Virtual Connection: Selective Connection System for Energy-Efficient Wearable Consumer Electronics

Gyeonghwan Hong and Dongkun Shin, *Member, IEEE*

***Abstract*—Wearable consumer electronics such as smartwatch and smartglasses generally use peer-to-peer (P2P) communications to connect to their companion devices such as smartphones. The recent wearable consumer electronics are equipped with multiple heterogeneous network interfaces, which have different power consumption and transfer bandwidths. To optimize energy efficiency, a network interface must be selected for communication depending on workload or system state. Such a technique is referred to as selective connection. However, the current selective connection systems cannot support the P2P-based wearable devices. We propose a protocol-independent user-level system, called virtual connection, that decouples applications from network-specific operations for selective connections. The proposed virtual connection system consists of user-level API and user-level middleware components. To establish the usefulness of our virtual connection, we present a case study of the selective P2P connection for wearable consumer electronics equipped with Bluetooth and Wi-Fi Direct. We demonstrate that, compared to existing selective connection systems, the P2P-based selective connection system implemented with the virtual connection can reduce the energy consumption of the wearable consumer electronics by 1.26-2.31 times.**

***Index Terms*—Communication framework, low-power system, peer-to-peer communication, wearable consumer electronics, wireless communication.**

I. Introduction

Wearable consumer electronics such as smartwatch and smartglasses generally have no direct connection to Internet. Instead, they use peer-to-peer (P2P) communications such as Bluetooth or Wi-Fi Direct (WFD) to connect with a smartphone, which provides Internet services [1]–[7] or computing resources [8]–[12] to its companion wearable devices. This is because the limited battery capacity of wearable consumer electronics cannot afford to use cellular networks causing a significant energy consumption. The Wi-Fi connection through access point (AP) cannot also be used due to the mobility of wearable devices.

Manuscript received June 19, 2020; revised October 20, 2020; accepted October 31, 2020. Date of publication November 3, 2020; date of current version November 25, 2020. This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korea Government (MSIT) under Grant IITP-2017-0-00914 (SW Starlab) (Software Framework for Intelligent IoT Devices). *(Corresponding author: Dongkun Shin.)*

The authors are with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea (e-mail: redc7328@skku.edu; dongkun@skku.edu).

Digital Object Identifier 10.1109/TCE.2020.3035689

To support P2P communications, recent wearable consumer electronics are equipped with multiple heterogeneous network interfaces, which have different power consumption and transfer bandwidths. For example, smartwatches have both Bluetooth and Wi-Fi interfaces. Bluetooth is generally used for transmission of small messages since it requires lower power consumption than Wi-Fi. When the message size is large, Wi-Fi is more energy-efficient due to its high network bandwidth. Therefore, the trade-off between power consumption and network bandwidth of wireless interfaces must be considered to optimize the energy efficiency, and one of multiple network interfaces must be selected for communication considering workload or system state. We call such a technique *selective connection*.

The selective connection systems require three basic functions: *network selection*, *network switching*, and *seamless handover*. The *network selection* function selects a network connection based on the feature of each network interface and the network workload to minimize the communication energy consumption. The *network switching* function opens the connection selected by the network selection function and closes the previous network connection to put the unused network interface into the power save mode or sleep mode. The *seamless handover* function transfers messages without interruptions through any available connection during the network switching. With the seamless handover, the message transfer will not cease during the network switching.

Several selective connection systems have been proposed [5]–[8], and they provide all the basic functions. For example, MPWear [5] is a user-level daemon that selectively activates Wi-Fi or Bluetooth to send or receive messages to or from the Internet, as shown in Fig. 1(a). The throughput-aware network selection policy, the multipath, and the Wi-Fi/Bluetooth adapters of MPWear are in charge of network selection, seamless handover, and network switching, respectively. First, when an application sends messages to the Internet, the MPWear daemon gets the packets by hooking them at the TCP/IP stack through netfilter [13]. In the daemon, the multipath module forwards the message to the currently activated network adapters. After that, a Wi-Fi adapter or a Bluetooth adapter sends the message to the Internet through the underlying network protocol stack. During the execution of the daemon, the network selection policy periodically monitors the system state and selects one of the adapters to activate the proper network interface.

However, when we try to use the previous selective connection systems for P2P-based wearable consumer electronics, several problems must be resolved. First, even though the previous systems [5]–[7] provide all the basic functions, the functions are not designed with considering the characteristics of P2P network devices, which are different from those of normal Internet network devices. When the network device is deactivated by the selective connection, it can be changed to the power save mode (PSM) or the sleep mode. Since the network connection is maintained at the PSM mode, the communication interface can be reactivated with a short delay. However, the PSM of WFD consumes several times higher power compared to other network devices such as Bluetooth [6], [14]. Therefore, it is better to close the WFD network connection and change the WFD device into the sleep mode if it will not be used for a long period. However, it generally takes a significantly long time and requires a high power consumption to establish a P2P connection again owing to the discovery operation of the P2P network. Considering the high cost of network switching, the selective connection for P2P communications must change the network interface only when the currently activated network interface is not useful for a long period. To do that, the network selection policy must predict the future workload. However, the previous techniques [5]–[7] consider only the current system status such as network throughput or network signal strength to select a network interface, and thus they cannot minimize energy consumption due to frequent network switchings.

In addition, P2P network must control the network interfaces of the remote peer device during network switching. For example, when a smartwatch tries to send a large size of data via WFD to a smartphone which is currently connected to the smartwatch through Bluetooth, the WFD adapters on both the smartwatch and the smartphone must be activated. For the purpose, the selective connection system of a device needs to control the network adapters of remote peer devices by sending control messages. However, since the previous techniques [5]–[7] were not designed considering the features of P2P communication, they do not support such a remote control.

