connections will cause performance degradation. Considering there are large search space for connection parameters. How to find the optimal connection parameters to meet the latency requirement and minimize the energy consumption is challenging.

To solve these, we first propose the wireless bus-based edge-centric programming model, including wireless interface, driver interface, and sensor/actuator data structure. The wireless interface is used to handle all connection-related tasks and it can be compatible with other wireless protocols other than BLE. The driver interface and sensor/actuator data structure help the user easily port existing drivers to BLEdge and access the data with unified APIs. Second, we propose the collision-aware BLE multi-connection optimization algorithm. We model the connection optimization as a non-linear optimization problem and design a genetic algorithm to quickly find the optimal connection parameters from a large connection parameter space.

We have implemented a prototype of *BLEdge* based on RIOT OS [8] and NimBLE BLE stack [4]. Then, experiments with nine real-world IoT nodes are conducted. Results show that the **LOC (Lines of Code)** reduction of the *BLEdge* programming model is from 62.50% to 90.55%. With our connection optimization technique, *BLEdge* can reduce 11.80%–42.23% energy consumption, and the error of our theoretical model is no more than 1.01%. We summarize our contributions as follows:

— We present *BLEdge*, a novel system that includes a wireless bus-based edge-centric programming model. Users can access remote devices without handling any low-level connection-related operations, which can significantly reduce the code consumption for BLE application development.
— We propose a *collision-aware BLE multi-connection optimization algorithm*. By giving the latency requirements and basic application-layer information, the algorithm optimizes connection parameters that reduce energy consumption while ensuring the latency requirements.
— We extensively evaluate the performance of *BLEdge*. Results show that it can greatly reduce the LOC for developing IoT applications, ensure the latency requirements, and reduce energy consumption in the multi-connection scenario.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 introduces *BLEdge* through some usage examples and overviews our system. Sections 4 and 5 describe the design of the wireless bus and connection optimization method, respectively. Section 6 shows the details of implementation. Section 7 evaluates the performance extensively. Section 8 discusses the future works of *BLEdge*. Finally, Section 9 concludes the paper.

2 Related Work

Rapid development: In recent years, the rapid development of IoT applications has attracted great research attention from both industrial and academic communities [31, 32]. Hodge proposed the rapid development idea in 2016 [20]. Then, a code-first design to prototype wearable devices is proposed in [17]. Guan et al. propose the TinyLink [18] and the improved version TinyLink 2.0 [19]. It provides a top-down approach for designing both the hardware and the software of IoT applications. Based on TinyLink, Li and Dong propose AutoLink [23]. It can optimize both the cost and energy consumption of IoT applications. A Linux-based driver model adapted to Tiny OS is proposed in [14], which makes it easier for users to implement new wired peripheral-device drivers. However, these works mainly focus on reducing the code for operating peripheral-devices that are physically attached to the device, without considering the wireless communications. To connect the edge node with multiple remote peripheral-device nodes, users still need to manage the wireless connections, and program both on the edge and remote nodes. LEGO achieves chip-level plug-and-play functionality on IoT devices. LEGO [36, 37] allows the gateway to access the chips on the remote IoT end devices. However, LEGO requires special hardware to convert a variety of chip signals. Besides, when adding a new device, the underlying hardware driver needs to be completely rewritten according to the description language provided by LEGO, and the existing driver code cannot be reused. *BLEdge* provides an edge-centric programming model. Based on the wireless bus, users only need to program on the edge node and can access the remote peripheral-devices as if they are physically attached to it. *BLEdge* is based on COTS BLE chips and it is easy for users to port drivers by just providing a device interface.

Bluetooth Low Energy: Numerous works have been done to improve the functionality and performance of BLE. An optimization method for BLE advertising and scanning is proposed in [25]. BLEach [29] has implemented IPv6-over-BLE stack in Contiki OS. Then it formulates the latency in

a single BLE connection scenario. Furthermore, [30] explores the end-to-end transmission latency from BLE nodes to the cloud server. However, these two works only consider uni-directional data transmission. In IoT applications, it is common for the edge node to control the remote actuators and ask the remote node to reply with an acknowledgment, which limits the usage scenario of [29, 30]. An **Adaptive Online Power-Management (AOPM)** system is proposed in [21]. It is also designed for single-connection scenarios. The key idea is to calculate the average throughput so that it can predict the expected latency of data transmission. The policy of AOPM is to increase the connection interval as long as possible while ensuring no buffer overflows and the transmission latency can meet the requirement. These three works mentioned above are designed for signal-connection scenario. While there are multiple connections, the collisions between them will significantly impact the connection performance. In [13], connection collision is addressed as the overlap of the start time of connection events, but they do not take the length of connection events into consideration. Park et al. proposed BLEX [27]. It investigates a runtime multi-connection scheduling algorithm by dynamically tweaking the anchor points to avoid connection collisions and allocates enough time resources according to some link-layer information. However, BLEX [27] requires the connection interval value must be one of the power series of 2, only nine connection intervals are available, which makes it hard to meet diverse application requirements. Besides, BLEX does not explicitly model the throughput in the multi-connection scenario, it still requires users to manually tweak parts of the connection parameters. RT-BLE [24] ensures the worst-case latency of BLE transmission in the multi-connection scenario. It uses a binary resource tree to avoid connection collision and its latency model does not consider the collision. Same as the BLEX, it also sets the connection interval value to be an integer power of 2. Besides, it relies on the new connection subrating feature in the Bluetooth specification 5.3, which is not fully supported by many chips and protocol stacks yet. This paper proposes a model that formulates the data transmission latency and energy consumption in the multi-connection scenario. Once some application-layer information is provided, *BLEdge* can automatically generate optimal connection parameters that minimize energy consumption and ensure latency requirements.

3 System Design

In this section, we first present three cases to show the usage of *BLEdge*. Then we discuss the target user groups of *BLEdge*. Finally, we propose the design goals of *BLEdge*, overview our system design, and introduce the details of some essential components.

3.1 *BLEdge* Usage

In general, a BLE-based IoT application contains multiple remote peripheral-device nodes for sensing and actuating and an edge node for data processing and forwarding. All the remote peripheral-device nodes connect to the edge node with BLE to form a star-topology network. For the traditional developing method, users have to program both on the edge node and peripheral-device nodes to achieve application logic and deal with BLE connections. *BLEdge* proposes an *edge-centric programming model* for IoT applications development. Users only need to program on the edge node without considering the BLE connection. More specifically, the development process can be concluded as four steps. First, users write their IoT application using the programming model provided by *BLEdge*. With this, users can access both local and remote peripheral-devices using *unified APIs* as if all of them are physically attached to the edge node. When the programming is done, the *BLEdge* chooses the optimal connection parameters for BLE connections and generates the binary executable files. Then, users burn the binary files into corresponding nodes and deploy them. During the system operation, users may need to redirect several connections to new nodes because of node failure, battery drain, and so on. Towards the redirection, *BLEdge* provides a command-line

```
1  DEVICES:
2   remote-temp-1:
3   ADDR: e1:22:ed:f1:cf:81 [or LOCAL]
4   PERIPH_TYPE: BLEDGE_SA_SENSE_TEMP
5   DRIVER_NAME: dht_temp
6   remote-imu-2:
7   ...
8  BIND:
9   temp: remote-temp-1
10  imu: remote-imu-2
11 OPTIMIZATION:
12  imu:
13  ACCESS_INTERVAL: 1000
14  WORK_MODE: ACTIVE [or PASSIVE]
15  LATENCY: 50
16  temp:
17  ...
```

Fig. 1. Configuration file example written in YAML format.

```
1  bledgedat_t imu_dat;
2  int16_t buf[3];
3  int imu_cb(bledgedev_t *dev, void *arg) {
4   /*Receive from ACTIVE mode IMU*/
5   bledgedev_data(dev, &imu_dat, sizeof(buf));
6   double scale = pow(10, imu_dat.scale);
7   double acc_x = imu_dat.data[0] * scale;
8   ... /* Process IMU data */
9  }
10 int main() {
11  double t;
12  int res;
13  imu_dat.data = (uint8_t *)buf;
14  res = bledge_core_init(5);
15  bledgedev_t *imu = bledgedev_find("imu");
16  bledgedev_t *temp = bledgedev_find("temp");
17  bledgedev_read_active(imu, 1000, imu_cb, NULL);
18  while (true) {
19  /*Read the PASSIVE mode temperature sensor*/
20  t = bledge_temperature_read_syn(temp, 800);
21  ... /*Process temperature data*/
22  xtimer_msleep(1000);
23  }
24 }
```

Fig. 2. Example code for temperature and IMU sensor reading.

interface tool for *dynamic connection redirection* without reprogramming the edge nodes. We use three cases to show the details of *BLEdge* usage.

