SDN-based Service Automation for IoT

Mostafa Uddin Sarit Mukherjee Hyunseok Chang T.V. Lakshman

Nokia Bell Labs, Holmdel, USA
Email: {firstname.lastname}@nokia-bell-labs.com

Abstract—**Bluetooth Low Energy (BLE) is a personal area wireless network technology that is of increasing importance for emerging Internet of Things (IoT) deployments. By design, BLE supports short-range, single-hop communication between a pair of BLE devices. As such, native BLE does not allow network-based policy control or in-network functions for service enhancement. These limitations are impediments to any large-scale BLE based IoT deployment (e.g., in hospital environments), where such sophisticated network-based visibility and control may be required. Relying on cloud-based solutions to meet these requirements has many known shortcomings. This paper proposes an SDN-based architecture for enabling wide area IoT deployments using BLE devices at the edge. We introduce a programmable BLE service switch (BLESS) that is transparently inserted between two communicating BLE devices. BLESS can be programmed at the service layer by a central controller to enable flexible, policy-based switching, as well as various in-network operations in BLE networks. We describe the design of BLESS, its implementation using P4 and OVS, and illustrate its utility through practical use cases.**

I. Introduction

With the rapid growth in the Internet of Things (IoT), Bluetooth Low Energy (BLE) [3] devices are finding widespread applications in medical and health care services, home automation, factory automation, sensing and control, consumer electronics etc [4]. By 2020, the global value of BLE devices deployed for IoT services is expected to be as much as $5.57 billion, with most of the value coming from devices in health care and manufacturing [5]. This popularity of BLE stems from its two key attributes [4]: (i) a low power physical and data link layer suited to low cost devices with long operating life on battery, (ii) a transaction oriented service layer, comprising of Attribute Protocol (ATT) and Generic Attribute Profile (GATT), that facilitates fast development of different services on the devices.

BLE devices fall into two categories; peripheral and central. In health care applications, for example, heart rate monitors, blood pressure monitors, weighing machines, etc. are peripheral devices. Devices like smart phones, tablets, laptops are central devices that access or modify data from the peripherals. Creating an IoT-service needs establishing transactional relationships between central and peripheral devices so that a central device can read/write parameters of the service profiles offered by the peripherals. For example, with a native BLE-based patient vitals monitoring service, an application running on a tablet transacts (i.e., communicates) with heart rate and blood pressure monitors' service profiles, and reports them on a GUI for each patient. With the envisaged growth in feature-rich devices incorporating useful service profiles, the potential for creating a variety of networked services comprising large numbers of interconnected BLE devices is expanding [6]. These networked services will require establishing transactional relationships amongst large numbers of BLE devices and managing them to permit service-enhancing operations such as selective set-up or tear-down of transactions, filtering on BLE packet fields, in-network packet generation etc. (see Section II for motivating use cases needing these capabilities). While each individual device can be (re)configured to comply with the necessary dynamic relationships, it quickly becomes unmanageable as devices get reassigned, new devices get introduced, new services are created, etc. This is a major drawback in using basic BLE that needs to be addressed for large-scale BLE-based IoT service deployment.

978-1-5090-6501-1/17/$31.00 © 2017 IEEE

Existing schemes for IoT service creation and management rely solely on cloud-based solutions where all IoT transactions are sent to a cloud-hosted application which decides on any needed follow-up action. As argued in [1], [2], a cloud-only solution is not best-suited for IoT-based applications due to its inadequate support for privacy and security, scalability, latency and bandwidth guarantees, etc. While a cloud-based approach may work for "ambient data collection and analytics" applications [7], [1], "real-time applications with low-latency" work better by "giving users some control over where applications execute" [1]. In addition, cloud-based approaches either require IP protocol stack in an IoT device, or need to connect the device to the cloud through a gateway. The former approach is not well-suited for devices with small foot-prints [8], [9]. The latter approach requires development of special application level or operating system support (e.g., programming model [10] or virtualization of BLE device [11]). Neither of this is satisfactory. Section VII discusses inadequacies of the current solutions in more detail.

Our premise, in this paper, is that transactional relationships in BLE are best managed when the transactions are monitored and controlled *within* the network, and not solely at individual devices which are resource constrained and hard to modify [6]. Toward this end, we apply the concept of software defined networking (SDN) to BLE networking. To control BLE communication using SDN flow rules, we introduce a new "switch node", which we call the BLE Service Switch (BLESS), into the data path between peripheral and central devices. BLESS maintains link layer connections to the devices to support peer-to-peer connectivity, but controls ATT packet flows at the service layer using SDN rules that are installed by a central controller. In particular, BLESS exploits recent efforts in protocol independent packet processing (e.g., P4 [12]) to program the BLE-based data plane. To our knowledge, BLESS is the first use of SDN in the BLE context, and the first P4 use case for non-Ethernet based packet processing.

We implement BLESS by extending and customizing Open vSwitch (OVS) and its P4 adaptation called PISCES [13]. We develop practical use cases on our current prototype, and show how incorporating SDN technology enables IoT-based service automation in a BLE network. The main contributions of this paper are: (i) Extending SDN to BLE networking for in-network BLE transacation management for large-scale wide-area BLE-based IoT services, (ii) A prototype implementation using P4 with extensions, which to our knowledge is the first use case of P4 in non-Ethernet context, (iii) Practical use cases that illustrate the benefits of the developed system for creating large-scale IoT services using native BLE applications.

II. Motivating Use Cases

Here we describe example services that illustrate why and how control of ongoing transactions from within a BLE network facilitates management of large-scale IoT-based services.

Service slicing: In a hospital environment, BLE peripherals like heart rate and temperature monitors typically offer two service profiles: one for the monitored vitals and the other for device battery status. A medical professional is interested in the vitals of his patients only, while the administrator responsible for device maintenance needs to access the battery status of any device, but without access to the vitals (for privacy reasons). This motivates the creation of device-dependent, service-specific network slices. A physician's slice may consist of the physician's central device and all the health related services of the peripherals carried by this physician's patients only. An administrator's slice may span across his central devices and the battery status service of all available peripheral devices. Note that while these two slices must remain isolated at the service layer, they do overlap at the device connectivity layer. Slices may overlap at the service layer, for example, when a patient is monitored by a nurse in addition to the physician. Service slices must be dynamically created, and be amenable to frequent and real-time association changes in both central and peripheral devices, and associated services.

