Ambient Virtio: IO Virtualization for Seamless Integration and Access of Devices in Ambient Computing

Seongjoon Park, Kiseok Kim, and Hwangnam Kim, *Member, IEEE*

***Abstract*—Ambient computing creates a digital environment where a variety of technologies, including hardware, software, and even machine learning, are integrated into most useful and autonomous digital device that we want to use right here right now. The main obstacle for this computing is a barrier of vendor-specific details, programming interfaces, and access procedure even though all the things come to have network interfaces and interact with each other through the Internet. However, this configuration defies the expectation that ambient computing can reduce the demand for human attention. In this article, we propose *Ambient Virtio*, a device virtualization technology that virtualizes devices with the `virtio` framework. This allows devices in close proximity to users to be accessed without human intervention and integrated into metadevices that users want to create and access on the fly. We newly defined a `virtio`-based backend device named `virtio-ambient` and also constructed an ambient device architecture for virtual machines to read and write the remote device only with built-in system calls. We completely implemented the system and performed comparative evaluation with the existing application-level data networking system.**

***Index Terms*—Ambient computing, device virtualization, VirtIO.**

I. Introduction

OVER the past few years, a lot of technology companies have been working to tightly incorporate many technologies and computing platforms into one digital environment. One of their resulting goals is to assimilate devices and computers into the environment to the point where we can use any computing power for services that we want to use immediately. This is called *ambient computing*, and it is based on the proliferation of IoT devices.

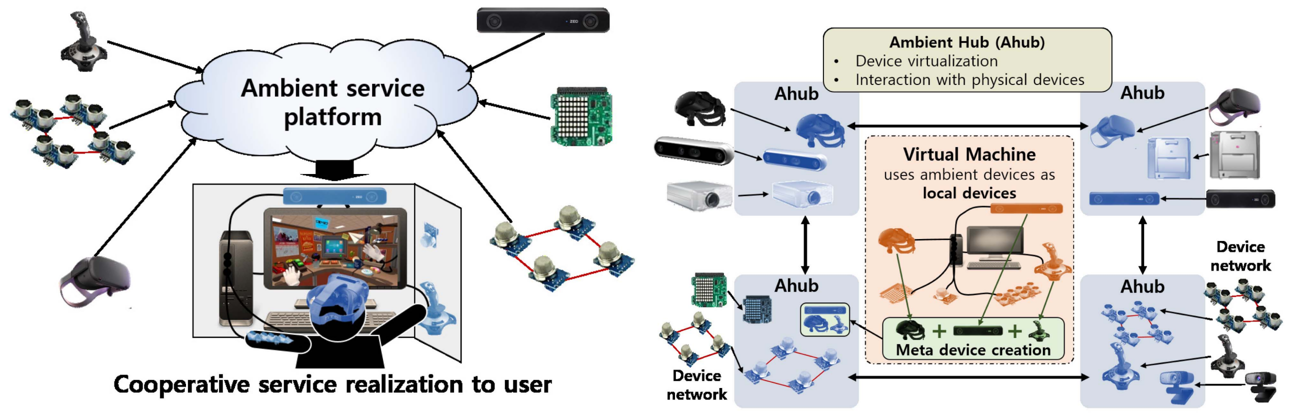
Considering that highly sophisticated IoT devices are pervasive in our lives today and these are connected to the world for exchanging useful data for the convenience of users, such an ambient computing is likely to develop further in the future [1]. However, even though the realization of this blueprint for ambient computing seems imminent, there are several barriers to building an open digital world where all computing platforms are connected. The main cause of the barriers comes from vendor-specific details; IoT vendors release products with their own device drivers and protocols to promote their proprietary systems made up of their own components. Such a strategy is natural from an economic or security point of view, but it makes difficult to integrate all components around the user into a unified computing platform, which lowers the extensibility of the ecosystem for the product. This can be approached with a concept of connectivity network, routing-based networking by connecting segmented devices, and/or networks using network or networking-based system solutions, but, on the other hand, we can also think of ways to build an integrated computing platform for users to access and control devices over a single interface by virtualizing heterogeneous edge devices at device level. For example, assuming that a host machine has a locally equipped edge device, an application that utilizes it can simply access and manage edge devices using the application programming interfaces (APIs) provided by the operating system or third party. However, when remotely connected, the application requires more complex considerations about the network fabric through which data will pass. In this regard, we propose a novel mechanism for providing "*seamless ambience*" for virtual machine (VM) users by eliminating the hassle in application layer and the network overhead experienced by VM through unified encapsulation for both local and remote devices.