Case 1: In this example, the user accesses remote IMU and temperature sensor data. To implement this application, users should provide a YAML format configuration file for the application (Figure 1) and a code file for the edge node (Figure 2). The configuration file contains three parts: `DEVICES`, `BIND`, and `OPTIMIZATION`. The `DEVICES` field provides the basic information of peripheral-device nodes, i.e., address, the types of the sensor/actuator, and the corresponding

driver. In the ADDR field, users should either claim the peripheral-device is locally attached to the edge node or provide the MAC address of the remote peripheral-device. With this information, *BLEdge* can decide which drivers are required and compile them into peripheral-device nodes' binaries. After the peripheral-device node is powered up, it can enumerate all available drivers and uses them to initialize the sensors/actuators. BIND maps the physical peripheral-devices into the application domain. This field contains multiple key-value pairs, where the key is the name of the peripheral-devices in the application domain and the value is the name in the configuration. In the code file, users can get the device object with the key (e.g., Line 15–16 in Figure 2). The BIND field is useful if there are multiple same applications. Users only need to change the DEVICE field without changing the code file. The other situation is that the DEVICE field contains some extra devices in case of node failure. Once a device is failed it can update the key-value pairs with connection redirection (detailed in case 3). OPTIMIZATION provides some application-level information for *BLEdge* to choose the optimal connection parameter for each BLE connection. *BLEdge* abstracts the WORK_MODE of remote peripheral-devices into active mode and passive mode. The active mode peripheral-devices will report the sensor data to the edge node with the given ACCESS_INTERVAL. The IMU in this example is an active mode peripheral-device, and users subscribe the data with one-second access interval. As for the passive mode peripheral-devices, the edge node should first send a request to it, and the peripheral-device will reply with the result. The actuators should always work in passive mode. The passive mode can be further divided into synchronous (PASSIVE_SYN) and asynchronous modes (PASSIVE_ASY). For the synchronous mode, the edge node will stop and wait for the reply from the peripheral-device node, such as the temperature sensor reading in Figure 2 Line 18–23. As for the asynchronous mode, the edge node sends a request and continues executing the program. Once the reply is received, a callback function is called to process the reply. Users can define the latency requirement of peripheral-device accessing in LATENCY field. *BLEdge* will minimize energy consumption while ensuring the average latency is smaller than the requirement.

The code file is written in C language to implement the application logic on the edge node (Figure 2). To use the *BLEdge*, users first initialize the BLEdge core with the function bledge_core_init() (Line 14). This function starts the BLEdge and connects to all the remote nodes. The parameter of this function is the timeout to discover all devices (5s in the example). The function bledgedev_find() is used to get the object of the remote peripheral-device and the parameter is the name of the device. If there is a error of device discovery (e.g., the node is removed), this function just returns a NULL pointer. Users can access both the local peripheral-devices and remote peripheral-devices with a unified API. For the active mode peripheral-devices, once the edge node receives the data, a callback function is called to process the data (the function imu_cb() in this example). *BLEdge* provides synchronous and asynchronous access for passive mode peripheral-devices. In this example, users read the passive mode temperature sensor data every second in a synchronized manner and the edge node will wait until the reply is received or a timeout happens (800ms for the temperature sensor in this example). If the peripheral-device works in asynchronous passive mode, the edge node just sends a request with bledgedev_read_asy() and continues to execute the following program. The reply will be processed with a callback function similar to the active mode peripheral-devices. Once there is a reading error (e.g., connection lost), it will return an error code. To further simplify peripheral-device accessing, *BLEdge* designs abstract drivers for common peripheral-devices. Line 20 in Figure 2 calls the abstract driver for the temperature sensor. As for the custom peripheral-devices, users can read and write data with the bledgedat_t structure. The IMU data in this example is read from this structure. The data field is the buffer to store the custom data, and the scale is used to represent sensor data in decimal form.

Case 2: In the example below, a sound sensor is defined and users can use the `readstream()` API to get the byte stream, just like the block device reading in Linux. If the connection of the corresponding node is lost, this function just returns a negative error code.

```
1 uint8_t buf[500];
2 bledgedev_t *sound = bledgedev_found("sound");
3 int len = bledgedev_read_stream(sound, buf, 500);
```

Case 3: Users can redirect connections with command-line tools. Assuming a backup IMU is defined in the configuration file, users can redirect the `"imu"` devices in the application code to a backup device named `"imu_bak"`. Besides, users can directly assign a new MAC address.

```
1 >bledge -r imu imu1_bak [or {MAC address}]
2 >bledge --apply-redirection
```

3.2 Target User Groups

In this section, we discuss the differences between the programming model proposed by *BLEdge* and the native BLE programming model, and show the expected user groups.

(1) Beginners of BLE programming. For now, most of the open-source and commercial BLE protocol stacks use event-driven based programming methods, which contain numerous events For example, we investigate the open-sourced NimBLE stack [4] and commercial ESP-IDF [15]. The ESP-IDF has 65 events about BLE, and the RIOT OS has 31 events. Users have to understand all of the events and properly handle them. With BLEdge, all the underlying details of BLE protocols will be masked out, the beginners can focus on implementing the application logic.

(2) Product Engineers. To implement an IoT application with BLE, burdensome procedures of BLE connections are required with the native BLE application development method. More specifically, at the Central side, users first start the BLE scanning procedure and check every advertising packet. Once it finds the advertising packet from the target peripheral-device, it chooses appropriate connection parameters and creates a BLE connection. After the connection is established, it enumerates all services and characteristics on the Peripheral. If the target services and characteristics are available, the Central will subscribe to the corresponding characteristics and services. Finally, the Central periodically receives and processes the notifications from the peripheral-device. On the peripheral-device side, users have to implement advertising functions and register characteristics and services. Product engineers have to program a number of different IoT applications. It is tedious to program this complicated procedure according to different application requirements. With *BLEdge*, users can just provide a configuration file and program the application logic on the edge node. Without managing the burdensome BLE connections, the application development time will be significantly reduced.

(3) Experienced programmer. The experienced programmer can find it easy to program. However, the traditional development method leaves parameter optimization entirely to the users, it is hard and time-consuming for users to get the optimal connection parameter, especially in multi-connection scenarios. Besides, the link-layer scheduling cannot be optimized with cross-layer information (i.e., application requirements). With the application layer requirements, BLEdge can automatically generate the optimal connection parameter which has the lowest energy consumption while meeting the latency requirements.

3.3 System Architecture

Figure 3 depicts the *BLEdge* architecture. There are five essential modules.

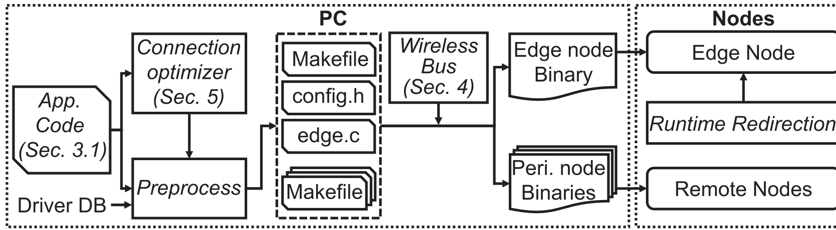


Fig. 3. System architecture of *BLEdge*.

Application Code. The application code provided by users includes a device configuration file and a code file for the edge node written with an edge-centric programming model.

Preprocessing. *BLEdge* implements a parser to analyze the YAML format configuration file to a configuration header file for the edge and one Makefile for each node. The configuration header file (i.e., config.h in Figure 3) includes the basic information of remote devices and the corresponding connection parameters. The basic information is parsed from the `devices` part in the device configuration file. The connection parameters are available from the output of *connection optimizer*. The YAML file gives the basic information of the remote sensors/actuators and *BLEdge* can compile the corresponding drivers into the binaries for peripheral-device nodes. More specifically, in Figure 1, the `DEVICES` field tells *BLEdge* what is the type of the peripheral-device node, and specifies the driver according to the hardware model. With this information, *BLEdge* can generate the Makefile for each peripheral-device node during the preprocessing. Since the drivers are implemented as system modules, the Makefile just includes the corresponding modules, so the drivers will be compiled into the binaries. Once the peripheral-device node is powered up, the wireless bus will enumerate all drivers available and initialize them. As for the edge node, *BLEdge* writes the information (i.e., MAC addresses) into the code file before compilation. It is worth mentioning that one driver module can contain multiple drivers. For example, the `dht` driver module includes the dht temperature sensor driver and the dht humidity sensor driver. To find the correct driver module with the driver name, *BLEdge* maintains a driver database to store the relationship between driver modules and drivers.

Connection Optimizer. The connection optimizer is responsible for choosing the optimized connection parameters for each remote node. Take use of the application-level information in the configuration, the optimizer will minimize the energy consumption and ensure the *average latency* is lower than the design goal. Connection optimization is performed once on the PC before the program compilation. We will further give the details of connection optimization in Section 5.

Wireless Bus. The wireless bus frees users from implementing cumbersome code for BLE connection management. Users can access and control both local and remote devices with a unified API. The design of the wireless bus is detailed in Section 4.

Runtime Redirection. It is a command-line tool integrated on the edge node and users can access this tool through the serial port of the node. Without changing the application logic, *BLEdge* will terminate the corresponding connection and connect with the alternative node. After that, the edge node will check whether the alternative node has the same type as the previous one.

4 Wireless Bus Model

In this section, we first propose the wireless bus architecture in Section 4.1. Then, we discuss how the bus transmits data and manages the devices in Section 4.2. Finally, we present the design of the peripheral-device driver and abstract driver in Section 4.3.