Currently, a BLE central device that connects (at the link layer) to a peripheral gets *unrestricted* access to any profile information, making service slicing and access policing impossible at any layer of BLE. While applications running on the central device can be made to filter out unauthorized access, this is not an acceptable solution from either a security or management perspective. However, with full control of transactions from within the BLE network, service slicing can be enabled by pushing policy-control flow rules that allow only certain group of transactions based on their origin, destination and type of query, and deny the rest. Use of SDN eases the manageability and installation of the flow rules in the network.

Service enrichment: In the above scenario, when a patient is monitored by both a nurse and a physician, their central devices carry transactional relationships with the same heart rate monitor. It may be desirable to report the heart rate periodically to the nurse, but only the anomalies to the physician (e.g., rates above or below some threshold values based on the age of the patient). The heart rate monitor service is equipped to notify the current vital that can satisfy the nurse' requirement, but not the physician's. While an application at the physician's central device can filter out unnecessary notifications, this does not reduce the flow of transactions into the device. Moreover, the application must be reconfigured for every patient according to the patient's age. Using simple flow rules in the network, one can simply forward all notifications to the nurse's device, and filter out unnecessary transactions for the physician based on the value of the packet field for heart rate, while keeping the native BLE applications unmodified.

Service composition: In a smart home environment, suppose a smart thermostat (a central device) controls an outdoor smart air conditioner (a peripheral) by turning it on or off based on the indoor temperature. A simple BLE app running on the thermostat can perform this task quite efficiently. Now suppose the same household installs a dehumidifier in the basement and would like to run it all the time except when the air conditioner is on. Definitely the dehumidifier can be connected to a smart power socket (a peripheral), but the thermostat cannot control the socket without changing its native BLE app. To support this scenario, the BLE network is programmed such that when a "turn on" transaction from the thermostat to the air conditioner is observed, a new "turn off" transaction to the smart socket is triggered (and vice versa).

III. Background and Contributions

A. Bluetooth Low Energy Protocol

BLE is a connection-oriented peer-to-peer communication technology where peripheral and central devices communicate in a peer-to-peer fashion [3]. Prior to communication, peripherals announce their discoverability and connectability via BLE advertisement packets. Any central device listening to the advertisement can initiate a link layer connection with the peripheral. After establishing a link layer connection, higher layer communication can happen in both directions. In the communication, a BLE device is identified using a 48 bit address. Device addresses can be of two types: a public device address that never changes, and a random address that may change during device boot up. Once a connection is set up, the peripheral stops advertising. Typically, the centrals that run the client applications send requests to the peripherals for the services they support, and the peripherals, running as servers, respond. A central can connect to multiple peripherals giving rise to a star topology. Since BLE version 4.1, a peripheral also can connect to multiple central devices simultaneously, though this is rare in practice [11]. Connectivity to centrals by a peripheral can be restricted by white-listing only the allowed centrals. Also devices can be bonded so that a peripheral advertises only for the central that it intends to connect to.

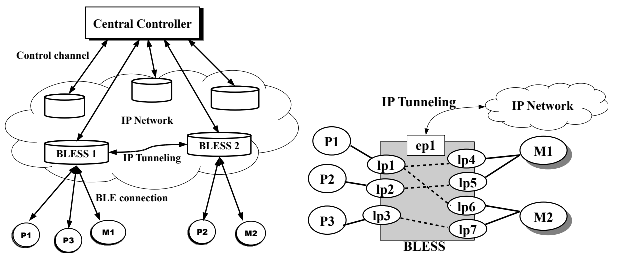
As BLE does not support packet routing natively, reachability from a device to other devices is severely limited unless the devices are directly connected.

On top of the link layer, BLE runs a Logical Link Control and Adaptation Protocol (L2CAP) that is used for multiplexing different higher layer protocols and keeping them oblivious to link layer's packet structure through fragmentation and reassembly. The Generic Attribute Profile (GATT) layer provides a data abstraction and service description model for a BLE device using *attributes* expressed as key-type-value tuples. The attribute key is a 16 bit *handle* used to access it. The type is a universally unique identifier (UUID) which can be 128 or 16 bits. GATT runs atop the Attribute Protocol (ATT) which is a stateless transaction-oriented command-response protocol used to exchange the attributes between peers on top of L2CAP. ATT commands can be of three types, namely, `Read`, `Write` and `Notify`. ATT maintains strict sequencing of requests in the sense that if there is any outstanding response from a peripheral device, no further requests can be sent to it till the response is received. A peripheral provides one or more services using GATT profiles (either vendor or Bluetooth SIG defined), and exposes the service(s) to the world by behaving like a server at the L2CAP layer. A central device connects to the peripheral at the L2CAP layer and accesses the service using ATT transactions. Access control to a device in BLE is mostly limited to the link layer during connection setup. As a consequence, once a central device gets connected to a peripheral, the former can access *any* service offered by the latter. So dynamic device selection based on service is not possible at a higher layer where the service is created [11].

B. Challenges and Contributions

The first challenge in extending SDN to BLE networking is to decide at which protocol layer packet flow rules should be introduced. As noted before, when a central device establishes a link layer connection to a peripheral device, it can access all the services supported by the peripheral device. However, to offer flexibility in service creation, the access to the service profile (or different sub-profiles within a profile) must be dynamically controllable depending on the role of the central device (refer to Section II). To keep a clear separation between connectivity and service, we choose to apply policy control at the service layer of BLE (composed of ATT and GATT) where service profiles are maintained and accessed. This implies that even if a peripheral device is connected to multiple central devices, accessibility to service profiles is controlled based on the identity of the central devices. Moreover, as the access policy becomes dynamically modifiable from SDN's central controller, new services can be added, and existing services modified without interrupting ongoing services.

The next challenge is to determine a node in the BLE network that is suitable for applying the policy control as ATT packets flow through the node. BLE is a peer-to-peer protocol and does not require packet forwarding using an intermediary switch between the devices. Without a switch in place it is not possible to inspect the packet flow between the devices, which in turn prohibits any policy enforcement. To address this challenge, we introduce BLESS, a switch in the BLE network that is transparently inserted between the devices without violating any BLE protocol. BLESS resides in the data plane, has access to the full protocol stack, and is used to enforce policy at the service layer.



(a) (b)

Fig. 1: (a) SDN controlled BLE network. (b) BLESS and BLE device link layer connection.

Challenges in building BLESS are manifold. Existing SDN switches are IP protocol based. Their connectionless architecture is not suitable for BLESS as BLE is natively connection oriented between the peers. Hence, we design BLESS so that it behaves as a central device to a peripheral device and as a peripheral device to a central device. In its role as a peripheral device, BLESS advertises on behalf of a peripheral device that is connected to it. In its role as a central device, BLESS maintains a link layer connection with each peripheral device within its range. Since a central device connects to each peripheral device separately, a single link layer connection between BLESS and a central device is not enough to multiplex across different peripheral devices. Therefore, in its role as a peripheral device, BLESS maintains one connection per peripheral device that the central device needs to communicate with.