For doing so, we expanded a para-virtualization scheme called `virtio`. The `virtio` manages requests and responses in a ring-shaped structure called `vring` to read or write device data. However, existing `virtio` has a limitation in that multiple kinds of submodules should be provided for different types of device. For example, publishing a new device requires users to create a separate binary translation that implements a driver with a new way of operation and allows it to run on a host machine. Given that heterogeneous IoT devices residing in edge environments are used by physically remote VM infrastructures, it means that a functional and design basis for supporting specific edge devices must already be in place. However, it is not easy to meet an acceptable level of structural overhead to cope with a large number of heterogeneous devices in various

Manuscript received 27 January 2022; revised 8 June 2022; accepted 30 July 2022. Date of publication 22 August 2022; date of current version 8 June 2023. This work was supported in part by the Human Resources Program in Energy Technology of the Korea Institute of Energy Technology Evaluation and Planning (KETEP) and the Ministry of Trade, Industry and Energy (MOTIE) of the Republic of Korea under Grant 20204010600220 and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) under Grant 2020R1A2C1012389. *(Seongjoon Park and Kiseok Kim contributed equally to this work.) (Corresponding author: Hwangnam Kim.)*

The authors are with Electrical Engineering, Korea University, Seoul 02841, Republic of Korea (e-mail: psj900918@korea.ac.kr; kisuk528@korea.ac.kr; hnkim@korea.ac.kr).

Digital Object Identifier 10.1109/JSYST.2022.3196624



(a) (b)

Fig. 1. Ambient Virtio system design concept. The left shows how the system provides ambient services to user, and the right shows the detailed diagram of the system. (a) How the system provides ambient services to users. (b) System detailed diagram.

ambient environments. Separately, there are various approaches for enabling users to access remote resources via the network. Most representative solution is desktop remote control services, such as Chrome Remote Desktop [2] or TeamViewer [3]. Those solutions enable to exchange the input events of input devices (mainly mouse and keyboard) and the output events of display devices (mainly monitor). Second, wireless sensor network attempts to collect the data from multiple sensor devices through wireless communications [4]. We argue that these remote access technologies are *not ambient* since each requires different data, protocol, standards, and APIs to users. To overcome the limitations of the above and provide an ambient integrated computing platform for heterogeneous edge devices with conventional different interfaces and communication/control protocols, we propose device-level virtualization to establish networking between computing platforms in addition to device sharing or in-device resource sharing. With such device-level virtualization, users can access any device in their proximity without human interaction and even create a new metadevice that does not physically exist but virtually exist by combining multiple remotely located devices. Based on this motivation, we named our proposed system to *Ambient Virtio* that has been designed and implemented in the extended line of aforementioned `virtio` protocol. In specific, utilizing `virtio` protocol, we declare a new type of virtual device called `virtio-ambient` that interacts with the network environment and performs `read/write` operation as local device in the viewpoint of the VM guest. Here, the fact that the `virtio` interface is used as a standard technology for implementing device access in various virtual machines and shows high I/O performance between VM-to-host compared to other methods suggests a strong insight for device-level virtualization. Fig. 1 shows our concept briefly. In the proposed system, VM connects to the ambient service platform and provides the virtualized device resources to the user. As Fig. 1(a) implies, a user utilizes the service enabled by the cooperation of the ambient devices. The user does not need to prepare the detailed network specification for each device since any devices are seen as local device at the viewpoint of VM application. Each device is locally or remotely connected to an adapter called *Ambient Hub* (Ahub). Through the communication between VMs and Ahubs, the VM can read its available device lists and request the access via OS system call. We summarized the characteristics of our proposed system as follows:

1) *Standard OS:* We used Ubuntu 20.04 LTS desktop image for virtual machine without any modification. This image equips Linux kernel 5.4, which has had built-in `virtio` module for a long time. Thus, the proposed design is available for any VM whose kernel supports virtio above version 1.0 specifically. We confined the modification within the hypervisor, which is covered in detail in Section III-A.
2) *Unified device:* To drive the ease of use, we abstracted ambient devices into `virtio-blk` module. If the VM user activates `virtio-ambient` module, the hypervisor exposes a newly defined device based on `virtio-blk` to a VM and adds or deletes devices during runtime as needed. As a result, VM guest can treat any ambient devices as a local block device and do read/write operations. This mechanism not only helps the VM user to use the ambient device easily but also helps the simple development of *metadevice*, which is addressed in Section III-C.
3) *Access control:* Since the Ahub processes actual read/write operation, access control rights for users should be regulated and adjusted for the safety. We designed Ahub to have an access control mechanism that defines device accessibility for user groups and subdivides permissions into visibility, readability, writability, and configurability. Section III-D addresses how Ahubs configure the accessibility of ambient devices in detail.
4) *Total separation of network layer:* One of the key contributions of the proposed system is the complete separation of the network part from remote device access. The hypervisor and host machine are responsible for the entire networking process, allowing VM guests to use peripherals without complex network configuration. This comprehensive device networking also enables opportunities for comprehensive network resource management, improving the robustness of the overall system.

The rest of this article is organized as follows. Section II addresses the previous works in virtualization and virtio. Section III describes the details of Ambient Virtio, and Section IV shows the performance evaluation of the system. Finally, Section V concludes the article.

II. Preliminaries

This section introduces the researches on virtualization technology and virtio protocol which attempts to accelerate the device operation performance of virtual machine.

A. Virtualization

Virtualization refers to a solution that creates software-based IT services such as servers, storage, and networks by abstracting the computing resource which is dependent on hardware. Because most traditional enterprises used physical servers and single-vendor IT stacks, legacy applications could not run on some hardware from other vendors. Virtualization enables legacy applications to run on separated operating systems; so the user can increase server efficiency and reduce infrastructural costs. Virtualization software, called a *hypervisor*, abstracts physical resources into a virtual environment while running on physical machine. When an application running in a virtual environment requires additional resources, the hypervisor forwards the request to the physical interface and caches the state change. For instance, hypervisor processes a request sent from a kernel virtual machine (KVM) [5] in the kernel context.

We briefly list virtualization technologies according to the service target. *Hardware virtualization* creates a virtual machine which runs a set of software with its own architecture, storage, memory, and computing power. *Desktop virtualization* provides a separated desktop environment to the user. Through virtual desktop interface, user can simultaneously deploy an emulated desktop environment to a number of physical machines and concurrently manage the systems. *Operating system virtualization*, also known as *containerization*, provides an isolated user space instance while sharing the same architecture such as arm or x86. The proposed system can be considered as hardware virtualization, specifically virtualization and abstraction of both remote and local devices. To emulate the physical hardware, existing virtualization techniques defined various types of virtual devices. However, vendor dependency raises again at the peripheral devices; differences in device connection protocols, physical ports, and bandwidth constraints pose challenges for VM clients as well as users of physical machines. Specifically, containerization allows the direct use of physical devices and limits their own access to local devices. The philosophy of virtio-ambient is the virtualization of the interface of any devices, regardless of the method of access, even including remote access.