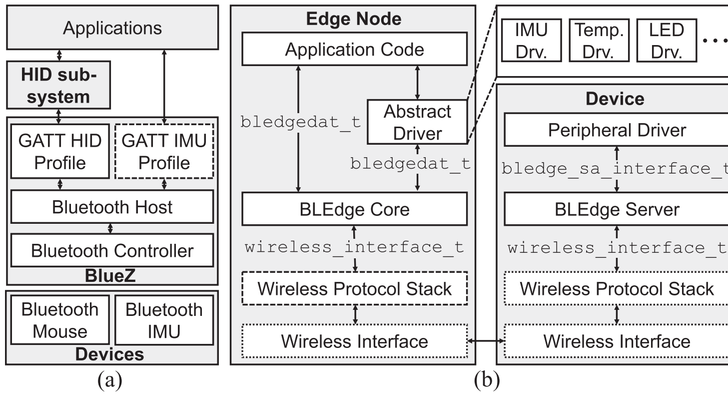


Fig. 4. (a) The existing architecture for Bluetooth devices in Linux. (b) BLEdge wireless bus architecture.

4.1 Wireless Bus Architecture

BLEdge is designed to block the details of a wireless protocol, so the users can focus on implementing the application logic. An intuitive way is using the transport layer protocols, such as BLEach [29] implements IPv6-over-BLE stack to support UDP. However, most of the transport protocols in IoT operating systems are tailored for resource-constraint devices and the performance is limited. Besides, the headers of IPv6 and TCP are non-negligible for the link layer of low-power wireless protocols, leading to lots of extra energy consumption and higher transmission latency [22]. Another way to achieve this goal is using the **Generic Attribute Profile (GATT)** to access the Bluetooth devices, which is the same as what Linux does (Figure 4(a)). The BlueZ Bluetooth stack manages connections and each device provides several profiles. There are global UUIDs for some common profiles, such as the **Human Interface Device (HID)** profile. Furthermore, Linux provides subsystems for a few of them to facilitate the interaction with applications. However, this architecture suffers from two major issues: (1) Global profiles do not cover all common IoT devices, such as IMU sensors. Users must negotiate for the same GATT profile design and assign a 128-bit UUID for each custom profile. (2) To use GATT, users need to create and manage GAP connections. Users still need to have expertise in BLE.

The **LDM (Linux Driver Model)** is widely used to support various wired devices with a unified architecture. The basic driver model contains three parts: driver, bus, and device. Once a new device is physically attached to the bus, its basic information can be read by the bus. Then, the bus uses the basic information to match the compatible driver in the kernel space. After that, the driver initializes the device and starts data transmission via the bus.

We propose a wireless bus architecture for IoT applications, and the details are depicted in Figure 4(b). The BLEdge core and wireless protocol stack form the wireless bus. Like the bus in LDM, the wireless bus takes over all data transmission between the remote peripheral-devices and the edge node through BLE L2CAP channels. The drivers in LDM are running on the host, and they can directly control the hardware through the bus. However, in wireless bus, the edge node and peripheral-device nodes are physically distributed. The wireless bus on the edge node cannot directly operate the peripheral-device hardware. Therefore, the wireless bus architecture has a distributed design. There is a BLEdge server running on the remote peripheral-device node. It will process the command from the edge node, and operate the hardware. The edge node does not directly operate the remote peripheral-device hardware but uses data requests and actuator control

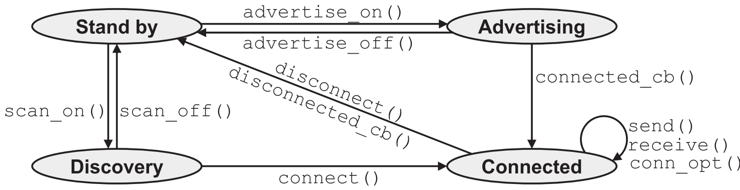


Fig. 5. The connection state machine includes four states. Each function in `wireless_interface_t` can lead to a state transition.

commands to access the peripheral-device. The reason is two-fold. First, direct manipulation of remote hardware needs to configure a number of registers of remote sensors and actuators, which can cause significant network traffic. Second, some peripheral-devices have strict requirements for real-time operation. For example, the ultrasonic distance sensor requires interface voltage measurement every 1ms. This model brings two advantages. First, the wireless bus is responsible for data transmission, which masks the communication protocols to users. Second, the modular design of drivers brings high extensibility. To support a new device, a peripheral-device driver module is needed. Then, it uses a unified interface `bledge_sa_interface_t` that connects peripheral-device drivers with the wireless bus, so users do not need to negotiate for a common design. The workflow of the wireless bus can be summarized into three steps: (1) The remote devices start-up the BLEdge server and initialize the peripheral-devices attached to it; (2) The remote nodes start advertising, and the BLEdge core discovers and connects to the remote nodes according to the configuration; and (3) BLEdge core registers the local and remote peripheral-devices and the applications can access devices directly through the BLEdge core or using abstract drivers.

As for the driver providers, BLEdge has provided some of the drivers and the community can provide more drivers in the future, just like the RIOT OS does. Besides, if the driver is available in the OS, it can be quickly implemented as a driver in BLEdge. Users only need to provide the `bledge_sa_interface_t`. For example, to implement the existing DHT11 driver to BLEdge, only 61 LOCs are required.

4.2 Handling Connection-related Tasks

The goal of BLEdge core and BLEdge server is to handle all connection-related tasks for users, mainly including device discovery and connection management.

Device discovery. To achieve the device discovery, the BLEdge server on the remote node should first start periodic advertising. The advertising data packet contains the MAC address, node name, and the type of the devices that are physically attached to the remote node. The edge node starts scanning and receives the advertisement. We implement a device filter in the BLEdge core to find the target devices and establish connections. The policy of the filter is generated from the configuration header file.

Connection management. First, to enable the BLEdge core and server to interact with various wireless protocol stacks, we propose an interface (i.e., `wireless_interface_t`). It mainly contains eight functions to control the wireless protocol stack, and four callback functions are used by the wireless protocol stack to inform the BLEdge core and server about the state change. To manage the connections, the BLEdge core and server maintain a connection state machine for each connection. The details of the connection state machine and wireless interface are depicted in Figure 5. Each connection has four states. It will be updated by BLEdge core and server once a function is called. It is worth noting that besides the NimBLE stack, the other wireless bus can adapt to other protocol stacks, such as the LPL MAC for IEEE 802.15.4 [35].

4.3 Peripheral Driver and Abstract Driver

Peripheral Driver. In *BLEdge* implementation, each peripheral-device driver is stored as a system module, which includes three files. The first is the driver file. It operates sensors and actuators through physical connections. Usually, driver files have already been provided by manufacturers or operating systems. Second, a porting file is needed to adapt the driver into *BLEdge* compatible sensors/actuators interface, i.e., `bledge_sa_interface_t`. This interface includes information about drivers (e.g., name, type) and function pointers to access and control the device. Third, an initialization code is needed. So the peripheral-device will be initiated and the driver interface will be registered once the node powers up.

Abstract Driver. Each type of bus provides a data structure for data transmission (e.g., USB request block for USB bus). In the field of IoT, RIOT OS proposes sensor-actuator uber layer and an abstract physical data message type [7]. However, the length of the message is limited to 6 bytes, which is not enough for stream data, such as sound sensor data and GPS data. *BLEdge* proposes `bledgedat_t` data structure for message passing, and dynamically allocates memory for it. So, it supports both short data and stream data. The idea of the physical data message is preserved in `bledgedat_t`, and the peripheral-device type, unit, and data length are stored in it. Users can access the data within it, like the IMU data reading in Figure 2. Based on this message type, *BLEdge* proposes abstract driver, which provides a more intuitive way to read sensors or control actuators. Users can directly get the sensor data or control the actuators without operating the `bledgedat_t`. *BLEdge* designs abstract drivers for common peripheral-device types (e.g., temperature sensor).

5 Multi-connection Optimization

In this section, we start with the background of BLE. Then we formulate the BLE communication performance in the single-connection scenario. Finally, we address the overlaps issue and propose an optimization algorithm.

5.1 Background of BLE

The BLE uses a TDMA polling scheme for data transmission. An example of BLE timing diagram is shown in Figure 6. The BLE devices have two roles: Central and Peripheral. In the following section, we use "Peripheral" with a capital at the beginning to represent the connection role in BLE. They will synchronize their clock during connection establishment. Therefore, both of them will periodically turn on their radio to start a *connection event*. The start time of each connection event is *anchor point*, and the time between two consecutive anchor points is *connection interval*, t_{ci}. The connection interval is an integer multiple of 1.25ms between 7.5ms and 4s. Each connection event consists of one or more data exchanges. At the start of each data exchange, the Peripheral is listening on the channel and the Central will send a data packet or an empty packet to the Peripheral. Then the Peripheral will delay **Inter-Frame Size (IFS)** and reply with a data packet or an empty packet. If none of the Central or Peripheral has more data to transmit, the current connection event is terminated. Otherwise, another data exchange will start after the **Minimum Subevent Space (MSS)**. For ease of subsequent modeling, we denote the time for one uni-directional data exchange as $t_{\text{exch}}(l)$, in which Central or Peripheral transmits l bytes data through L2CAP and the other one transmits an empty packet as the link-layer ACK. Formally, it can be presented as:

$$t_{\text{exch}}(l) = \frac{l_{\text{LL}}}{B} + T_{\text{IFS}} + \frac{l_{\text{LL}} + l_{\text{L2CAP}} + l}{B} + T_{\text{MSS}}. \quad (1)$$

Where B is the data rate of BLE physical layer. After Bluetooth 5.0, the data rate can be 1Mbps, 2Mbps, 125kbps, and 500kbps. l_{LL} and l_{L2CAP} are the length of link-layer fields (10 bytes) and L2CAP header (4 bytes), respectively. While transmitting the empty packet, the L2CAP header is

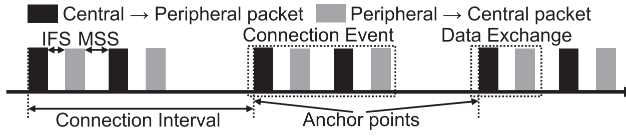


Fig. 6. A basic BLE link-layer time diagram.