Clearly, BLESS (transparently) breaks the native peer-to-peer connection model and introduces the capability to "forward" packets. This necessitates device addressing and identification in BLESS, which are not available in the service layer. BLESS adapts metadata structures and exploits ATT protocol's serialization to resolve correct packet addressing and forwarding (see Section IV).

Once BLESS is inserted in a network of BLE devices, a central controller pushes ATT and GATT layer service rules into it in real-time. BLESS employs a stateless match-action packet forwarding model to examine each packet and apply the action of the matched rule that enforces access and policy control.

To extend reachability, we augment BLESS with backhaul IP connectivity so that multiple such switches can be connected over an IP network (see Fig. 1a). Once a peripheral device gets connected with a BLESS node, the central controller determines the set of BLESS nodes that should announce on behalf of the device via BLE advertisements. This makes the peripheral device "reachable" to a central device that is connected to a remote BLESS which is beyond the peripheral device's physical range. If the central device wants to connect to the peripheral device, it simply responds to the advertisement packet originating from the local BLESS. For example, in Fig. 1a, BLESS 2 can advertise for peripheral device P_3 and let central device M_2 connect to it even though they may be far apart. If peripheral and central devices are connected to two different switches, the SDN controller installs forwarding rules to carry packets from one BLESS to another within an IP tunnel.

IV. BLESS Architecture

In order to implement rule-based control in native BLE, BLESS is transparently interposed between any two communicating BLE devices, and then acts as a programmable software switch for the BLE communication between connected devices. Multiple of such BLESS nodes are interconnected with a backhaul IP network to form the data plane of BLE network infrastructure. In such network deployment, a BLESS node has three key functional components. The *BLE connectivity module* creates and maintains link layer connections with peripheral and central devices within its vicinity. The *BLE packet switching module* applies rule-based control on native BLE packets between connected devices based on match-action rules. The *IP connectivity module* creates and manages backhaul IP connections which are used to tunnel BLE packets to other remote BLESS nodes over an IP network. Essentially the BLE/IP connectivity modules are responsible for BLESS management plane functionality (i.e., creating links/ports), while the packet switching module is responsible for BLESS data plane operations which can be programmed with an OpenFlow control plane protocol. In the following, we describe the BLESS components in more detail.

A. BLESS Components

1) BLE Connectivity Module: At the link layer, BLESS needs to provide a transparent one-to-one connection between a pair of peripheral and central devices. To achieve this, BLESS operates a designated port on which all advertisements from nearby peripheral devices are received. Upon receiving an advertisement, this module by default forwards it to the controller, which decides whether to accept the advertisement. If it is accepted, the controller instructs the connectivity module via a management plane protocol to set up a link layer connection with the peripheral device and to create a corresponding port (e.g., `lp1`, `lp2`, `lp3` of Fig. 1b). We can ensure that no other central device directly connects to the advertising peripheral device by white-listing only the BLESS nodes in the peripheral device and/or by adopting an advertisement jamming scheme [14]. From then on, BLESS advertises on behalf of the connected peripheral device, using the peripheral device's address. When multiple BLESS nodes are interconnected via backhaul IP connections, depending on connection policies, the controller can send the received advertisement to other remote BLESS nodes as well, so that they can also participate in the advertisement for this peripheral device within their own coverage areas.

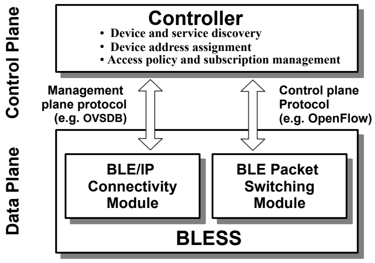

Control Plane
Data Plane
Controller
• Device and service discovery
• Device address assignment
• Access policy and subscription management
Management plane protocol (e.g. OVSDB)
Control plane Protocol (e.g. OpenFlow)
BLE/IP Connectivity Module
BLE Packet Switching Module
BLESS

Fig. 2: BLESS architecture.

In response to an advertisement from BLESS, a central device connects to BLESS at the link layer. Since BLESS uses the same advertisement packet with the same address as received from the peripheral device, the central and the peripheral devices get connected via BLESS at the link layer transparently. This also means that when a central device connects to n peripheral devices via BLESS, there are n link layer connections between the central device and BLESS. This is in conformance with BLE standard 4.1 and later that allows a peripheral device (BLESS in this case) to connect with multiple central devices. Moreover, by allowing multiple link layer connections from a central device to BLESS (instead of a single connection), the client application(s) running on the central device can remain unmodified (see Section IV-A2).

The connectivity module creates a port for each link layer connection from a central device, which has a one-to-one mapping to a port created for a peripheral device (as shown in Fig. 1b). When a new port is created for a peripheral or central device, the controller is notified of this event (e.g., `port-status` message in OpenFlow [15]), upon which it assigns a unique address to the connected device (see Section IV-A2).

A link layer connection between BLESS and a BLE device can be closed if the device permanently becomes out of range or moves away to connect with a different BLESS node. When this happens, the connectivity module removes the corresponding port in BLESS, and the controller is notified of this event, similar to port creation events. In response, the controller removes the corresponding match-actions rules across different interconnected BLESS nodes. To avoid frequent port deletion/re-addition and spurious rule updates from transient connection failures, the connectivity module sets the timeout for a port (instead of removing the port right away) when a connection associated with the port gets closed. If a new connection is re-created within the timeout period using the same device address recorded in the port, the port is simply updated with the new connection.

2) BLE Packet Switching Module: After BLESS is transparently placed in between two communicating devices at the link layer, the BLE packet switching module examines the packet flow at the service layer to enforce rule-based packet forwarding and control. While packet forwarding requires per-packet source and destination addresses, the service layer of BLE uses L2CAP packets which are devoid of device addresses. In our model, the central controller assigns a unique address to each device when the device connects to BLESS. A public or a static device address can be used as is since it remains fixed for the duration of the connection. Only a random address may be mapped to a fixed ephemeral address by the controller (e.g., by using hashing on some of the device parameters). The address is used globally for addressing a device (including advertisement as described above) and orchestrating flow rule instantiation and packet forwarding while the device is connected to the BLESS network. The address is carried as metadata with a packet in the service layer to make packet forwarding decisions.