Second, P2P networks usually have their proprietary user-level network protocol stack. For example, Bluetooth network stack requires user-level daemons such as BlueZ [15]. Bluetooth LE and Z-wave also use their proprietary network protocol stacks instead of using the in-kernel standard TCP/IP stack. However, the previous techniques such as MPWear [5] and CoolSpots [6] hook communication messages only at the TCP/IP stack by netfilter [13]. The target devices of the techniques are assumed to be connected to Internet requiring every message to pass through the TCP/IP stack. To use the selective connection at P2P-based consumer electronics, it is required to hook communication messages between applications and the user-level P2P network protocol. So, the recently-proposed WearDrive [8] provides user-level API functions to directly get messages from applications, as shown in Fig. 1(b).

Third, the current P2P-based selective connection systems such as WearDrive [8] provide only high-level API functions designed for specific applications such as remote key-value

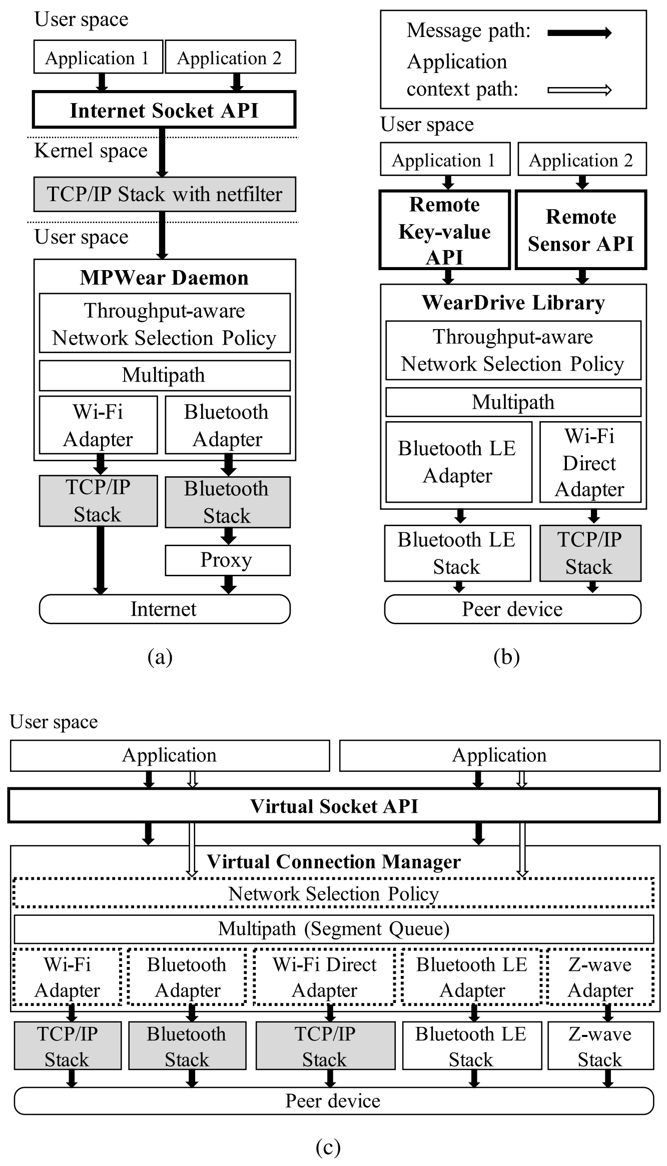


Fig. 1. Comparison of selective connection systems: (a) MPWear [5]; (b) WearDrive [8]; (c) proposed virtual connection system. The arrows between boxes denote the direction of message transfer. The white boxes are user-level components whereas the grey boxes are kernel-level components. The boxes with thick borders denote the communication APIs. The boxes with dashed borders denote programmable components.

and remote sensor, which are not compatible with the standard Internet socket API. Therefore, WearDrive must be modified to provide additional API functions to support other applications. If a selective connection system provides a standard socket-like API, various applications can be supported without modifying the system.

Lastly, many wearable consumer electronics are equipped with various P2P network interfaces such as WFD, Bluetooth LE, and Z-wave. To support a new P2P network interface, the selective connection system must be programmable to add a network adapter easily. In addition, the network selection policy also must be programmable to adapt it to various network environments and applications. However, most of the previous techniques provide no programming APIs to add a new network adapter or a new network selection policy. Therefore, the supported network adapters and the network selection policy are fixed.

In summary, selective connection systems for P2P-based wearable consumer electronics have four requirements.

- *Workload Prediction:* The network selection policy must be able to predict the future workload to select the network interface considering the high network switching cost of P2P network interfaces.
- *Remote Adapter Control:* The network switching function is required to control the P2P network adapters on remote peer devices.
- *Socket-Like User-Level API:* The selective connection systems must provide applications with socket-like user-level communication API to get messages directly from various applications.
- *Programmability:* The selective connection systems must be programmable to support various network adapters and network selection policies.

In this article, to meet the aforementioned requirements, we propose a *virtual connection* system that comprises *virtual socket API* and *virtual connection manager*, as shown in Fig. 1(c). The virtual connection API enables applications to describe *what messages to transfer* whereas the virtual connection manager determines *how to transfer* the messages.

The virtual socket API completely decouples applications from the network connection operations so that the applications can send or receive messages without considering the underlying network connections. Since the virtual socket API is a user-level API, the virtual connection system can hook any user message and forward it to the selected user-level P2P network service daemon. The virtual socket API is also designed to be similar to the Internet socket API, and thus various applications can use it. It also provides an additional function to enable applications to pass their contexts to the virtual connection manager. The network selection policy in the virtual connection manager predicts the future workload with the user context.