B. Virtio Protocol

Software I/O virtualization is a hypervisor technique based on the I/O device emulation. By utilizing the functionality of CPU (e.g., eventfd), VM invokes the I/O exception to hypervisor and the hypervisor progresses the proper I/O emulation. Through this process, hypervisor converts the commands requested by the VM into binary commands that the host machine can understand and delivers to the corresponding device driver. After the host device completes the task, the hypervisor transmits the result in the opposite direction. There are two major issues with software-based I/O virtualization: CPU overhead and latency. The exception emulation causes excessive CPU overhead in I/O request notification process, incurred by frequent CPU context switching and complicated system call stacks. Furthermore, since the VM and the hypervisor run in the user space, the work priority can be set to low, and request processing can be delayed. In order to improve the performance, I/O virtualization scheme virtio adopts *para-virtualization* technology, where the VM-side kernel notices that the system is running on virtual machine and provides a software interface instead of total emulation of hardware. Fig. 2 shows the difference of virtio comparing with exception-based system. If a user enables virtio, hypervisor creates virtio device instead of emulated device and attaches to the virtio-pci bus to invoke the virtio driver initiation of VM. In this article, we follow the traditional nomenclature of calling the hypervisor's virtio device as the *backend* and the VM's virtio driver as the *frontend*. The frontend driver generates a shared buffer named *virtqueue* that used to fill out the data buffer address when requests arrive. On each request from VM user, virtio driver allocates a fixed-size buffer from the shared memory space, writes the information of the request (buffer address, size, and so on) to the virtqueue, and sends notification signal to the host. The backend device processes the device operation by reading the buffer, fills out the virtqueue buffer with operation results, and sends notification signal as the same procedure of the above. Introduction of KVM can accelerate the notification process, and the shared memory access between the VM and the hypervisor can highly reduce the overhead. We adopted virtio protocol to secure the basis of high performance while operating device virtualization.

C. Ambient Network

Ambient network refers to a research field that proposes an integrated network framework for connection between user devices and from peripheral devices to user devices. The proliferation of IoT concepts resulted in the creation of numerous standards and protocols against resource limitations such as network bandwidth, power, and computing, which led users to experience increasing structural overhead. Ambient network technology aims to provide a unified environment to end users and application layer developers by presenting an integrated platform for the standards of differentiated communication and network technologies. Akasiadis et al. [6] suggest an IoT platform that supports five widely used application layer protocols, and Harman et al. [7] suggest a cloud-based networking platform with IoT devices for mobile actors such as robots. In addition, He et al. [8] present a blockstream-based cloud service platform for ambient computing, and Liang et al. [9] present a solution to the heterogeneous multitask assignment problem that can occur in

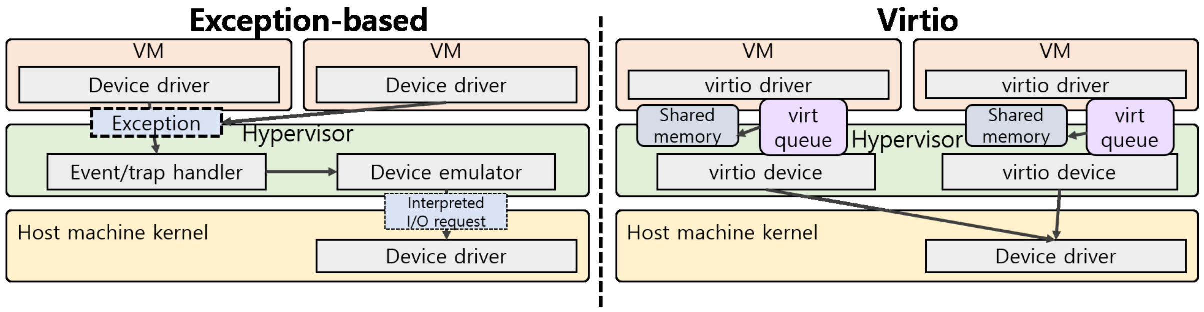


Fig. 2. Structural difference between exception-based device access and `virtio`.

large-scale crowd sensing. As seen in previous studies, ambient computing technology has mainly assumed IoT situations and presented solutions for sensitive data traffic, but essentially it has approached device access and networking separately, leading to various types of platform.