By default, packets are forwarded based on the destination device address. It is the responsibility of BLESS to gather the device address (and update the metadata field) based on the port the packet comes from. Since a central device has a unique port to connect to each peripheral device, BLESS can easily determine the destination peripheral device address of an incoming packet from a central device. A peripheral device maintains only one connection/port with BLESS, regardless of how many central devices it communicates with. Thus an incoming packet from a peripheral device can potentially be destined to any of the central device(s) that it communicates with. BLESS resolves the packet's destination device address in this case by leveraging ATT's Sequential Protocol feature [3], which restricts a central device to have at most one outstanding request per peripheral device at any time.

BLESS implements a serialization mechanism that emulates a sequential request-response at each peripheral device's port by controlling the flow of requests from the central devices to a peripheral device. Only the port that is locally connected with the peripheral device implements this serialization mechanism. The goal of serialization is to uniquely match the response from a peripheral device to the corresponding request, so that BLESS can determine the destination central device of the response. Since a central device can have multiple ports, the central device address alone is not enough to forward a response packet, but the central device's port address is needed as well to determine where to forward the response packet from the peripheral device.

BLESS can take a naive approach of sending one ATT request packet at a time till it receives a response from the peripheral device. In a more pipelined approach, BLESS can send multiple ATT request packets at a time so long as they are not conflicting in terms of protocol parameters and source device's address. There are a few ATT operations (e.g., *notification*) which do not follow the request-response model, and thus the serialization mechanism cannot be applied. In these cases, the controller keeps additional state information to determine the destination device address (see Section V-D).

3) IP Connectivity Module: While BLESS packet switching operates on the native BLE service layer, a backhaul IP connectivity is needed to apply rule-based control beyond the limited physical range of BLE devices (e.g., for communication between a pair of peripheral/central devices connected to two different BLESS nodes). For this we make each BLESS node support TCP/IP protocol stack. When two BLESS nodes are interconnected via an IP tunnel as part of connection policies, the IP connectivity module in either node establishes a TCP connection to the other, as instructed by the controller, and a corresponding port pair is created in these two nodes, and BLE packets are tunneled via this port pair between them.

B. Control Plane Operations

In a network of multiple BLESS nodes, each BLESS node connects to a central controller for management plane and control plane operations. As management plane operations, the central controller decides whether to allow a BLESS node to connect to a peripheral device, as well as which pair of BLESS nodes should be interconnected via IP tunnels. Once peripheral and central devices are connected to BLESS, and necessary IP connectivity is established among BLESS nodes, the controller performs various control plane operations as follows.

- **Device address assignment:** assigns a unique address to a connected peripheral device.
- **Device discovery:** makes connected peripheral devices discoverable from one or more BLESS nodes.
- **Service discovery:** discovers available services offered by a connected peripheral device.
- **Access policy enforcement:** makes services selectively accessible to connected central devices.
- **Subscription management:** manages the list of central devices subscribed for a specific notification service of a peripheral device.

C. Data Plane Operations

BLESS' data plane operations are performed at the ATT/-GATT service layer, where a pair of central and peripheral devices communicate with each other as a client and a server, respectively, for a particular GATT service. The underlying data exchange protocol called ATT supports three categories of ATT commands: read attributes, write attributes and attribute notification. As data plane operations, BLESS blocks, modifies and forwards these ATT packets, or even generates new ATT packets according to device-specific policies, without violating the semantics of the GATT service-level communication between the connected devices. For this BLESS relies on a set of match-action rules to apply to ATT packets.

V. BLESS Prototype

BLESS adopts the BLE protocol communication as the data plane and GATT/ATT-centric flow rules for control plane. This makes BLESS an ideal candidate for exploiting recent advances in protocol independent packet processing [12], [16], which lay the groundwork for the programmable switch architecture for non-Ethernet traffic. In particular, our prototype implementation of BLESS builds on an existing programmable, protocol-independent software switch called PISCES [13] which allows a datapath compiler to convert P4-based datapath specifications [17] into a target software switch derived from Open vSwitch. In our prototype, we address the limitations of PISCES' datapath compiler, and implement BLESS-specific components (e.g., BLE/IP connectivity modules) and custom actions (e.g., rematch). In the following, we describe our prototype in detail.

A. P4-based Datapath Compilation

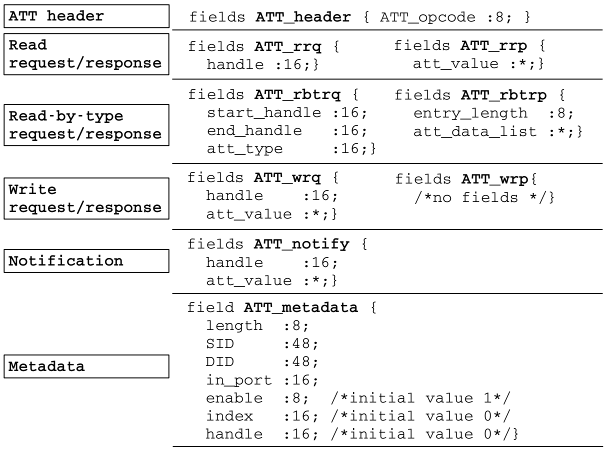
A P4-based datapath protocol specification (known as a "P4 program") contains the definitions of several key components of switch processing, such as packet headers, parsers, tables, actions and control flows, the combination of which defines the packet processing pipeline of a switch. To instantiate our prototype, we develop a custom P4 program that defines the headers and parsers for processing native BLE packets, and compile it into OVS code base with the PISCES's datapath compiler [13].

Fig. 3 shows BLE ATT command packet definitions in our P4 program (more details in Section V-D). Different ATT commands can have different packet fields depending on the `ATT_opcode` in `ATT_header` which appears in all ATT packets (details in [18], [19]). BLESS maintains additional per-packet metadata (`ATT_metadata`), which contain relevant lower layer information like source/destination device addresses (`SID`/`DID`), the packet's ingress port (`in_port`) and total length (`length`). The additional metadata fields such as `enable`, `index`, and `handle` are used to implement custom actions (see Section V-C).

The existing PISCES datapath compiler is unable to support many new specifications of P4, which are required for BLESS. The following are a few extensions we add to the compiler to support BLESS.

Variable packet field: Many ATT command packets have a variable length field which appears at most once as the last field of a packet. For example, an ATT write request packet (`ATT_wrq` in Fig. 3) has a variable length field called `att_value`. Unfortunately, the PISCES compiler only supports fixed length fields in header definitions. To get around the limitation, when a given header definition contains a variable length field, we set the width of the field as the maximum possible length, assuming that the MTU of a BLE packet is 23. Based on the total length of the ATT packet (`length` metadata field), BLESS can derive the actual length of a variable field.