The virtual connection manager is a user-level daemon that provides all the required functions of selective connection to support P2P network devices. It provides programming interfaces to add new network adapters and network selection policies. It can also control the remote network adapters through remote control messages.

Compared to our earlier work [7], the proposed virtual connection system is designed with the consideration of the characteristics of P2P network devices. The virtual connection system additionally provides the socket-like user-level API to support various applications. In addition, new network adapters and network selection policies can be easily added to the virtual connection system via the programming interfaces.

In the case study presented in this article, we demonstrate how the selective connection can be easily implemented through the virtual connection system in wearable consumer electronics which have Bluetooth and WFD interfaces. We implemented a context-aware network selection policy that can predict the future network traffic based on the application context. We show that the context-aware network selection policy can reduce the communication energy consumption of wearable consumer electronics by up to 2.31 times.

II. Background and Related Work

A. Characteristics of P2P Networks

Some P2P networks support power save modes (PSMs) to reduce energy consumption during idle time. For example, Bluetooth supports a sniff mode [16]. Wi-Fi Direct (WFD) can use the PSM of classic Wi-Fi by the help of additional PSM techniques for WFD group owners [17]. Device drivers usually detect the idleness of a network interface and put the interface into a PSM without disconnecting the user-level connections. The device driver of WFD in PSM can awaken the network interface if there are pending messages, whereas the driver cannot awaken the network interface in the sleep mode.

Each P2P network shows a different energy efficiency in PSM. For example, Bluetooth sniff mode is several times more energy-efficient than the PSM of Wi-Fi or WFD [6], [14]. Therefore, if there is no data transmission through the network connection for a while, it is more energy-efficient to destroy WFD connection rather than to use the PSM.

Meanwhile, WFD has a high network switching cost. Especially, establishing WFD connection takes a long time and shows high power consumption. Even if a WFD interface uses the autonomous group formation method, which has the shortest connection latency, it must perform several operations such as discovery, Wi-Fi protected setup (WPS) provisioning, and address configuration to establish WFD connection [17]. For example, our measured latency for WFD connection between an embedded board and a smartphone was about 5.4 seconds. In addition, establishing WFD connection consumes about 13.2 times more power compared to the Bluetooth idle state power consumption. Therefore, to reduce the overall energy consumption of selective connection systems, the high-cost WFD connection must be chosen only when it is predicted that many messages will be transferred over a long period.

B. Selective Connection Systems

There are several researches to support multiple heterogeneous network interfaces at consumer electronics. For example, the protocol adaptation layer (PAL) between Bluetooth and ultra-wideband (UWB) has proposed for high definition video transmission [18]. However, it has a strong dependency on the specific network interfaces requiring significant modifications in both hardware and software. Since it does not support automatic network selection, applications must make explicit network selection commands to change the network interface.

To solve such a problem of manual network selection, several selective connection systems have been proposed [5]–[8]. These systems are compared with our virtual connection in Table I. MPWear [5] is a selective connection system to support smartwatch applications connected to the Internet via Wi-Fi or Bluetooth with a minimal energy consumption. CoolSpots [6] enables wireless mobile devices to connect to access points through either Wi-Fi or Bluetooth to save power.

TABLE I
COMPARISON OF SELECTIVE CONNECTION SYSTEMS

Feature	MPWear [5]	CoolSpots [6]	WearDrive [8]	Virtual Connection
Network selection	✓	✓	✓	✓
Network switching	✓	✓	✓	✓
Seamless handover	✓	✓	✓	✓
Workload prediction				✓
Remote adapter control			✓	✓
User-level API			✓	✓
Proprietary API			✓	
Programming interfaces				✓

Network Virtualization [7] is our prior work, which implements a selective communication between IoT devices via Bluetooth or Wi-Fi Direct. WearDrive [8] is a remote storage framework for wearable devices, which communicates with the paired smartphone via Bluetooth LE or WFD.

As shown in Table I, the previous selective connection systems provide all the required functions: network selection, network switching, and seamless handover. However, they did not consider the characteristics of P2P networks. For example, the network switching functions in MPWear and CoolSpots do not provide the remote adapter control, so they cannot activate or deactivate P2P network interfaces on the peer device. This is because they target Internet-connected devices. In addition, MPWear and CoolSpots get communication messages by hooking them from the kernel-level network protocol stack, and thus it is difficult to apply them to P2P devices which use various application-level protocol stacks.

In contrast, WearDrive can be applied to the P2P devices since it provides the remote adapter control and the user-level API. However, WearDrive uses only the proprietary API functions designed for specific applications. For example, WearDrive provides `InsertKV()`, `ReadKV()`, and `DeleteOldData()` for the remote key-value storage applications. It also provides `RegisterForSensor()` for the remote sensor storage applications.

Since the network selection functions in the previous techniques [5]–[8] do not support the workload prediction, they cannot consider the high network switching cost of P2P networks. Besides, they also do not provide any programming interface functions to add network selection policies and network adapters.

Unlike the previous techniques, our proposed virtual connection is designed with considering the characteristics of P2P networks. For the purpose, in the virtual connection, the network selection function provides the workload prediction, and the network switching function supports remote adapter control. To minimize the modifications on P2P network protocol stacks, the virtual connection communicates with user applications via user-level API. Since the virtual connection API is socket-like, it can support various applications without significant changes of applications. In addition, a new network adapter can be easily integrated to the virtual connection and it is easy to implement a new network selection policy.