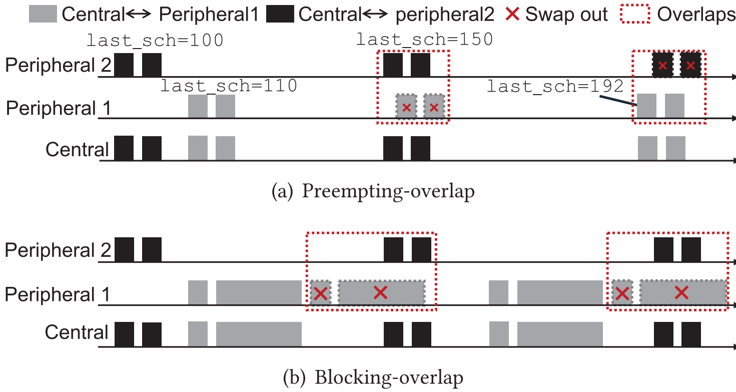


Fig. 7. Two types of overlap. The swapped-out data exchanges will be delayed for one or more connection intervals.

not necessary. The length of each connection event should not exceed the max connection event length T_{maxce}. If both the Central and the Peripheral have no data to transmit, they still exchange empty packets in the connection event as keep-alive packets.

5.2 Connection Overlaps

Collisions can lead to significant performance degradation in the multi-connection scenario. For BLEs, the collisions can be specified as the overlaps between two or more connection events. We divide the overlaps into two types.

(i) Preempting-overlap: Figure 7(a) shows an example of preempting-overlaps. The BLE link layer will reserve a minimum duration for each connection event. It would be enough for both the Central and the Peripheral to transmit a data packet with max payload length. So, at least one data exchange can be performed in each connection event. The preempting-overlaps occur when the minimum duration of different connection events overlapped. The connection event with the highest priority is scheduled and the others are swapped out. Existing BLE protocol stacks often prioritize connections to ensure fairness among them. Take the NimBLE [4] as an example, once a connection is scheduled, its `last_sch` value will be updated to the current CPU time. If preempting-overlap happens, BLE chooses to schedule the connection with the minimum last scheduled time. And the other connection events are swapped out. For the first overlap in Figure 7(a), the Peripheral 2 preempts the Peripheral 1 since the `last_sch` of Peripheral 2 (100) is smaller than the Peripheral 1 (110). After that, Peripheral 1 preempts Peripheral 2 in the second overlap. The BLE stack in Zephyr OS [10] uses a similar strategy. It considers the more swapped-out connection events, the higher the priority for that connection.

(ii) Blocking-overlap: The connection event is terminated earlier than expected if a new connection event starts. An example is shown in Figure 7(b), the Peripheral 1 is sending large data

packets. After each data exchange, the Central estimates the time for the next data exchange and measures the remaining time from now to the next closest anchor point. If the remaining time is not enough to finish the next data exchange, Central will block the current connection event and schedule the others.

If the preempting-overlaps exist, the link layer cannot always transmit any data in each connection event. The average time interval between two effective connection events (without preempting-overlap) is *Effective Connection Interval* t_{eci}, which is no less than the original connection interval. Besides, the blocking-overlaps decrease the length of connection events. We use the *Effective Data Exchange Number* n_{eden} to indicate the average number of data exchanges with one maximum payload length l_{max} bytes data packet that can be transmitted in each effective connection event. Once the connection intervals and anchor points placement are determined, these two values can be obtained as follows. First, calculate the repeating unit t_{ru}, which is the least common multiple of all connection intervals. Then, schedule connections with the same policy used in the BLE link-layer within one t_{ru}. During the scheduling, for each connection, two values should be recorded: (i) n_{eff}, the number of effective connection events; (ii) n^i_{maxce}, the number of maximum data exchanges with l_{max} can be transmitted in the ith connection event. Finally, the t_{eci} and n_{eden} can be calculated for each connection as:

$$\begin{aligned} t_{eci} &= t_{ru}/n_{eff} \\ n_{eden} &= \textstyle\sum_{i=0}^{t_{ru}/t_{ci}} n^i_{maxce}/n_{eff}. \end{aligned} \tag{2}$$

5.3 Formulation of BLE Communication

In this section, we formulate the latency and energy consumption for BLE communication. To simplify the model, we neglect the packet losses. This will not seriously decrease the accuracy of the model because the channel hopping mechanism and **Adaptive Frequency Hopping (AFH)** of BLE can automatically select high-quality channels from the total 37 data channels [29]. The remote peripheral-device nodes work as Peripherals and the edge node works as Central.

Active mode. The existing works have already proposed models for uni-directional data transmission. The AOPM [21] calculates the equivalent BLE throughput according to t_{ci} and t_{maxce}. The latency can be calculated by dividing the packet length by the throughput. The model of BLEach [29] considers the details of the BLE link layer. We extend it to show the end-to-end latency in a multi-connection scenario.

The timing diagram of the Central is shown in Figure 8. The Peripheral node first takes t_{init} time to generate the application data and deliver it to the link layer with a given interval t_{app}. Then, the packet waits until the next effective connection event to start transmission. The waiting time is evenly distributed between 0 and t_{eci} [29]. Therefore, the average waiting time is $0.5t_{eci}$. At last, the Peripheral node transmits the data to the edge node. An application data packet with l_{app} bytes payload is fragmented into multiple L2CAP data packets. The exact number $n_{pak}(l_{app}) = \lceil l_{app}/l_{max} \rceil$, where l_{max} is the maximum payload length of L2CAP packet. The number of connection events to transmit all fragments n_{ce} and the number of data exchanges in the last connection event n_{last} are:

$$\begin{aligned} n_{ce} &= \lceil n_{pak}(l_{app})/n_{eden} \rceil \\ n_{last} &= \mathrm{mod}(n_{pak}(l_{app}), n_{eden}) \end{aligned} \tag{3}$$

The last data exchange contains $l_{rem} = \mathrm{mod}\,(l_{app}, l_{max})$ bytes data. Therefore, the time for an active Peripheral node to transmit a uni-directional data packet can be presented as:

$$\begin{aligned} t_{uni}(l_{app}) =& (n_{ce}(l_{app}) - 1) \times t_{eci} \\ &+ t_{exch}(l_{max}) \times (n_{last} - 1) + t_{exch}(l_{rem}). \end{aligned} \tag{4}$$

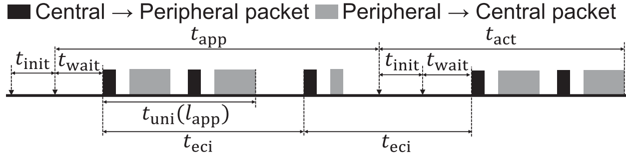


Fig. 8. Central's link-layer timing diagram for active mode Peripheral data transmission.

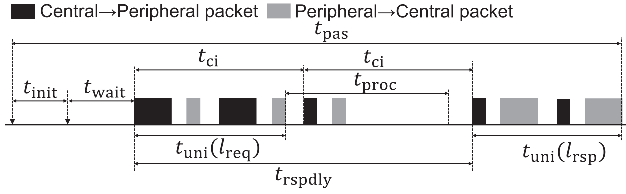


Fig. 9. Central's link-layer timing diagram for passive mode Peripheral data transmission.

The total latency to access the active mode Peripheral node is:

$$t_{act} = t_{init} + 0.5t_{eci} + t_{uni}(l_{app}). \tag{5}$$

Because the length of the application data is determined, the energy for data transmission is fixed. To minimize the total BLE energy consumption, we should minimize the number of keep-alive connection events. Even if preempting-overlaps happened, the Peripheral still turns on its radio. So, the average number of connection events between two consecutive data transmissions is $(t_{app} + t_{init})/t_{ci}$. The number of keep-alive connection events in one second is:

$$n_{ka} = \frac{1}{t_{ci}} - \frac{n_{ce}(l_{app})}{t_{init} + t_{app}}. \tag{6}$$

Passive mode. The AOPM [21] formulates a similar send-and-confirm procedure in BLE. However, it considers only half a packet can be transmitted in one connection event on average. We propose an accurate model for accessing passive mode Peripheral nodes in multi-connection scenario and the timing diagram of the edge node link layer is shown in Figure 9. For the request data transmission, we can consider it as an uni-directional data transmission from the edge node to the Peripheral node. Set the length of the request is l_{req} bytes and the time to transmit the request is $t_{uni(l_{req})}$. Once the request is received, the Peripheral node consumes t_{proc} for processing and then waits until the next connection event. The time between the start of request data and response data transmission can be presented as:

$$t_{rspdly} = \left\lceil \frac{t_{uni}(l_{req}) + t_{proc}}{t_{eci}} \right\rceil \times t_{eci}. \tag{7}$$

Then the Peripheral node consume $t_{uni}(l_{rsp})$ to transmit the response. The total time to access a passive node is:

$$t_{pas} = t_{init} + 0.5t_{eci} + t_{rspdly} + t_{uni}(l_{rsp}). \tag{8}$$

To access the passive Peripherals, *BLEdge* supports both synchronous and asynchronous calls. During the synchronous call (i.e., temperature reading in Figure 2), the application will be blocked until it receives the response data. After that, the application timer will restart. As for the asynchronous call, the application timer will be restarted once the packet is forwarded to the link layer.