Per-packet metadata: The PISCES compiler converts metadata definitions in a P4 program as flow-level metadata in OVS. Thus, if an action modifies any flow-level metadata and applies `rematch` action (Section V-C), the modified flow-level metadata will be lost during the next match-action process. However, processing a BLE packet with a list of attributes (e.g., *read-by-type response*) requires metadata to persist across iterative executions of `rematch` action on the packet. We extend the PISCES compiler to support persistent packet-level metadata.



```
ATT header          fields ATT_header { ATT_opcode :8; }

Read                fields ATT_rrq {      fields ATT_rrp {
request/response      handle :16;}          att_value :*;}

                    fields ATT_rbtrq {    fields ATT_rbtrp {
Read-by-type          start_handle :16;     entry_length  :8;
request/response      end_handle   :16;     att_data_list :*;}
                      att_type     :16;}

Write               fields ATT_wrq {      fields ATT_wrp{
request/response      handle    :16;        /*no fields */}
                      att_value :*;}

                    fields ATT_notify {
Notification          handle    :16;
                      att_value :*;}

                    field ATT_metadata {
                      length  :8;
                      SID     :48;
                      DID     :48;
Metadata              in_port :16;
                      enable  :8;  /*initial value 1*/
                      index   :16; /*initial value 0*/
                      handle  :16; /*initial value 0*/}
```

Fig. 3: Packet field and metadata definitions for BLESS.[1]

B. Ports in BLESS

The P4-compiled datapath allows BLESS to process native BLE packets based on match-action rules. Still missing, however, is the BLE-native port interface for receiving and transmitting BLE packets, which is tied to target software switch implementation (e.g., `netdev` in OVS). We decide to implement BLE-specific port interface in *userspace* because the BLESS datapath processes service-layer BLE packets which are userspace packets captured from L2CAP layer sockets. Capturing packets in the BLE kernel space would have the overhead of processing and filtering out different types of irrelevant L2CAP layer packets (e.g., L2CAP empty PDU), which are not part of BLE service layer protocol.

For BLE-native port interface, we implement two specializations of the generic OVS `netdev` port: `netdev_ble` for receiving and transmitting BLE data packets, and `netdev_adv` for receiving BLE advertisement packets from peripherals. These BLE-specific `netdev` ports are created and maintained by the BLE connectivity module described in Section IV-A1. As part of port configurations, each BLE `netdev` port has an associated file descriptor for either an active L2CAP socket (in case of `netdev_ble`) or a listening socket (in case of `netdev_adv`), via which the port reads and writes native BLE packets. The file descriptor information in `netdev` is persisted in `ovsdb-server`, and can be updated on the fly (e.g., in case of temporary connection failure and recovery).

The `netdev_ble` port adds metadata (`ATT_metadata` in Fig. 3) in each ingress packet, which are removed before the packet is transmitted. Among metadata fields, `DID` (destination address) and `SID` (source address) are set differently by the port depending on its target device. If the port is created for a central device, the port uses fixed `DID` and `SID`, as assigned by the controller. If the port belongs to a peripheral device, it determines `DID` of each ingress packet by using serialization (Section IV-A2), while using fixed `SID`. For serialization, the port maintains a FIFO queue which enqueues `SID` of each egress request packet. Then for each ingress response packet, `SID` dequeued from the queue is used as `DID` for the packet. The serialization is disabled for a server-initiated ATT packet (e.g., *notification*), in which case `DID` remains unassigned and this field is ignored in the match-action rules. As an artifact of the datapath compiler unable to support variable length packets, the `netdev_ble` port adds padding to each received

[1] The width of "*" denotes a variable length in P4 language.

packet to MTU size, which is removed before the packet is transmitted.

C. Actions in BLESS

The BLESS prototype supports a range of actions for rule-based control on BLE packets. These actions are realized using a set of built-in actions (e.g., `output`, `goto_table`) from OVS and a set of primitive P4 actions automatically generated by the datapath compiler (e.g., `add_header`, `remove_header`, `add_to_field`, `subtract_from_field`, `deparse`). Some of these primitive actions are used as is, while others are used to define more complex *compound* actions. The following is a list of compound actions implemented in BLESS.

1) Block: This action drops a matching request packet, creates an *error response* packet from the packet, and sends the response packet back to `in_port` of the dropped packet. This action is implemented with a series of `remove_header` and `add_header` actions to convert a matching packet into an error response packet, `set_field` actions to change necessary fields in the response (e.g., swapping `DID` and `SID`), `deparse` and `output` actions to send the response packet back to `in_port`.

2) Rematch: This action allows a given packet to be re-submitted to the BLESS processing pipeline multiple times. This is useful when applying match-action rules to the same packet iteratively to policy control ATT commands that carry a variable length list of attributes (e.g., *read-by-type response*). The number of re-submissions by this action is controlled by updating persistent per-packet metadata fields (`index`, `enable` and `handle`). That is, each time the rematch action is performed on a given packet, the `index` field increments by one, and the `handle` field gets updated to point to the next matching attribute in the attribute list. The `enable` field, which indicates whether or not it is the last attribute in the list, is used as a flag to terminate the iteration of the rematch action.

3) Remove: This action is similar to the aforementioned rematch action, except that it removes a currently pointed handle-value pair from the attribute list.

4) Comparison: This action is similar to the *conditional actions* defined in [13]. It compares a field value with a constant and applies `goto_table` action based on the comparison result.

D. Rule-based BLE packet Processing

BLESS uses policy-specific match-action rules based on native BLE packet fields as well as per-packet metadata. In the following, we describe possible policy controls for three basic ATT commands, and show how they are translated into match-action rules in BLESS.

Read attributes: The *read request/response* ATT command pair (`ATT_rrq` and `ATT_rrp` in Fig. 3) allows a GATT client to read a certain attribute's value from a GATT server using the attribute's handle. The *read request* packet contains the attribute handle which, along with a few other fields, is used by BLESS to find a matching rule. When a *read request* packet is received, BLESS can apply either **forward** or **block** action on the packet. In the latter case, BLESS sends an *error response* packet to the client with "Read Not Permitted" ATT error code. For a *read response* packet, BLESS applies **forward** action to send it to the client.

When multiple attribute values of a certain attribute type are communicated, client/server use a pair of *read-by-type request/response* commands (`ATT_rbtrq` and `ATT_rbtrp` in Fig. 3). For a *read-by-type request* packet, BLESS applies either **forward** or **block** action. For a *read-by-type response* packet which contains a list of attribute handle and value pairs, BLESS can remove or modify one or more handle-value pairs in the packet using **rematch** and **remove** actions according to policies (e.g., to permit the client to access only a subset of the list).