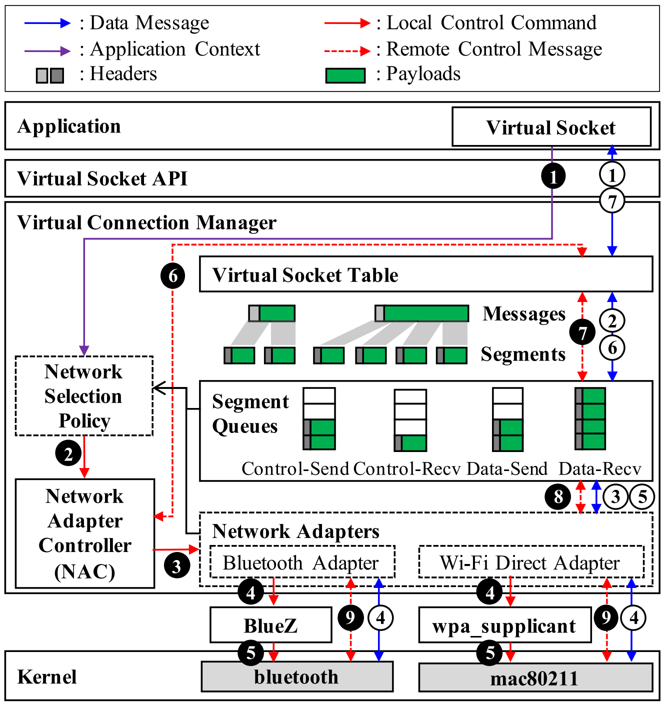


Fig. 2. Architecture of the virtual connection system. The boxes with dashed borders denote programmable modules, and the boxes with solid borders denote non-programmable modules. The white boxes denote user-level modules whereas the grey boxes denote kernel-level modules.

III. OVERALL ARCHITECTURE

The overall architecture of the virtual connection system is shown in Fig. 2. It consists of the virtual socket API and the virtual connection manager. Basically, the *virtual socket API* allows applications to send or receive messages to or from a peer device through any wireless network. More details of the virtual socket API will be presented in Section IV.

The *virtual connection manager* selects the most suitable network connection and controls the activation of underlying network connections. Because the virtual connection manager is a user-level daemon, the network adapters can communicate with other user-level network daemons such as BlueZ [15] or wpa_supplicant [19] through IPC. Detailed descriptions of the virtual connection manager will be given in Section V.

A. Data Transmission

The virtual connection allows applications to send or receive messages to or from their peer devices without being aware of the underlying network connection. As shown in Fig. 2, the application first opens a virtual socket and sends messages to the virtual connection manager through the *virtual socket API* (①). Each message is split into fixed size segments, which are inserted into the segment queues (②). The *segment queues* are used for the seamless handover between different network interfaces. Each segment can be sent through different *network adapters*, depending on the current connection status (③). The network adapter sends the segments to the peer device through the network-specific kernel-level modules or user-level daemons (④). For example, the Bluetooth network adapter sends segment transmission requests to the kernel-level Bluetooth module through the RFCOMM [16] socket API. The WFD network adapter sends the segment transmission request to the mac80211 module via the Internet socket API. For the BLE transmission, segment transmission requests are sent to BlueZ [15] supporting GATT profile [20].

For incoming segments through a network connection, *the network adapter* puts the segments into the *segment queue*, where the split segments are reassembled into the original message (⑤). Owing to the message assembly function of the segment queue, the segments of a message can be transferred via different network adapters, and thus the seamless handover can be supported (⑥).

The incoming messages must be sent to the corresponding application. Each message has a virtual socket name in its header field. From *the virtual socket table*, the virtual connection manager can find which application is associated with the virtual socket. The virtual connection manager sends the message to the target application process through IPC (⑦).

B. Network Connection Control

After launching the virtual connection manager, the *network adapter controller* (NAC) establishes a connection through the most power-efficient network interface such as Bluetooth. After that, the NAC can change the network connection in response to the request of the network selection policy.

As shown in Fig. 2, an application can pass its context as a hint to the network selection policy through the virtual socket API (❶). The *network selection policy* selects a network that can achieve the minimum energy consumption based on the application context and system status. When the network connection has to be changed, the NAC makes a new connection in the selected network and closes the connection of the currently connected network (❷). To control the network connections, the NAC issues control commands to the local network adapters (❸). Each local network adapter controls its own network connections through the associated network daemons (❹) and kernel-level drivers (❺). The NAC also sends a remote control message to the remote network adapter on the peer device in the same way the data message is delivered to the peer device (❻-❾). When the NAC on the peer device receives the remote control message, it changes the states of network interfaces to be activated or deactivated through local control commands (❸-❺).

IV. Virtual Socket API

Table II shows the list of virtual socket API functions. The virtual socket API includes several functions to send or receive messages to or from the peer device through a virtual socket. To make a connection with a peer device, applications must first open a virtual socket by calling *open()*. The virtual socket is an endpoint for sending or receiving messages in P2P networks. A virtual socket is identified by its name, which is specified by the application. In the example of Fig. 3, a camera streaming application creates a virtual socket with the name of `video_frame`. Then, it calls *send()* to send a message to the application running on the peer device or *receive()* to receive a message. In the example, a video frame is sent to the peer device through *send()*. The virtual socket API is designed to be similar to the existing Internet socket API functions. The only difference is that a virtual socket name is used instead of an IP address as a socket's identifier.

TABLE II
List of Virtual Socket API Functions

Function	Description
VirtualSocket open (String virtual_socket_name)	Open a virtual socket
int send (VirtualSocket s, byte[] message)	Send a message to peer device's virtual socket
int receive (VirtualSocket s, byte[] message)	Receive a message from peer device's virtual socket
void send_application_context (VirtualSocket s, String context)	Send an application context to network selection policy

```
VirtualSocket s = open("video_frame");
send_application_context(s, "on-streaming");
byte[] video_frame = get_video_frame();
int result = send(s, video_frame);
```

Fig. 3. An example of virtual socket API usage in a video streaming application.