BLEdge can also be extended for the burst data, the t_{app} is very large, and the energy for connection keep-alive dominates the total energy consumption. Therefore, the n_{ka} for passive Peripheral nodes can be concluded as:

$$n_{\text{ka}} = \begin{cases} \frac{1}{t_{\text{ci}}} - \frac{n_{\text{ce}}(l_{\text{req}}) - n_{\text{ce}}(l_{\text{rsp}})}{t_{\text{pas}} + t_{\text{app}}}, & \text{Synchronous} \\ \frac{1}{t_{\text{ci}}} - \frac{n_{\text{ce}}(l_{\text{req}}) - n_{\text{ce}}(l_{\text{rsp}})}{t_{\text{init}} + t_{\text{app}}}, & \text{Asynchronous} \\ \frac{1}{t_{\text{ci}}}, & \text{Burst} \end{cases} \tag{9}$$

5.4 Connection Parameters Optimization

The optimization goal is to minimize energy consumption while ensuring the average latency is less than the design goal. It can be presented as:

$$\begin{aligned} \min_{t_{\text{ci}}, t_{\text{offset}}} \quad & n_{\text{ka}} \\ s.t. \quad & t_{\text{act}} \le t_{\text{des}};\ t_{\text{pas}} \le t_{\text{des}}. \end{aligned} \tag{10}$$

Where the t_{des} is the designed latency. The optimization is an integer optimization problem but is hard to solve. The reason is the calculation includes rounding up, rounding down, and modulo operations. Besides, the search space of connection parameters is extremely large. The connection interval can be set to 3,195 values and the searching space contains $3,195^n$ combinations with n node. The offset of each connection is between 0 and its connection interval, which makes the search space larger. Therefore, *BLEdge* utilizes a genetic algorithm to get the optimal answer. Genetic algorithm is widely used for solving integer problems with large searching space [34] and has the ability to solve nonlinear problems [26, 33].

The basic idea of the genetic algorithm is to iteratively eliminate parameter sets that experience timeouts or have high energy consumption while retaining and updating parameter sets that meet the application requirements. More specifically, at the beginning of the algorithm, it initializes the population P, which contains multiple individuals. Each individual is a set of connection intervals and the initial offsets of anchor points corresponding to each connection. The pseudo-code is shown as Algorithm 1. The main difficulty of applying the genetic algorithm is to design the proper ranking function. During the iteration, each individual should be ranked (Line 10). However, in a multi-connection scenario, the latency and energy consumption are difficult to calculate mathematically. The reason is two-fold. First, the optimization problem includes rounding up/down and modulo, making it a non-linear optimization problem. Second, the BLE scheduling policy considers the transient last scheduled time of overlapped connections, and connection overlaps are hard to predict mathematically. To solve this, the ranking function schedules the connections and returns the rank of each individual. Specifically, given the connection the ranking function first uses the `schedule_and_record()` function (Line 2) to simulate the BLE link-layer scheduling within the repeat unit. During the scheduling, it records the time between two consecutive available connection events and the maximum length of each connection event (0 if the connection event is swapped out). Then, the genetic algorithm gets the equivalent connection interval t_{eci} and effective data exchange number n_{eden} with Equation (2) according to the simulation scheduling results (Line 3). After that, the average transmission latency can be calculated according to Equations (5) and (8) (Line 4). Finally, the rank of the individual can be calculated. If transmission latency of all n_{P} Peripherals are smaller than t_{des}, the function returns the reciprocal of average n_{ka}. Some individuals would cause the latency of some connections to exceed their design average latency. Let I_{timeout} denote the set of connections whose latency is over its design average latency. To discard these parameter sets, an intuitive method is to set the return value to 0. However, this led to a sudden change to the return value, and the algorithm would fall

ALGORITHM 1: Genetic Algorithm for Connection Optimization

Require: Device work mode M; Application layer length L_{app}; Device access interval T_{app}; Latency requirements T_{des}; Number of iterations $ITER_NUM$;
Ensure: Connection intervals T_{ci}; Anchor point offsets T_{offset};

```
1:  function RANK_FUNCTION(p)
2:      schedule_and_record(p);
3:      params = cal_equivalent_connection_parameter();
4:      T_lat = cal_average_latency(params, L_app);
5:      cal_rank(T_lat, T_des);
6:  end function
7:  Initiate the population P;
8:  for i = 1 to ITER_NUM do
9:      for each p in P do
10:         RANK = [RANK, rank_function(p)];
11:     end for
12:     P = tournament_selection(P, RANK);
13:     P = multipoint_crossover(P);
14:     P = swap_mutation(P);
15: end for
16: return find_history_best();
```

into the local optimum. To apply a relaxation on the return value, a penalty item is added. It is a negative value, and the more timeouts, the smaller the value. In summary, the rank is presented as:

$$rank = \begin{cases} \frac{1}{\sum_{i=1}^{n_{\text{P}}} n_{\text{ka}}^i}, & \text{on time} \\ \frac{1}{\sum_{i=1}^{n_{\text{P}}} n_{\text{ka}}^i} - \alpha \sum_{i \in I_{\text{timeout}}} \frac{t^i - t_{\text{des}}^i}{t_{\text{des}}^i}, & \text{time out} \end{cases} \tag{11}$$

Where the α is the scale factor and the experience value is between 10^4 and 10^6. With the rank of each individual, the algorithm uses tournament selection to choose some of the individuals that with high ranks can be preserved for the next round of iteration (Line 12). Then, the algorithm uses multi-point crossover and swap mutation to create some new individuals for the next round of iteration. After the iteration is done, the algorithm chooses the individual with the highest rank from the entire history individuals as the optimal connection parameter (Line 16).

6 Implementation

We use nine Nordic nRF52840DKs [6] for the experiments, in which one node works as the edge node, and the rest of the work as remote nodes. The system is built on RIOT OS [8] and Apache NimBle BLE stack [4], which supports Bluetooth 5.4.

We use scikit-opt [9] to implement the genetic algorithm. We set the accuracy of the connection interval value to 10 (i.e., 12.5ms). The accuracy of anchor point offset is 5035us. This value appears unusual because the time NimBLE reserves for each connection event is 5 ms. Since the clock on the nRF52840 runs at 32,768 Hz, and the closest value to 5 ms is 5035 μs. To speed up the optimization, connection intervals of nodes with the same application requirement are the same. We set the search range for connection interval and offset around the values given by the latency model in non-conflict scenarios or as the existing work (i.e., AOPM).

To apply the anchor points offset and the connection interval. *BLEdge* first connects to all nodes with the same connection interval. To change the anchor points placement, we modified

the `ble_ll_sched_master_new()` in the BLE link layer. Then, *BLEdge* updates the connection interval of each connection to the optimal value by the connection update function provided by BLE standard.

We compare *BLEdge* with AOPM [21] and BLEach [29]. The AOPM is based on the traditional BLE programming method and proposes a BLE connection parameter optimization method. The BLEach implements IPv6 protocol over BLE to simplify the BLE application development and proposes a connection parameter optimization method. For the connection parameter optimization, BLEach only models the active mode transmission, and we model a single passive mode transmission as two active mode transmissions. AOPM calculates the latency with the throughput $R_{\text{BLE}} = \frac{l_{\max} N_{\text{seq}}}{T_{\text{ci}}}$, where N_{seq} is the maximum packet number that can be transmitted in the connection interval for active mode and fixed as 0.5 for passive mode [21]. The size of the packet is considered as l_{app} for active mode and $l_{\text{req}} + l_{\text{rsp}}$ for passive mode, respectively.

7 Evaluation

In this section, we evaluate the performance of *BLEdge* in various aspects.