Write attributes: The *write request/response* command pair (`ATT_wrq` and `ATT_wrp` in Fig. 3) allows a client to write a single attribute value at the server based on handle information. When a request packet comes to BLESS, it performs either **block** or **forward** action based on the handle information. A corresponding write response packet is simply **forward**ed to the destination. There is a special write request used when a client subscribes to a service attribute for notifications from the server. In this case, the client sends a write request with the attribute handle with *client characteristic configuration* type (i.e., UUID = 0x2902) to the server. When BLESS applies **forward** action on such a request, it also performs **controller** action to send a copy of the packet to the controller. This enables the controller to manage the set of client devices (referred to as *Notify-Set*) that subscribe to the notification for the corresponding service attribute from the server and to push appropriate rules for delivering the same.

Attribute notification: This is a server initiated ATT command that notifies subscribed client(s) that an attribute value has changed. In our model, a server sends only one notification packet for an attribute value as it connects to only one BLESS node acting as a central device's client. Upon receiving a notification packet from the peripheral device, BLESS applies **forward** action on the packet to each subscribing client in the Notify-Set based on rules pushed by the controller.

VI. Evaluation

For evaluation we deploy the BLESS prototype on Raspberry Pi 3 with four Bluetooth 4.0 USB dongles attached to it. One dongle is designated to connect with commercial peripheral devices, and the rest three dongles are used to connect with three individual Android-based central devices.[2] As peripheral devices, we use August door lock, Eve Energy power socket, Avea LED bulb, and Withings Pulse Ox. Our proof-of-concept prototype and testbed are built for checking the feasibility of BLESS and testing its functionality. The evaluation is not meant for real-life stress testing with a large number of devices under high traffic condition. We first

[2] Using one dongle per central is because Bluetooth 4.0 allows one peripheral to connect with only one central. Bluetooth 4.1 and later supports one peripheral to connect with multiple centrals, in which case a single USB dongle would be enough to connect with multiple centrals.

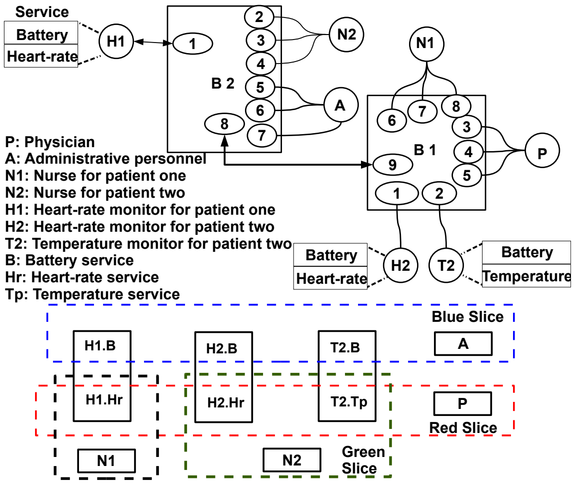


Fig. 4: The hospital scenario with BLESS. Dotted rectangles depict network slices among different groups of BLE devices and services (represented in solid rectangles).

present the implementation of several use cases, followed by performance tests to show the overhead introduced by BLESS.

A. Service Automation using BLESS

To illustrate the inner workings of BLESS, we implement three use-case scenarios of *service slicing*, *service enrichment* and *service composition* described in Section II. Instead of using the actual devices from the example scenarios, we implement the scenarios using the aforementioned commercial off-the-shelf BLE devices with similar interaction among them.

1) Service Slicing: For a service slicing use case, we consider the hospital scenario described in Section II, where different, potentially overlapping, GATT-level service slices are enforced between central devices operated by hospital personnel and peripheral devices carried by patients. Fig. 4 describes this scenario with central and peripheral devices connected to two different BLESS nodes (**B1** and **B2**). Also shown are three GATT services (battery, heart rate and body temperature monitors) offered by the peripheral devices, and several slices (shown in dotted rectangles) created to police access to these services by central devices. As evident, slices can be independent (e.g., Blue and Red) or overlapping at various services (e.g., Red and Green). BLESS ensures that each slice is created and maintained in isolation without interrupting any ongoing services. Fig. 5 shows a snippet of rules installed in **B2** to implement the administrator's service slice (blue slice in Fig. 4) which is composed of his central device **A** and the battery services of all the peripheral devices **H1**, **H2**, and **T2**. In this slice, BLESS must ensure not only that the device **A** can connect to all remote peripheral devices (**H2** and **T2**) at the link layer even when **A** is out of their broadcasting range (`Rule1`), but also that **A** can only access their battery status (`Rule1` and `Rule4`). Any attempt to access non-battery related information on the peripheral devices will be blocked with an error response (`Rule2` and `Rule3`).

```
Rule1:table=0,SID=A,DID=H2,in_port=6,att_opcode=READ_REQ,
     att_rrq_handle=BATTERY, actions=output:8
Rule2:table=0,SID=A,DID=H2,in_port=6,att_opcode=READ_REQ,
     att_rrq_handle=HEART_RATE,
     actions=remove_header:att_rrq,
     add_header=att_err,resubmit(,1)
Rule3:table=1,SID=A,DID=H2,in_port=6,att_opcode=READ_REQ,
     att_rrq_handle=HEART_RATE,
     actions=set_field:0x05->att_length,
     set_field:ERROR_RESPONSE->att_opcode,
     set_field:READ_REQ->att_err_att_request_opcode,
     set_field:HEART_RATE->att_err_att_handle_error,
     set_field:READ_ERROR_CODE->att_err_att_error_code,
     output:6
Rule4:table=0,SID=H1,DID=A,in_port=1,
     att_opcode=READ_REPLY, actions=output:6
```

Fig. 5: Flow rules in B2 for implementing the administrative personnel's service slicing.

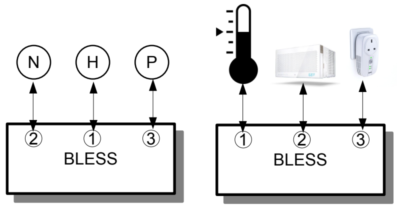


Fig. 6: (a) Service enrichment scenario. (b) Service composition scenario.