To enable the network selection policy to predict the future network traffic, the virtual socket API also provides the function of *send_application_context()*, through which applications can pass their contexts to the network selection policy in the virtual connection manager. Applications tend to generate similar patterns of network traffic depending on the *application context*. In this article, the application context represents the application mode initiated by a user input event. For example, the News application [3] on a smartwatch generates a burst of network traffic to download a list of articles and thumbnail pictures when its refresh button is clicked. When the hyperlink of each article is clicked, a different pattern of network traffic is generated to download the article's contents and photos. In the virtual connection system, applications can pass their contexts to the underlying virtual connection manager. Because the Internet socket API functions used by MPWear [5] and CoolSpots [6] call the kernel system calls, many modifications are required in both user space and kernel space to add the context passing function. In contrast, the virtual socket API is at the user level, so it is not difficult to add the hint-passing function.

In the example shown in Fig. 3, the application sends the `on-streaming` context to the network selection policy. While the application runs, the network selection policy profiles the network traffic in this context. The profiling result is used by the network selection policy to predict future network traffic when the application re-enters the context.

V. Virtual Connection Manager

A. Segment-Based Message Transfer

As shown in Fig. 2, each message is split into multiple segments, each of which can be transmitted through different network connections for the *seamless handover*.

The messages are categorized into two types: data messages and remote control messages. Whereas data messages are made by applications, remote control messages are generated by the NAC to control the network adapters on peer devices.

When the virtual connection manager receives a user message via *send()*, it adds a message header that contains the length of the message payload and the target virtual socket. The length of the message payload is used to reassemble the original message from its segments at the peer device. The target virtual socket is used to identify the virtual socket on the peer device where the message is delivered.

The segment is the minimum data unit that can be transferred through a network connection. Each segment has a segment header that contains the length, type, and more-segment-flag of the segment. The segment type can be control segment or data segment, and is determined according to the message type. The more-segment-flag indicates whether it is the last segment in a message or not, similar to the more-fragment-flag in IP [21].

The virtual connection manager maintains four segment queues: control-send, control-receive, data-send, and data-receive. The segment queues for control segments are separated from the data segment queues in order to give a higher priority to the control segments over the data segments.

During a network switching, the previous network adapter continues to transfer the segments until a new network adapter is connected. Once the new network adapter starts to handle segments, the previous network adaptor is deactivated.

B. Network Selection Policy

The *network selection policy* module retrieves the current system information such as the network throughput of each network adapter and the length of each segment queue. It can also receive the context of the current application. Based on this information, the network selection policy decides whether the network connection must be changed or not. If the network connection must be changed, the network selection policy sends a network switching request to the *network adapter controller (NAC)* and the NAC controls the network adapters.

The virtual connection system provides a programming interface for the network selection policy, which includes a callback function, *select_network()*. This function, which is periodically called by the virtual connection system, will select the most appropriate network considering the system status and the application context. The system developer can change the network selection policy only by changing the function.

C. Network Adapters

The *network adapters* are responsible for controlling each network connection and transferring segments over the network connection. As shown in Table III, each network adapter must have the control and segment transmission functions of the network adapter interface.

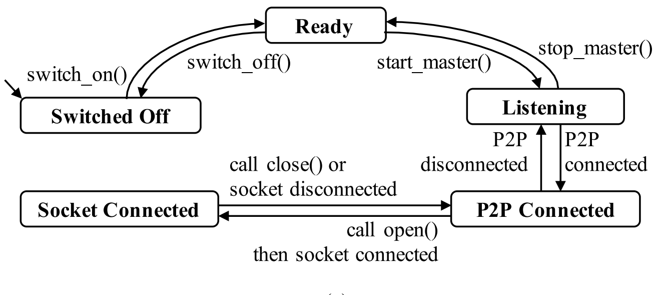
As shown in Fig. 2, when the NAC receives a network switching request from the network selection policy, the NAC sends local control commands to the target local network adapter and the network adapter calls its own corresponding control functions. To change the power mode of the remote network interface on the peer device, the NAC also sends remote control messages to the peer device. When the NAC of the remote device receives the remote control messages, it calls its own control functions.

The virtual connection system manages the state change of each network adapter, as shown in Fig. 4. There are six different network adapter states: *switched-off, ready, listening, discovering, P2P-connected*, and *socket-connected*. We define a device that advertise itself as a *master device*, whereas *slave device* is defined as a device that discover the master device.

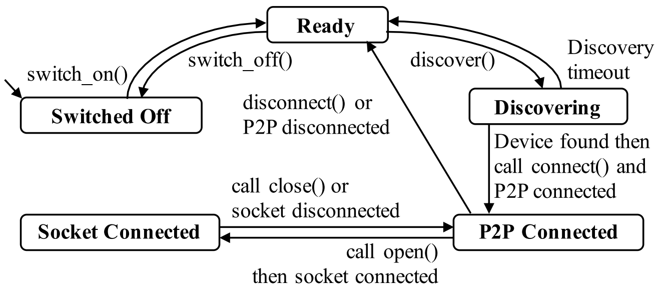
Initially, each network adapter is in the *switched-off* state, indicating that the adapter is not switched on. When the virtual connection manager is launched, the NAC calls the *switch_on()* functions of all the network adapters and the adapters transit to the *ready* state.