7.1 Overall Evaluation

We compare the overall evaluation by implementing temperature data-collecting applications in four cases:

- **Case 1:** One active mode Peripheral node
- **Case 2:** One passive mode Peripheral node
- **Case 3:** Eight active mode Peripheral nodes
- **Case 4:** Eight passive mode Peripheral nodes

We compare the *BLEdge* with the other two existing works. (1) GATT + AOPM; (2) BLEach. We propose a **Quality of Experience (QoE)** metric for users of *BLEdge*. Considering users should develop the application and maintain it (i.e., change the batteries on the IoT devices), the QoE metric takes both the application development LOCs and energy consumption into consideration. Formally, the QoE metric can be presented as:

$$QoE_m = 3600/LOC_m + 1\text{mJ}/E_m.$$

Where the LOC_m represents the LOC of the application with method m, and the corresponding energy consumption is E_m. The first term represents the time to finish one line of code while assuming the total development time is one hour. The larger the first term is, the easier the application development is. The second term represents how long it takes for the BLE connection to deplete 1mJ of energy, which reflects the lifetime of the node. A higher value indicates a longer battery lifetime, reducing maintenance costs. Table 1 shows the QoE metric in the four cases. The *BLEdge* can increase the QoE by 26.77% to 109.08% compared with GATT+AOPM, and increase 28.57% to 128.29% with BLEach. The reason is two-fold. First, *BLEdge* proposes an edge-centric programming model that masks out the complex details of BLE protocols. Second, *BLEdge* automatically chooses the optimal connection parameter to minimize energy consumption while meeting the latency requirements. In the following section, we evaluate the LOC reduction and connection optimization in detail.

7.2 Lines of Code Reduction

We compare the LOC needed by implementing temperature data collecting applications in the above four cases. In each case, we implement the application with *BLEdge*, GATT profile, and IPv6 over BLE (same as BLEach [29]), where the IPv6 service is provided by GNRC UDP protocol stack

Table 1. QoE of Implementing Different IoT Applications

Method	Case 1	Case 2	Case 3	Case 4
BLEdge	***312.67***	***191.12***	***221.00***	***115.56***
GATT+AOPM	192.62	91.41	174.33	88.82
BLEach	199.22	83.72	171.90	80.42

Table 2. LOCs Comparison of IoT Applications

Method	File	Case 1	Case 2	Case 3	Case 4
BLEdge	Edge Code	17	20	23	30
	Config.	12	12	75	75
	Total	***29***	***32***	***98***	***105***
GATT	Edge Code	112	111	145	146
	Periph. Code	139	115	139	115
	Makefile	19	19	19	19
	Total	**270**	**245**	**303**	**280**
IPv6 over BLE	Edge Code	126	141	154	171
	Periph. Code	158	142	158	142
	Makefile	23	23	24	24
	Total	**307**	**306**	**336**	**337**

in RIOT OS. To remove interference from other codes, we only print the sensor data without any further data processing.

The result is shown in Table 2. The *BLEdge* reduces 62.50% to 89.26% LOC compared with GATT-based implementation, and 68.84% to 90.55% compared with IPv6-over-BLE proposed in BLEach [29]. Although the GATT has provided an abstraction for BLE data transmission, it cannot significantly reduce the LOC. The reason is twofold. First, users still need to manage the GAP connections. Second, besides the temperature service, users also need to implement the basic device information service on Peripheral nodes, which consumes 35 LOCs on the GATT Peripheral node code. For the IPv6-over-BLE, we implement the application with the GNRC protocol stack in RIOT OS. However, users still need to establish a BLE connection and handle the packet headers of UDP and IPv6 packets. *BLEdge* proposes a wireless bus architecture that totally blocks the details of BLE protocol to the users. For *BLEdge*, the *BLEdge* server and Peripheral drivers achieve these functions, and users can focus on application logic development on the edge node.

7.3 User Study

To evaluate the user acceptance of *BLEdge*, we designed and carried out a user study. The participants are 21 students, 5 of them are Ph.D. students, 5 are masters, and 11 are bachelors. All of the participants are computer science students but 18 of them are novices to BLE. We let them try the *BLEdge* and the native BLE development method and estimate their completion time to implement a single or multiple connection application with *BLEdge* and native BLE APIs. Figure 10 shows the result. For single connection applications, 76.19% of users estimate their completion time with *BLEdge* is less than 30 minutes, and there are only 28.57% with the native BLE development method. As for multi-connection scenario, 90.48% users think they can finish developing in 60 minutes with *BLEdge*, but only 47.62% with native BLE APIs.

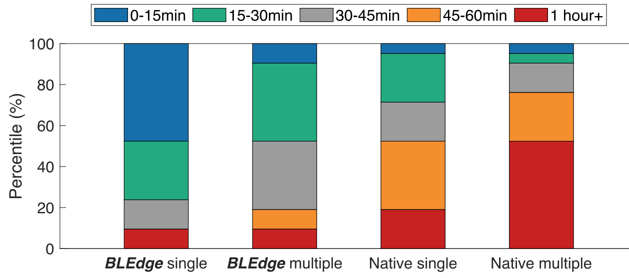


Fig. 10. Estimated development time.

Further, we investigate users' reactions to *BLEdge*. There are 85.71% of the users think *BLEdge* is easy or very easy to program with, and 14.29% of users hold a neutral opinion. When compared with the native developing method, all users think *BLEdge* is better than the native development method.

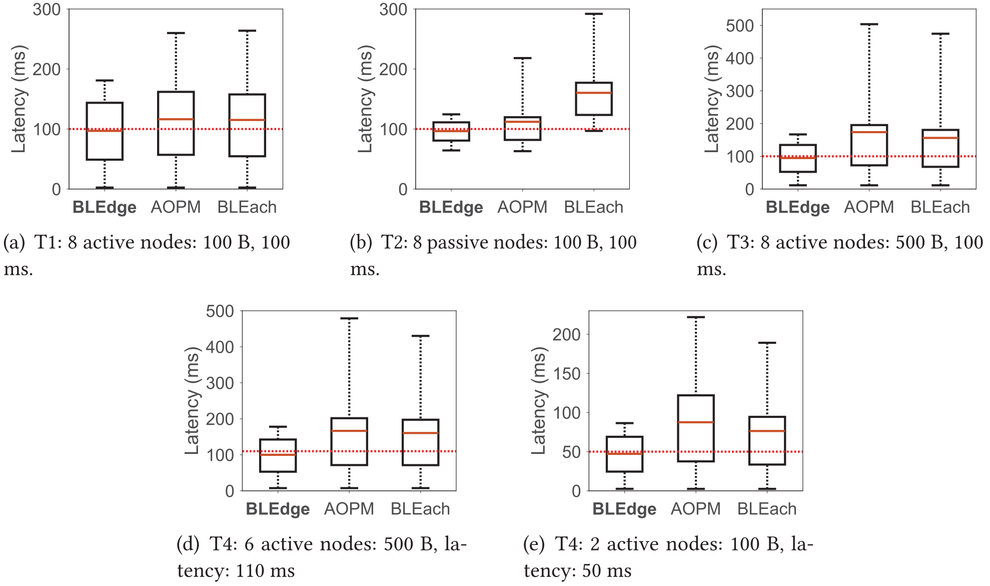
7.4 Connection Optimization

We define four benchmarks with different traffic patterns:

- **T1:** Eight remote nodes work in active mode and send 100 bytes data to the edge node every 1 s. The designed average latency is 100ms.
- **T2:** The edge node reads 100 bytes data from eight passive mode nodes every 1s. The latency requirement is 100 ms.
- **T3:** Eight remote nodes work in active mode and send 500 bytes data to the edge node every 1s. The designed average latency is 100 ms.
- **T4:** Six active nodes transmit 500 bytes packets every 1 s, with 110 ms average latency. Two active nodes transmit 100 bytes packets each 1 s, with 50ms average latency.

The T1 and T2 represent scenarios for accessing sensors or actuators with small amounts of data. For example, reading temperature data, and operating a motor. The T3 and T4 represent the performance in high-traffic scenarios, such as reading sound sensor data.

We compare *BLEdge* with BLEach [29] and AOPM [21]. The experiment results are shown in Figure 11. In all these four benchmarks, the *BLEdge* can ensure the average latency meets the application requirements, while the latency of AOPM and BLEach both exceed the design goals. This is because the AOPM and BLEach are designed for the single-connection scenario and do not consider the connection collisions in the multi-connection scenario. Table 3 shows the average number of overlaps in one second in T1 and T3 benchmarks. In T1, the BLEdge and AOPM have preempting-overlaps. This is because they do not optimize the anchor point offset, and we find that the default NimBLE anchor point scheduling can result in minimal duration overlap of different connections [2]. In T3, both the preempting-overlaps and blocking-overlaps exist, which significantly increase the latency. The reason is, that the packet length in T3 is 500 bytes, which requires a longer connection event for transmission. The longer connection event can overlap with the connection events of other connections and cause the blocking-overlap. The *BLEdge* can cancel the overlaps by tweaking the anchor points offsets and connection intervals in these four benchmarks. For example, in T4, there are two different application requirements. In the single-connection scenario, the connection interval values to meet the 110 ms and 50 ms average latency are 160 and 70, respectively. However, these two values lead to numerous overlaps in the multi-connection scenario. The *BLEdge* automatically sets the connection interval values to 140 and 70. It can guarantee the application requirements in the multi-connection scenario with only a slight increment in energy consumption. In Figure 11(b), BLEach has much higher latency than AOPM. The



(a) T1: 8 active nodes: 100 B, 100 ms.

(b) T2: 8 passive nodes: 100 B, 100 ms.

(c) T3: 8 active nodes: 500 B, 100 ms.