2) Service Enrichment: In Fig. 6(a), we have a service slice in which a heart-rate monitoring device (`H`) periodically notifies BPM (heart-beat per minute) of a patient to a monitoring app running at a nurse's device (`N`) and a physician's device (`P`). In this slice, one would like to tailor the frequency/condition of notifications to meet the monitoring requirements by different hospital personnel. For example, the physician is set to receive notifications only when the BPM is over 111. Without touching the native monitoring app on `N` and `P`, one can implement this notification policy within this slice by instantiating device-specific rules in BLESS as shown in Fig. 7. `Rule1` selectively passes a matched notification packet to either table 1 or 2 based on `att_notify_value`. In table 1 (`att_notify_value`>111), BLESS forwards the notification to both `N` and `P`, whereas in table 2, the notification is sent to `N` only. In a large-scale hospital environment which involves many peripheral and central devices, notification policies can be fully tailored and easily updated from the central controller based on origin peripheral devices and target central devices.

3) Service Composition: In the service composition scenario (Fig. 6(b)) described in Section II, the thermostat as a central device cannot control *both* the AC and the dehumidifier. In this scenario, one can instantiate rules in BLESS as shown in Fig. 8 to keep either the AC or the dehumidifier switched

```
Rule1:table=0,in_port=1,att_opcode=NOTIFY,
     att_notify_handle=BPM,
     actions=comparison(att_notify_value,111,1,2)
Rule2:table=1,in_port=1,att_opcode=NOTIFY,
     att_notify_handle=BPM, actions=output:2,3
Rule3:table=2,in_port=1,att_opcode=NOTIFY,
     att_notify_handle=BPM, actions=output:2
```

Fig. 7: Flow rules for implementing the heart rate notification service enrichment.

```
Rule1:table=0,in_port=1,att_opcode=WRITE_REQ,
     att_wrq_handle=AC_STATUS,
     att_wrq_att_value=ON,actions=output:2,goto_table:1
Rule2:table=0,in_port=2,actions=output:1
Rule3:table=0,in_port=3,actions=
Rule4:table=1,in_port=1,
     actions=set_field:POWER_SOCKET_STATUS->att_wrq_handle,
     set_field:OFF->att_wrq_att_value,output:3
```

Fig. 8: Flow rules for implementing the AC-dehumidifier service composition.

on, but not both, to save energy at home. According to `Rule1`, when a "turn-on" ATT write request is forwarded from the thermostat to the AC, a *copy* of this request is sent to table 1 as well, where it is modified to a "turn-off" ATT write request for the smart power socket, and sent out by `Rule4`. Theses rules implement a new service interaction whereby turning on the AC automatically shuts down the dehumidifier. Similar rules can be defined for turning off the AC.

B. *Performance*

We are particularly interested in the performance impact of introducing BLESS in between peripheral and central devices.

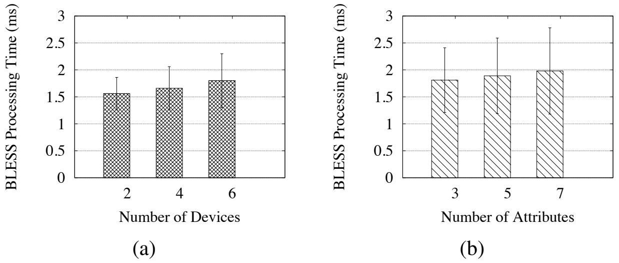
With BLESS				Without BLESS	
CI (ms)		Round-trip delay (ms)		CI (ms)	Round-trip delay (ms)
C↔B	B↔P	Read	Read by type	C↔P	Read
10	10	15.1	16.88	10	12.13
30	10	30.12	31.08	30	30.6
50	10	50.1	50.6	50	50.08
100	10	101.3	102.5	100	101.1
100	20	101.35	102.9	100	101.1
100	30	101.79	102.8	100	101.1
100	50	102.85	103.9	100	101.1

TABLE I: Average round-trip delay of a request-response packet pair between a central and a peripheral through BLESS for different connection intervals.

1) User Perceived Delay: Connection Interval (CI) is the key parameter of BLE link layer that influences the packet round-trip delay on a connection between two devices [11]. It ranges from 7.5ms to 4.0s. BLESS maintains two such connections: one between a central device and BLESS (C↔B), and the other between BLESS and a peripheral device (B↔P). While BLESS (as a central device) can control the CI for (B↔P), the CI for (C↔B) is set by the central device, and is not under BLESS' control.

Table I shows end-to-end round trip delays for *read request/response* and *read-by-type request/response* (with seven attributes) transactions, with and without BLESS, when the CI values vary. For a direct connection between devices without BLESS, CI of (C↔P) is set to the same as that of (C↔B) for the above mentioned reason. Note that with BLESS in the middle, juxtaposing the two CI values yields the same result, and is not shown in Table I.

With BLESS in the middle, the overall round-trip delay is influenced by the link layer connection with a relatively higher CI (C↔B in the experiment). The *read-by-type request/response* transaction incurs more overhead compared to the *read request/response* transaction as BLESS recirculates the response packet multiple times in the pipeline, but the additional delay is very marginal. Having BLESS in the middle hardly lengthens the round-trip delay, compared to the direct connection scenario. This trend is more prominent with higher values of CI, which is more common in practice.



(a) (b)

Fig. 9: (a) Packet processing delay of *read-request-response* transaction in BLESS for different number of connected peripheral/central devices. (b) Packet processing delay of iterating a *read by type response* packet in BLESS for different number of attributes.

2) Processing Time within BLESS: We further investigate the aforementioned experimental result by zooming in the packet processing pipeline within BLESS. We instrument BLESS to timestamp all ingress/egress packets and calculate the time spent by a packet in BLESS. Fig. 9a and 9b show the processing time incurred for *read request/response* and *read-by-type request/response* transactions, respectively.

Fig. 9a shows that per-packet processing increases slightly with load. The more interesting case is Fig. 9b which shows that per-packet processing time in BLESS can vary depending on the packet contents. When an ATT packet carries a list of attributes, BLESS can apply `rematch` action to iterate through the list. The number of iterations required to process such a packet depends on the number of attribute values in the list. Fig. 9b shows that the delay overhead increases marginally as the number of attributes increases. Note that an ATT command packet can have at most seven attributes in the list due to MTU restriction. Comparing the first bar in Fig. 9a and the last bar in Fig. 9b (the overhead of *read-by-type* command), one can see that they differ by about 1ms, which is consistent with Table I. We expect the overhead to be reduced further with the availability of more efficient P4-to-OVS implementation.

VII. Prior Work

There is a large body of work that proposes cloud-based services for IoT devices, which requires every IoT transaction to be exposed to the cloud (see [22] and references therein). The cloud-based approach, however, cannot always satisfy various device/application requirements for privacy, security, scalability, QoS guarantee, etc. [1], [2]. On the other hand, IPv6 over BLE [23] has been proposed for wider reachability, but routing becomes far more expensive in a mesh topology, and IP-multicasting for notification becomes inefficient due to time-to-time sleep mode of BLE nodes [9], [8]. There are local IP-based BLE service gateway solutions [11], [20], where no standard protocol and/or API is defined for networking different IoT devices; each vendor ends up defining its own methodology which makes interworking almost impossible.