TABLE III
Function List of Network Adapter Interface

Category	Master	Slave
Device Control	switch_on(), switch_off()	
P2P Connection Control	start_master(), stop_master()	discover(), connect(), disconnect()
Socket Connection Control	open(), close()	
Segment Transmission	send(), receive()	



(a)


Ready
switch_on()
switch_off()
discover()
Discovery timeout
Switched Off
disconnect() or P2P disconnected
Discovering
Device found then call connect() and P2P connected
call close() or socket disconnected
Socket Connected
P2P Connected
call open() then socket connected

(b)

Fig. 4. State transition diagram of network adapter (a) on master device and (b) on slave device.

To activate a network connection, the NAC calls several control functions of the network adapter and changes the state of the network adapter as follows: First, *start_master()* on the master device or *discover()* on the slave device is called. After that, the network adapter on the master device listens for discovery requests from the slave device in the *listening* state. The network adapter on the slave device starts to discover the master device in the *discovering* state. After the slave device finds a master device, it calls *connect()* to make a P2P connection. If the P2P connection is successfully established, the network adapters on the master and slave devices reach the *P2P-connected* state. When a socket connection between the peer devices is established by *open()*, the network adapters attain the *socket-connected* state.

If there is no message traffic, network adapters need to put the network interface into a PSM or sleep mode. However, due to the high power consumption of WFD in PSM, we change the WFD interface into the sleep mode. Therefore, WFD interface enters *ready* state after both the socket and P2P connections are closed by the *close()*, *disconnect()*, and *stop_master()* functions. On the other hand, when Bluetooth is not used, the Bluetooth adapter must transit to the *P2P-connected* state by closing the socket connection through *close()* function. This is because many Bluetooth device drivers support the sniff mode, but maintaining socket connections of both Bluetooth and WFD adapters causes interference between the adapters while degrading the network bandwidth.

The segment transmission functions, *send()* and *receive()*, of the network adapter are called by two background threads, a sender thread and a receiver thread. The sender thread pops a segment from the send segment queues and calls *send()* to send it. The receiver thread calls *receive()* to receive a segment and pushes it to the receive segment queues.

VI. Case Study: Context-Aware Network Selection Policy

Previous selective connection systems such as MPWear [5], CoolSpots [6], and Network Virtualization [7] use a *throughput-aware policy* as their network selection policy, in which a network is selected based on the present network throughput. However, the throughput-aware policy does not consider the high network switching cost of P2P networks. In this article, we propose a *context-aware policy* that predicts the future network traffic based on the application context. We implemented the throughput-aware policy [5] and the context-aware policy using the proposed network selection policy interface.

Fig. 5 shows an example implementation of the context-aware policy. The function *select_network()* receives the network system status (`st`) and the application context (`ctx`) as input parameters. The context-aware policy predicts the future network traffic (`tf`) by referring to the network traffic profiling result. The waiting segments (`st.waiting_segments`) in the segment queues are also added to the future network traffic.

Based on the predicted future network traffic, the network selection policy estimates the future energy consumption

```
Selection select_network (SystemStatus st, String
     ctx) {
  // Step 1. Predict future network traffic
  tf = predict_traf(ctx, st.waiting_segments);
  // Step 2. Estimate energy consumption
  energy_bt = estimate_energy(tf, st, BT);`
  energy_wfd = estimate_energy(tf, st, WFD);
  // Step 3. Decide whether to switch networks
  if (st.adapters.BT.is_connected()) {
    if (energy_bt > energy_wfd) return WFD;
    else return BT;
  } else {
    if (energy_bt < energy_wfd) return BT;
    else return WFD;
  }
}
```

Fig. 5. An example of a context-aware network selection policy.

TABLE IV
CHARACTERISTICS OF THE TEST APPLICATIONS OF WEARABLE CONSUMER ELECTRONICS USED FOR THE EXPERIMENTS

Name	Idle time (%)	Peak request B/W (KB/s)	Avg. request B/W (KB/s)
Applications on Smartwatches			
Web browser (Web) [4]	73.79	448.42	23.74
Map [2]	55.05	389.32	63.14
News [3]	75.47	1399.39	80.81
Applications on Smartglasses			
Video streaming (VS) [22]	2.11	48.71	30.37
Remote storage (RS) [22]	87.02	1236.59	81.30

for different network connections. A linear regression-based power model is used to predict the power consumption for different network throughput values and the connection status of the network adapters. The network selection policy selects the network that will consume the minimum energy.

VII. Evaluation

A. Experimental Setup

1) Application Workload Trace: For experiments, we collected several application workload traces from two types of wearable consumer electronics, i.e., smartwatch, and smartglasses, as shown in Table IV. We used a commercial smartwatch equipped with a dual-core mobile CPU and 512 MB memory. It has 802.11-n and Bluetooth 4.0 network interfaces. We downloaded three applications [2]–[4] from an on-line application market and ran them respectively on the smartwatch for 250 seconds.

For the smartglasses, we implemented our own prototype by using an embedded board equipped with a quad-core mobile CPU and 1 GB memory. The smartglasses prototype has an embedded radio module with 802.11-n support, and it has a Bluetooth 4.0 adapter attached via a USB Bluetooth dongle. For the smartglasses prototype, we implemented two companion applications [10], video streaming (VS) and remote storage (RS) [8], based on a companion device software platform [22]. For experiments, VS sent the packets of a 240p resolution video continuously for 360 seconds, and RS sent 60 image files, of which size is 327 KB on average, over 360 seconds.

We executed the two applications simultaneously on the smartglasses prototype. The wearable devices were connected to a commercial smartphone with a quad-core mobile CPU and a 4 GB memory through Bluetooth or Wi-Fi Direct connections.

An application workload trace includes the sequence of network packets and application contexts. The network packets were obtained by sniffing packets transmitted over wireless networks from the target devices through a widely used network protocol analyzer, Wireshark [23].