(d) T4: 6 active nodes: 500 B, latency: 110 ms

(e) T4: 2 active nodes: 100 B, latency: 50 ms

Fig. 11. Data transmission latency. The red lines show the average latency and the upper bounds show the latency of 95% data transmission. In all four traffic patterns, *BLEdge* ensures the latencies meet the application requirements.

Table 3. Average Number of Overlaps in One Second

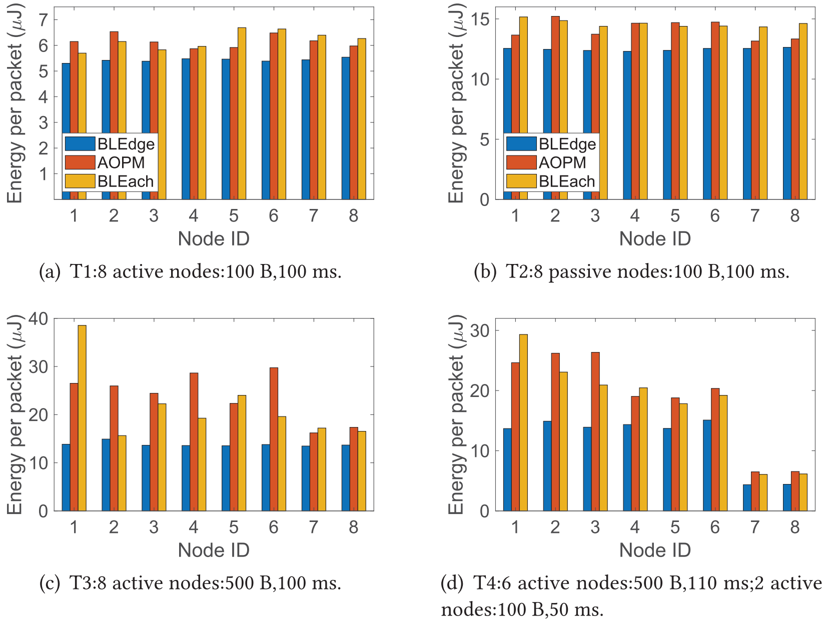
	Method	Preempting-overlaps	Blocking-overlaps	Total
T1	*BLEdge*	0	0	0
	AOPM	2.5313	0	2.5313
	BLEach	2.7271	0	2.7271
T3	*BLEdge*	0	0	0
	AOPM	1.3243	2.3735	3.6978
	BLEach	1.5318	2.1324	3.6641

reason is BLEach only formulates the uni-direction data transmission, while the data transmission is bi-directional for the passive mode device accessing, which increases the BLEach model error.

We evaluate the average energy consumption for transmitting one data packet, calculated as the product of average power and average transmission latency. The experimental result is shown in Figure 12. The *BLEdge* is more energy-efficient than the AOPM and BLEach. Especially in T3, the co-existence of preempting-overlaps and blocking-overlaps significantly increases energy consumption. Besides, the overlaps can lead to uneven energy consumption though several nodes have the same application requirement. In practice, users have to frequently recharge the batteries of some nodes with high energy consumption. In the four traffic patterns, the *BLEdge* can reduce 11.80%–42.23% energy consumption compared with AOPM, and reduce 12.53%–36.17% with BLEach.

7.5 Impact of Node Number and Application Requirement

In the experiments, we set the data packet length is 100 bytes, and the edge node accesses the Peripherals nodes every 1s. Figure 13 shows the impact of connection number. As the number of



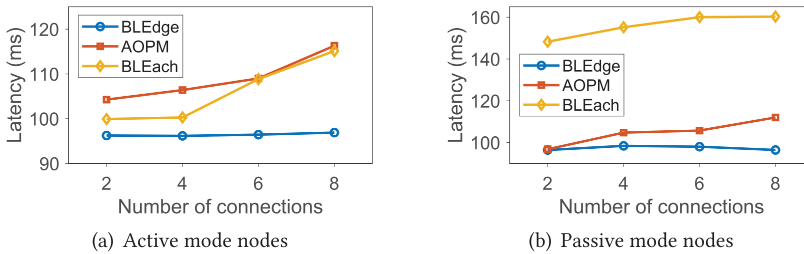
(a) T1:8 active nodes:100 B,100 ms.

(b) T2:8 passive nodes:100 B,100 ms.

(c) T3:8 active nodes:500 B,100 ms.

(d) T4:6 active nodes:500 B,110 ms;2 active nodes:100 B,50 ms.

Fig. 12. Average energy consumption for transmitting one data packet. The physical layer is working on 1Mbps and with 0 dBm transmission power.



(a) Active mode nodes

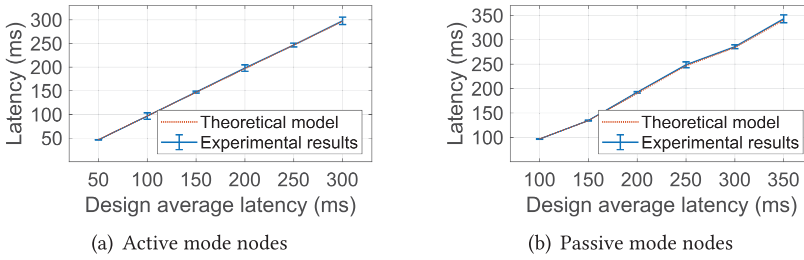
(b) Passive mode nodes

Fig. 13. The impact of Peripheral number. The expected latency is 100ms. The latencies of AOPM and BLEach increase with the node number since the connection collisions become more severe. The *BLEdge* can maintain the latency even with increasing node number.

connections increases, the possibility of connection overlaps is higher. There is a noticeable increase in latency with four or more passive nodes and six or more active nodes. This is because, to achieve 100ms latency, the connection interval value for active nodes is 150, and 50 for passive nodes. The passive nodes have a higher possibility of connection overlaps. *BLEdge* is stable facing the change of connection number. Figure 14 shows the *BLEdge* can precisely meet different application requirements and the theoretical model fits well with the experimental results. The error of the model is less than 0.72% for active nodes and 1.01% for passive nodes.

7.6 System Overhead

In this section, we evaluate the optimization running time and memory overhead, since we first evaluate the running time.



(a) Active mode nodes (b) Passive mode nodes

Fig. 14. The average transmission latency with different application requirements. Each transmission packet contains 100 bytes of data.

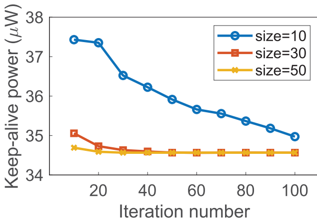
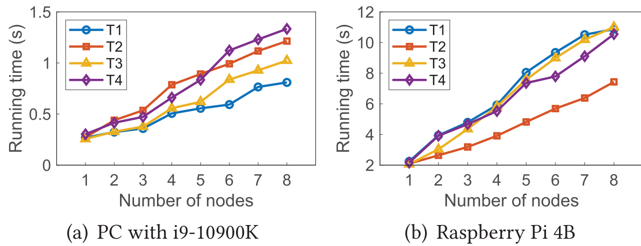


Fig. 15. Genetic algorithm convergence. 50 population size and 50 times iterations are enough to find the optimal connection parameter.



(a) PC with i9-10900K (b) Raspberry Pi 4B

Fig. 16. The optimization runtime. It is acceptable since the optimization is performed during the compilation and the edge node can upload the task to the cloud to accelerate computation.

We evaluate the runtime to perform connection optimization on a PC and a Raspberry Pi 4B. The PC is equipped with i9-10900K CPU and 16GB RAM and the Raspberry Pi 4B has BCM2711 CPU and 4GB RAM. Figure 15 shows the convergence of the genetic algorithm to get the optimal parameters with different population sizes. In our experiments, 50 population sizes and 50 iterations are enough for optimization. Figure 16 shows the running time of the genetic algorithm with different node numbers on PC and Raspberry Pi. For PC, The running time is nearly proportional to the number of nodes, and the time is no more than 1.34s. Because the connection optimization is performed during the compiling, the running time is acceptable. The optimization on Raspberry Pi is slower. Fortunately, the edge nodes are typically connected to the internet. If they require real-time updates to the connection parameters, they can upload the computation task to the cloud servers to accelerate the optimization.

We test the memory overhead of the five most important modules in *BLEdge*. Table 4 shows the details. The memory overhead is totally acceptable for modern BLE chips (i.e., nRF52840 has 256KB RAM and 1MB ROM). The RAM consumption of the BLEdge core, BLEdge server, and wireless interface are relatively large. The reason is these three modules have to allocate some buffer for data transmission and sensors/actuators access. The BLEdge device and BLEdge data are two helper modules, they do not require any buffer and almost all the overhead is in the ROM section (i.e., executable instructions).