In this article, we present a system that can relax the boundaries between local and remote for device access and provide an ambient device access to VM users by extending the `virtio` standard technology utilized by most VMs. We describe the details of our design in the following section.

III. System Design

The core objective of the proposed system is to provide a universal, consistent, and simple tool for ambient device access. CPU and memory virtualization techniques such as VT-x [10], AVIC [11], and paging [12] have been widely provided with type-1 hypervisors such as KVM and Xen [13]. I/O virtualization schemes such as `virtio` and `vhost` propose performance enhancement strategies, but the recent trends focus on the performance of the devices and barely cover the diversity of device access procedures. For instance, data plane development kit [14] and storage performance development kit [15] accelerate the performance by userspace driver operation. However, in order to support various types of devices, consequently, a new type of device should be declared every time. Furthermore, since `vhost` family aims to share the resources of the host with VMs, the resulting implementations depend on the portability of the host environment. In the case of `vhost-blk`, we considered to utilize this architecture at the design level but determined not to use due to the structural overhead and limitation of usage while connecting to the remote block device. To raise the sustainability and flexibility of the architecture, we devised a way to provide an abstracted interface of device for VM following `virtio`.

The motivation of our design starts from the virtual file system (VFS) of Linux. VFS provides an abstract layer for various types of file systems, including the device driver. For each device connection, VFS creates a device file that a user can communicate with the device driver and perform desired operation. Utilizing VFS, we attempted to design a backend device to let the VM show up an ambient device as a form of the device file and guest users open this file without being aware of the difference in connection details. For doing so, we analyzed how the device files of VM show up during operation and found

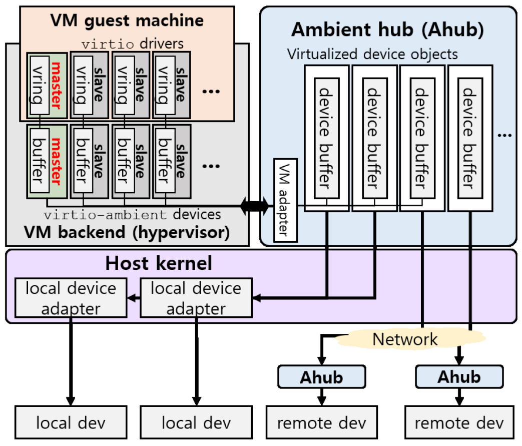


Fig. 3. Architecture of the proposed system.

that *the hypervisor invokes the device file generation of VM*. To guarantee the performance and portability, we chose `virtio` module and QEMU for our system design and implementation. We comprehensively investigated the `virtio` implementation in QEMU and designed a new type of `virtio` device `virtio-ambient` as an addition to the entire virtualization process. We built `virtio-ambient` by renovating `virtio-blk`, which defines block data transmission in duplex. The following subsections address each part of the system.

A. Architecture

Fig. 3 shows the architecture of our proposed system. A group of Ahubs plays a role in communicating with the actual devices and sustaining the list of virtualized devices by sharing the device information. Ahub does not explicitly call the `virtio` device realization but generates an ambient device object to perform actual operation on the request. Rather, VM hypervisor realizes a representative device called *master ambient device* which establishes the connection between Ahubs and stores the ambient devices information. By exchanging a certain sequence of messages between the master ambient device and Ahub, master ambient device creates or removes backend `virtio` devices, tracks the changes in the status and data, and attaches or detaches them to the VM. During the device realization process, we devised a simple trick that defines a new type of `virtio`

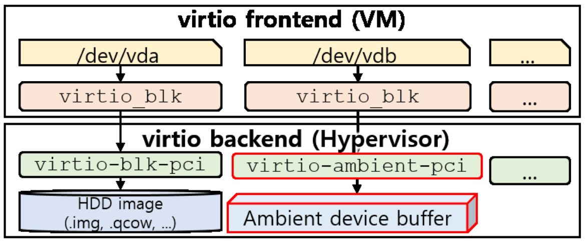


Fig. 4. virtio-ambient-pci attachment with virtio-blk front-end driver.