We also defined the types of application contexts for each application and added the contexts to the application traces. For example, the Web browser application (Web) has the context of *navigate-page*. The news application has the contexts of *refresh-article-list* and *load-article-contents*. The map application has the contexts of *panning*, *zoom-in*, and *zoom-out*. The video streaming application uses *on-streaming* as its application context, and the remote storage application has the contexts of *file-read* and *file-write*.

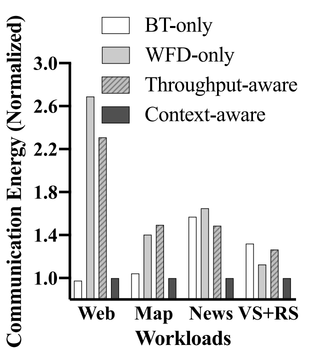
2) Testbed System: The collected traces were replayed in a testbed system to evaluate the effect of the virtual connection. The testbed system is same to that used for workload tracing. In the experiment, the smartwatch or smartglasses played the role of the master device, and the smartphone was the slave device. We implemented four different network selection policies: BT-only (Bluetooth-only), WFD-only, throughput-aware [5], and our context-aware policy. The BT-only policy used only the Bluetooth adapter, and the WFD-only policy used only the WFD adapter. We have made the source code of the virtual connection for the testbed system publicly available at https://github.com/SKKU-ESLAB/Virtual-Connection.

B. Communication Energy Consumption

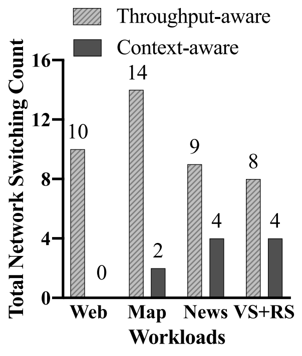
Fig. 6(a) shows a comparison of the communication energy consumption for the different network selection policies. Throughout all the workloads, the context-aware policy showed the minimum energy consumption with 1.26-2.31 times energy savings compared to the throughput-aware policy. This energy saving was mainly due to the future traffic prediction scheme of the context-aware policy. Because the throughput-aware policy considered only the current network traffic, on average, it invoked network switching 2.25 times compared to that invoked by the context-aware policy, as shown in Fig. 6(b). The excessive network switching resulted in higher energy consumption.

In the Web browser and map workloads, which had low peak request bandwidths, the virtual connection selected the Bluetooth connection most of the time. Therefore, the context-aware and the BT-only policies showed similar results.

Although Bluetooth generally shows lower power consumption than WFD, the BT-only policy consumed more energy than the WFD-only policy for the smartglasses workload (VS+RS). As shown in Table IV, the smartglasses workload consists of the video streaming application generating low network traffic and the remote storage application generating burst network traffic. Therefore, when the two applications ran simultaneously, the BT-only policy showed longer communication time and a large amount of energy consumption. In contrast, the context-aware policy always showed better results than the other policies for all the workloads by dynamically selecting the most efficient network connection depending on the change in request traffic.

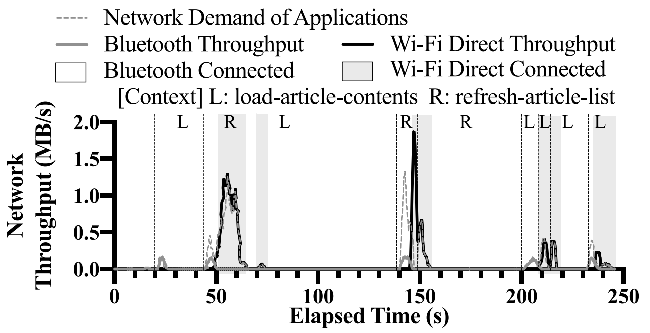


(a)

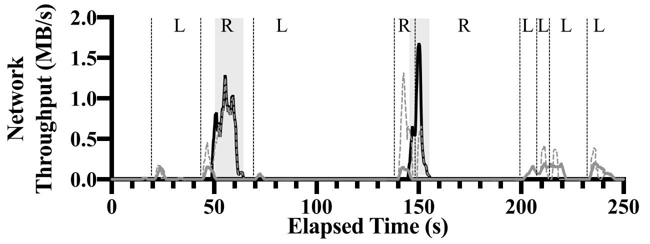


(b)

Fig. 6. (a) Communication energy consumption; (b) total network switching count for each application. The communication energy consumption is normalized to the energy consumption of the context-aware policy.



(a)



(b)

Fig. 7. Network throughput when the News application [3] ran with different network selection policies: (a) throughput-aware; (b) context-aware.

C. Network Switching

As shown in Fig. 7, we measured the variance in network throughput while running the news application workload. During the periods of 70-75 s, 210-220 s, and 235-250 s, the application tries to send small amount of messages but it exceeded the maximum Bluetooth bandwidth. The throughput-aware policy selected WFD whereas the context-aware policy did not switch the network connection. Because the throughput-aware policy monitored only the instantaneous network throughput, it did not consider the total network traffic over a long period when switching networks. On the other hand, the context-aware policy selected the network based on the application context, i.e., Bluetooth in *load-article-contents* context and WFD in *refresh-article-list* context. As a result, the context-aware policy reduced the number of network switching.

VIII. Conclusion

In this article, we proposed a novel selective network connection system for wearable consumer electronics called virtual connection that dynamically changes the network interface, to minimize the communication energy consumption. The virtual connection provides a virtual socket API that allows applications to use the selective connection system without considering the network connection details. Because the virtual connection manager is a programmable user-level daemon, a system designer can easily modify the network selection policy and add new network adapters. To demonstrate the virtual connection system, we implemented a context-aware network selection policy for wearable consumer electronics using P2P communications and showed that the context-aware policy resulted in an energy saving of up to 2.31 times compared to the policies of previous selective connection systems.