	BLESS	Gateway-based (e.g., Beetle [11], IoTivity [20], DeviceHive [21])	Cloud-based (e.g., Azure IoT, IBM Bluemix, OpenHAB, OpenIoT) [22]	IPv6-based [23]
Description	SDN-based solution that centrally monitors and controls the BLE transactions in the data plane between IoT devices	A service gateway that connects to the IoT devices over BLE as well as a remote service agent over IP.	Every IoT device reports to a cloud-hosted application (via hub), which ultimately decides the flow of device-to-device transactions.	IoT devices are equipped with IPv6 over BLE that widens the reachability of BLE devices over IP.
Policy control	In-network policy control on BLE transactions, which remains transparent to the devices.	Application-level policy control which requires creating proxy or virtualization of BLE devices.	Cloud-hosted application imposes policy on every device-to-device transaction.	In-network policy control on TCP/IP traffic.
APIs for device to device transaction	Uses native BLE APIs, which requires no modification in existing BLE applications.	Requires modification of client applications or the central device's native BLE APIs, which translate between the BLE protocol and the gateway protocol.	Requires special cloud based APIs to support device-to-device transactions.	Requires changes in native BLE API implementation (e.g., BlueZ [24]).
Standardization for connectivity management	No standardization is required. The central controller uses OpenFlow protocol to set up rules to orchestrate the connectivity requirements.	Needs special application level or operating system support (i.e., SDK or APIs) with user's involvement for managing the connectivity.	Requires standardization of a new protocol between IoT hubs and the cloud to manage the connectivity.	No standardization is required, but running BLE's native service layer (ATT/GATT) over IPv6 introduces extra burden and inefficiency for managing the connectivity.
Application scenarios	Applicable for real-time applications with low latency requirement. Multiple BLESS nodes with centralize control capability make this solution applicable for large-scale deployment.	Suitable for rapid prototyping in small scale environment (i.e, home deployment), but ad-hoc development at the gateway makes it expensive to support large scale deployment.	Applicable for large-scale data collection or analytics application, but not suitable for real-time applications using commercial cloud. However, open source cloud solution with on-premise deployment can provide low-latency.	TCP/IP-based solutions are not suitable as BLE's very small MTU size degrades protocol performance and introduces extra overhead on the resource-contained low power devices.

TABLE II: Qualitative comparison between different IoT solutions and BLESS.

There are also efforts to develop an open source generic IoT platform that mainly focuses on supporting interoperability by abstracting connectivity and management of the things [20], [21], [25]. Table II summarizes qualitative comparison among different IoT solutions (i.e., proxy-based, gateway-based, cloud-based, IPv6-based) and BLESS.

Moreover, most of the IoT cloud-based or local gateway-based platform focuses on rapid application prototyping and deployment. These solution uses static (re-)configuration which makes dynamic orchestration of services and devices challenging [26]. Unlike the previous work, BLESS focuses on re-configurability, through the use of dynamic orchestration of services and devices. Thus BLESS ultimately provides better application configuration and governance.

VIII. Concluding Remarks

We make a case for SDN-based in-network packet monitoring and control for BLE transactions. To this end, we address the feasibility and realization of the data plane by building a switch that can transparently reside in between communicating BLE devices. The switch is built using OVS and P4 platform, and can benefit from advances made in each of these active research fields. We leave the impact of BLESS in BLE security as a future work, which is out of scope of this paper. We, however, believe that BLESS is able to handle BLE's link layer encryption as it maintains a peering relationship with a device. We leave the details of the control plane realization, and flow rule optimization as a future work.

References

[1] N. Mor *et al.*, "Toward a Global Data Infrastructure," *IEEE Internet Computing*, vol. 20, no. 3, 2016.

[2] B. Zhang *et al.*, "The Cloud is Not Enough: Saving IoT from the Cloud," in *ACM HotCloud*, 2015.

[3] "Bluetooth Core Specification Version 4.2," Bluetooth SIG, 2014.

[4] C. Gomez *et al.*, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology," *Sensors*, 2012.

[5] "Bluetooth SIG 2014 Annual Report," 2014.

[6] T. Yu *et al.*, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things," in *ACM HotNets*, 2015.

[7] M. Hassanalieragh *et al.*, "Health Monitoring and Management Using Internet-of-Things (IoT) Sensing with Cloud-based Processing: Opportunities and Challenges," in *IEEE SCC*, 2015.

[8] W. Shang *et al.*, "Named Data Networking of Things," in *ACM/IEEE IoTDI*, 2016.

[9] W. Shang, Y. Yu, R. Droms, and L. Zhang, "Challenges in IoT Networking via TCP/IP Architecture," UCLA, Tech. Rep., 2016.

[10] W. McGrath *et al.*, "Fabryq: Using Phones as Gateways to Prototype Internet of Things Applications Using Web Scripting," in *ACM SIGCHI*, 2015.

[11] A. A. Levy *et al.*, "Beetle: Flexible Communication for Bluetooth Low Energy," in *ACM MobiSys*, 2016.

[12] P. Bosshart *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[13] M. Shahbaz *et al.*, "PISCES: A Programmable, Protocol-Independent Software Switch," in *ACM SIGCOMM*, 2016.

[14] K. Fawaz *et al.*, "Protecting Privacy of BLE Device Users," in *USENIX Security*, 2016.

[15] "OpenFlow Switch Specification 1.5.0," Open Network Foundation, 2014.

[16] H. Song, "Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane," in *ACM HotSDN*, 2013.

[17] "The P4 Language Specification 1.0.3," The P4 Language Consortium, 2016.

[18] R. Davidson, C. C. Akiba, and K. Townsend, *Getting Started with Bluetooth Low Energy*. O'Reilly Media, Inc., 2014.

[19] "ATT Spec," http://mutsughost1.github.io/2015/02/26/ATT-Spec/.

[20] "IoTivity," https://www.iotivity.org.

[21] "DeviceHive," http://devicehive.com.

[22] A. Botta *et al.*, "Integration of Cloud computing and Internet of Things: A survey," *Future Generation Computer Systems*, 2016.

[23] J. Nieminen *et al.*, "IPv6 over BLUETOOTH(R) Low Energy," RFC 7668, October 2015.

[24] "BlueZ," http://www.bluez.org.

[25] "openIoT," http://www.openiot.eu/.

[26] H. Derhamy *et al.*, "A survey of commercial frameworks for the internet of things," in *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE, 2015.