8 Discussion

8.1 Portability

In this paper, we implement *BLEdge* on RIOT OS. However, it would be easy to port *BLEdge* to other operating systems and BLE protocol stacks. For *BLEdge* core and *BLEdge* server, a small

Table 4. Memory Overhead of *BLEdge*

	BLEdge core	BLEdge server	Wireless interface	BLEdge device	BLEdge data
ROM (B)	1,853	947	1,909	1,822	136
RAM (B)	1,673	2,127	8,179	1,612	0

number of OS-related APIs are used. For example, starting threads for the two modules, using the *BLEdge* core using the serial to interact with users. Therefore, the modification is minor for those operating systems. For now, the wireless bus is based on the NimBLE stack. However, it is easy to implement the wireless bus with other protocol stacks. The reason is two-fold. First, we propose a wireless interface to unify the APIs that are needed to operate the underlying protocol stacks. The users only need to write a new interface to port the wireless bus on other protocol stacks. Second, NimBLE is a widely used BLE protocol stack and has already been ported to many operation systems, such as Apache MyNewt [3], FreeRTOS [5], and Linux [1]. As for the drivers, the abstract drivers can be reused with minimal modifications since it is based on the *BLEdge* core. The peripheral-device drivers can be different in each OS, so the `bledge_sa_interface_t` is needed to be re-programmed. However, the LOC of each interface is short. For example, the interface of DHT11 consumes 61 LOC.

BLEdge is compatible with different versions of BLE with minor modifications. The most significant differences between different BLE versions are the maximum L2CAP payload length $l_{\max}$ and physical-layer bit rate B. In versions 4.2 and below, the L2CAP payload length is only 23 bytes and only supports 1Mbps. The version after 5.0 supports 247 bytes and multiple data rates (e.g., 1Mbps, 2Mbps, 125Kbps). *BLEdge* is compatible with these differences by only modifying the two parameters. More specifically, in the connection optimization algorithm of *BLEdge*, the $l_{\max}$ and B in the model are used for calculating transmission time and energy consumption. The user only needs to update these two parameters according to the different BLE version specifications. *BLEdge* will optimize connection parameters adaptively.

8.2 Security

The security of *BLEdge* can be improved by integrating security protocols. Specifically, *BLEdge* can integrate with the **Secure Manager Protocol (SMP)** in BLE specification. At the beginning of device pairing, the SMP uses passkey entry or out-of-band pairing methods for authentication and exchanges a **Short-Term Key (STK)**. Then, it uses STK to encrypt the remaining pairing process and distribute a **Long-term Key (LTK)** for encryption of subsequent data transmission. Besides utilizing the SMP, during binary executable files generation, *BLEdge* can distribute private and public keys to the IoT nodes and the edge node, respectively. This can be used for device authentication and symmetric key distribution. The symmetric key is used for data transmission encryption.

9 Conclusion

In this paper, *BLEdge* first proposes an edge-centric programming model and wireless bus architecture, which reduce 62.50% to 90.55% LOC for IoT application development. Then, we formulate the connection performance and address the overlap issue. The model proposed in this paper is accurate, our experiment shows that the error of this model is less than 1.01%. Based on this model, an optimization method has been proposed to ensure the latency requirement and minimize the average energy consumption. By fine-grained connection parameters tuning, *BLEdge* can alleviate

the collision issue in multi-connection scenarios. Compared with the existing works, *BLEdge* can reduce 11.80%–42.23% energy consumption while ensuring the latency requirements.

Acknowledgments

We thank all the reviewers for their valuable comments and helpful suggestions.

References

[1] 2019. https://github.com/apache/mynewt-nimble/issues/615
[2] 2022. nimble/controller: Issue#1135 fix. https://github.com/apache/mynewt-nimble/pull/1138
[3] 2024. Apache MyNewt. https://github.com/apache/mynewt-core
[4] 2024. Apache NimBLE. https://github.com/apache/mynewt-nimble
[5] 2024. FreeRTOS. https://github.com/FreeRTOS/FreeRTOS
[6] 2024. nRF52840 Objective Product Specification v1.8. https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.8.pdf
[7] 2024. RIOT OS phydat. https://doc.riot-os.org/phydat_8h.html
[8] 2024. RIOT OS: The Friendly Operating System for the Internet of Things. https://www.riot-os.org/
[9] 2024. scikit-opt. https://github.com/guofei9987/scikit-opt
[10] 2024. Zephyr Project. https://github.com/zephyrproject-rtos/zephyr
[11] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. 2018. The signpost platform for city-scale sensing. In *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 188–199.
[12] Zhicheng Dai, Shengming Wang, and Zhonghua Yan. 2012. BSHM-WSN: A wireless sensor network for bridge structure health monitoring. In *2012 Proceedings of International Conference on Modelling, Identification and Control*. IEEE, 708–712.
[13] F. John Dian and Reza Vahidnia. 2020. Formulation of BLE throughput based on node and link parameters. *Canadian Journal of Electrical and Computer Engineering* 43, 4 (2020), 261–272.
[14] Soledad Escolar, Jesus Carretero, Florin Isaila, and Felix Garcia. 2007. A driver model based on Linux for TinyOS. In *2007 International Symposium on Industrial Embedded Systems*. IEEE, 361–364.
[15] Espressif. 2024. Espressif IoT Development Framework. https://github.com/espressif/esp-idf
[16] Yi Gao, Wei Dong, Kai Guo, Xue Liu, Yuan Chen, Xiaojin Liu, Jiajun Bu, and Chun Chen. 2016. Mosaic: A low-cost mobile sensing system for urban air quality monitoring. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.
[17] Daniel Graham and Gang Zhou. 2016. Prototyping wearables: A code-first approach to the design of embedded systems. *IEEE Internet of Things Journal* 3, 5 (2016), 806–815.
[18] Gaoyang Guan, Wei Dong, Yi Gao, Kaibo Fu, and Zhihao Cheng. 2017. TinyLink: A holistic system for rapid development of IoT applications. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. 383–395.
[19] Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong. 2020. TinyLink 2.0: Integrating device, cloud, and client development for IoT applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–13.
[20] Shayne Hodge. 2016. A rapid IoT prototyping toolkit. Retrieved on September 3 (2016), 2020.
[21] Philipp Kindt, Daniel Yunge, Mathias Gopp, and Samarjit Chakraborty. 2015. Adaptive online power-management for Bluetooth low energy. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2695–2703.
[22] Sam Kumar, Michael P. Andersen, Hyung-Sin Kim, and David E. Culler. 2020. Performant TCP for low-power wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 911–932.
[23] Borui Li and Wei Dong. 2020. Automatic generation of IoT device platforms with AutoLink. *IEEE Internet of Things Journal* 8, 7 (2020), 5893–5903.
[24] Yeming Li, Jiamei Lv, Borui Li, and Wei Dong. 2023. RT-BLE: Real-time multi-connection scheduling for Bluetooth low energy. In *Proc. of IEEE INFOCOM*. IEEE, 1–10.
[25] Andreina Liendo, Dominique Morche, Roberto Guizzetti, and Franck Rousseau. 2018. BLE parameter optimization for IoT applications. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.
[26] Chhavi Mangla, Musheer Ahmad, and Moin Uddin. 2021. Optimization of complex nonlinear systems using genetic algorithm. *International Journal of Information Technology* 13 (2021), 1913–1925.
[27] Eunjeong Park, Hyung-Sin Kim, and Saewoong Bahk. 2021. BLEX: Flexible multi-connection scheduling for Bluetooth low energy. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (co-located with CPS-IoT Week 2021)*. 268–282.

[28] G. Enrico Santagati and Tommaso Melodia. 2017. An implantable low-power ultrasonic platform for the internet of medical things. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.
[29] Michael Spörk, Carlo Alberto Boano, Marco Zimmerling, and Kay Römer. 2017. BLEach: Exploiting the full potential of IPv6 over BLE in constrained embedded IoT devices. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–14.
[30] Michael Spörk, Markus Schuß, Carlo Alberto Boano, and Kay Römer. 2021. Ensuring end-to-end dependability requirements in cloud-based Bluetooth low energy applications. In *EWSN*. 55–66.
[31] Kazuaki Tanaka and Hirohito Higashi. 2017. mruby–rapid IoT software development. In *Proc. of Springer ICCSA*. Springer, 733–742.
[32] Giacomo Tanganelli, Carlo Vallati, and Enzo Mingozzi. 2019. Rapid prototyping of IoT solutions: A developer's perspective. *IEEE Internet Computing* 23, 4 (2019), 43–52.
[33] Tawan Wasanapradit, Nalinee Mukdasanit, Nachol Chaiyaratana, and Thongchai Srinophakun. 2011. Solving mixed-integer nonlinear programming problems using improved genetic algorithms. *Korean Journal of Chemical Engineering* 28 (2011), 32–40.
[34] Pushpendra Kumar Yadav and N. L. Prajapati. 2012. An overview of genetic algorithm and modeling. *International Journal of Scientific and Research Publications* 2, 9 (2012), 1–4.
[35] Wei Ye, Fabio Silva, and John Heidemann. 2006. Ultra-low duty cycle MAC with scheduled channel polling. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. 321–334.
[36] Chong Zhang, Songfan Li, Yihang Song, Qianhe Meng, Minghua Chen, YanXu Bai, Li Lu, and Hongzi Zhu. 2023. LEGO: Empowering chip-level functionality plug-and-play for next-generation IoT devices. In *Proc. of ACM ASPLOS*. 404–418.
[37] Chong Zhang, Songfan Li, Yihang Song, Qianhe Meng, Li Lu, Hongzi Zhu, and Xin Wang. 2023. A lightweight and chip-level reconfigurable architecture for next-generation IoT end devices. *IEEE Trans. Comput.* (2023).

Received 16 October 2023; revised 19 March 2024; accepted 1 September 2024