backend device named virtio-ambient-pci while attaching the device to the type of virtio-blk. Fig. 4 describes this process. If a VM user configures to use an hard disk drive (HDD) with virtio interface, VM backend creates virtio-blk-pci device and VM frontend operates virtio-blk driver. If (s)he configures to use an ambient device provided by Ahubs, VM backend creates virtio-ambient-pci device and VM frontend operates virtio-blk driver.[1] Thus, the format of device file name is the same as /dev/vda, /dev/vdb, and so on.[2] If a VM user enables the use of virtio-ambient in VM initial configuration, usually by using libvirt or command line, the VM process creates a master virtio-ambient-pci device in backend. At the backend, this master ambient device opens a server socket and listens for the Ahub connection. Ahub always works on the host or other machines. Ahub uses a device-specific interface to create virtual devices with read and write operations implemented. Meanwhile, Ahubs establish a network and share the information of their own virtual devices. In the background, Ahub periodically collects the data from its local device and saves it to the device buffer. Ahub attempts to connect to a VM and sends the device information or the data stored in device buffer. Backend of VM accepts the Ahub connection and receives the message through socket, made in the master ambient device. If Ahub grants device registration message to the VM, the master ambient device operates the virtio-ambient-pci device generation and attachment procedure following virtio protocol. The master ambient device then stores the meta information of the newly added device in the buffer that the device owns and saves the buffer information (e.g., logical address) of the new device in its own database for future data transfer. If Ahub sends data transmission message to the VM, the master ambient device finds a buffer to store the received data. VM guest can obtain the available lists of ambient devices by reading the buffer of the master ambient device. Then, the VM guest can open and read/write the ambient devices with Linux system calls. VM-side virtio-blk driver allocates a shared memory space typed by iobuf, pushes the address of buffer to the vring buffer, and sends notify signal to the backend device. In backend-side, our virtio-ambient process handles the read or write operation by copying its buffer to the shared memory or by *sending data transmission message to the Ahub*, respectively. Ahub receives the message from VM and write to its own local device or forward to the other Ahub to let it handle this write operation. Some application-level architectures such as HTTP REST can be of consideration, but we determined not to specify the APIs for the flexibility of system design. We defined the following routines of virtio-ambient devices and Ahubs to be applied to various forms of network, from wired wide area network to wireless personal area network. In our implementation, we used transmission control protocol (TCP)/internet protocol (IP) for networking.

[1]Following design convention of QEMU, virtio-ambient-pci wraps the abstract module virtio-ambient connected to the virtio-pci bus. We implemented the core functions such as initialization, read, and write operation at virtio-ambient.

[2]The reason of device file name /dev/vdX is that the kernel generates a virtual block device via virtio-blk module. The format of device file names can vary according to the type of virtio submodule the system adopted.

B. Ambient Virtio Routines

This section describes how to enable ambient device operations in terms of message format and process flow. Note that the protocol specifications, such as field sizes in bytes, are not fixed in the current state and may change in public releases.

1) Ahub Message Format: We declared a simple 32-b message header format for Ahub network. All network participants including virtio-ambient exchanges data within the regulation of this header format. Fig. 5 shows the layout of the format, and the detail explanation is in the following.

a) *Message type* indicates what type of the information this message has. For instance, in the case of CODE_REG, which refers ambient device registration, the fields in the header message contains the information to generate the ambient device object.

b) *Device ID* is the unique 4-b number of the ambient device. During the ambient device creation phase, Ahub checks the list obtained from the Ahub network and assigns a unique number to the device. If a duplicate ID is issued to the network as CODE_REG, the recipient of the message will discard this message to avoid conflicts.

c) *Size info (buffer/data size)* is the size of buffer that the ambient device should have or the size of data being sent following this header. By setting this field to 1, ambient device can act as a character device. If CODE_DATA message has the larger value of data size than the Buffer size of this device, the message receiver abandons this message and the following data to avoid overflow.

d) *Fields 1, 2, ..., 4* are modifiable spaces for the requirements of each message type. CODE_REG fills out the basic information of new device such as device name and access configuration (Section III-D). Then, the rest of total 8-b fields will be used to cache the local information of VM or Ahub; in the case of VM, the master ambient device stores the name of device file (/dev/vda,...).