References

[1] W.-R. Yang, C.-S. Wang, and C.-P. Chen, "Motion-pattern recognition system using a wavelet-neural network," *IEEE Trans. Consum. Electron.*, vol. 65, no. 2, pp. 170–178, May 2019.

[2] Naver. (2019). *Naver Map*. Accessed: Aug. 22, 2019. [Online]. Available: https://galaxystore.samsung.com/geardetail/GjEbuFc12C

[3] Samsung Electron. (2019). *News Briefing*. Accessed: Aug. 22, 2019. [Online]. Available: https://galaxystore.samsung.com/geardetail/com.samsung.w-magazine-wc1

[4] C. Jeon. (2019). *Gear Browser*. Accessed: Aug. 22, 2019. [Online]. Available: https://galaxystore.samsung.com/geardetail/com.fin10.tizen.gearbrowser

[5] X. Zhu, Y. E. Guo, A. Nikravesh, F. Qian, and Z. M. Mao, "Understanding the networking performance of wear OS," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 1, pp. 1–25, Mar. 2019.

[6] T. Pering, Y. Agarwal, R. Gupta, and R. Want, "CoolSpots: Reducing the power consumption of wireless mobile devices with multiple radio interfaces," in *Proc. 4th ACM Int. Conf. Mobile Syst. Appl. Services (MobiSys)*, Jun. 2006, pp. 220–232.

[7] I. Hwang and D. Shin, "Application level network virtualization using selective connection," in *Proc. IEEE 36th Int. Conf. Consum. Electron. (ICCE)*, Jan. 2018, pp. 1–2.

[8] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale, "WearDrive: Fast and energy-efficient storage for wearables," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2015, pp. 613–625.

[9] M. Golkarifard, J. Yang, Z. Huang, A. Movaghar, and P. Hui, "Dandelion: A unified code offloading system for wearable computing," *IEEE Trans. Mobile Comput.*, vol. 18, no. 3, pp. 546–559, Mar. 2019.

[10] T. Braud, P. Zhou, J. Kangasharju, and P. Hui, "Multipath computation offloading for mobile augmented reality," in *Proc. IEEE Int. Conf. Pervasive Comput. and Commun. (PerCom)*, Mar. 2020, pp. 1–10.

[11] L. Rachakonda, S. P. Mohanty, and E. Kougianos, "iLog: An intelligent device for automatic food intake monitoring and stress detection in the IoMT," *IEEE Trans. Consum. Electron.*, vol. 66, no. 2, pp. 115–124, May 2020.

[12] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu, "DeepWear: Adaptive local offloading for on-wearable deep learning," *IEEE Trans. Mobile Comput.*, vol. 19, no. 2, pp. 314–330, Feb. 2020.

[13] H. Welte, "The netfilter framework in Linux 2.4," in *Proc. Linux Kongress*, 2000, pp. 8–10. Accessed: Apr. 7, 2020. [Online]. Available: http://archil.fr/Doc_NetFilter/netfilter.pdf

[14] Cypress Semiconductor. *CYW43438 Datasheet*. Accessed: Apr. 7, 2020. [Online]. Available: https://www.cypress.com/file/298076/download

[15] M. Krasnyansky and M. Holtmann. (2003). *BlueZ: Official Linux Bluetooth Protocol Stack*. Accessed: Apr. 7, 2020. [Online]. Available: http://www.bluez.org

[16] C. Bisdikian, "An overview of the bluetooth wireless technology," *IEEE Commun. Mag.*, vol. 39, no. 12, pp. 86–94, Dec. 2001.

[17] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano, "Device-to-device communications with Wi-Fi direct: Overview and experimentation," *IEEE Wireless Commun.*, vol. 20, no. 3, pp. 96–104, Jun. 2013.

[18] Y. Jeon, S. Lee, S. Lee, S. Choi, and D. Y. Kim, "High definition video transmission using Bluetooth over UWB," *IEEE Trans. Consum. Electron.*, vol. 56, no. 1, pp. 27–33, Feb. 2010.

[19] *Linux WPA/WPA2/IEEE 802.1X Supplicant*. Accessed: Apr. 7, 2020. [Online]. Available: http://hostap.epitest.fi/wpa_supplicant

[20] *Bluetooth Core Specification 4.0*, Bluetooth SIG, Kirkland, WA, USA, Jul. 2010.

[21] B. A. Forouzan, *TCP/IP Protocol Suite*, 4th ed. New York, NY, USA: McGraw-Hill, 2006.

[22] H. Lee, D. Sin, E. Park, I. Hwang, G. Hong, and D. Shin, "Open software platform for companion IoT devices," in *Proc. IEEE 35th Int. Conf. Consum. Electron. (ICCE)*, Jan. 2017, pp. 394–395.

[23] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal Network Protocol Analyzer Toolkit*, Rockland, MA, USA: Syngress, 2006.

Gyeonghwan Hong received the B.E. degree in computer engineering from Sungkyunkwan University, Suwon, South Korea, in 2013, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering. His research interests include embedded software, mobile system software, and Internet of Things.

Dongkun Shin (Member, IEEE) received the Ph.D. degree in computer science and engineering from Seoul National University, Seoul, South Korea, in 2004. He is currently a Professor with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, South Korea. From 2004 to 2007, he was a Senior Engineer with Samsung Electronics, South Korea. His research interests include embedded software, low-power systems, computer architecture, and real-time systems.