We describe the sequences of the processes in the next section.

2) Device Registration Process: Algorithm 1 shows the flow of ambient device registration and plug process. Basically, physical devices establish connection to a physical machine by wire or wireless. Ahub runs at this physical machine, initiates device operation interface, and generates a specification of the device. Then it broadcasts CODE_REG messages to the peer Ahubs and the connected VM backend. When the Ahub establishes a new

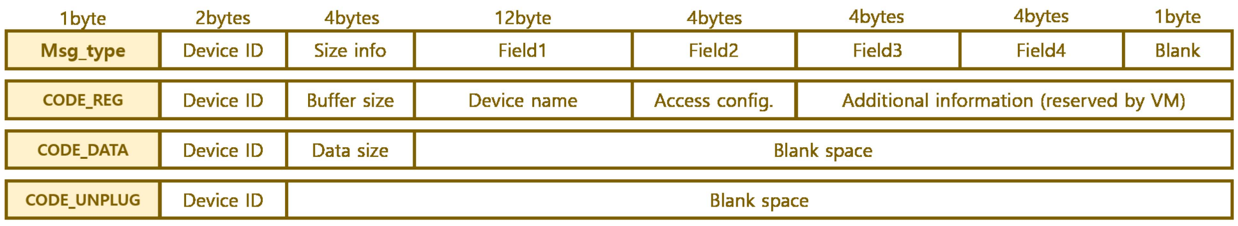


Fig. 5. 32-b message header format of Ahub.

Algorithm 1: Device registration process.

```
 1:  /* Ahub */
 2:  function Create_Ambient_DevdevInfo
 3:    $fd \leftarrow$ Get_File_Descriptor(devInfo)
 4:    $did \leftarrow$ Get_New_DeviceId()
 5:    $bsize \leftarrow$ Set_Buffer_Size()
 6:    $Adev \leftarrow$ New_Adev($fd$, $did$, $bsize$)
 7:    $msg \leftarrow$ Make_Header(CODE_REG, $Adev$)
 8:    Send $msg$ to nodes
 9:    Start_Thread(Dev_Read, $Adev$)
       ▷Algorithm 2
10:  end Function
12:  /* VM backend */
13:  function Handle_Message$msg$
14:    if $msg.type$ == CODE_REG then
15:      $buf \leftarrow$ Allocate_Buffer($msg.sizeInfo$)
16:
17:      /* QEMU Implementation */
18:      $opts \leftarrow$ qemu_opt_create()
19:      $dev \leftarrow$ qdev_device_add($opts$)
20:      drain_call_rcu()
21:
22:      Register_Buffer($dev, buf$)
23:      $adev \leftarrow$ Adev($msg, dev$)
24:      Copy $msg.field1$ to $adev.name$
25:      Copy $msg$ to $master.buf$
26:    end if
27:  end Function
```

Algorithm 2: Device read process.

```
 1:  /* Ahub */
 2:  function Dev_Read$adev$
 3:    while !$adev.closed$ do
 4:      $data \leftarrow adev.fd$.read($adev.bsize$)
 5:      if !$data$ then
 6:        continue
 7:      end if
 8:      $len \leftarrow$ Get_ByteLength($data$)
 9:      $msg \leftarrow$ Make_Header(CODE_DATA, $adev$,
         $len$)
10:      Send $msg$ to nodes
11:      Send $data$ to nodes
12:    end while
13:    Unplug_Adev($adev$)        ▷Algorithm 4
14:  end Function
16:  /* VM backend */
17:  function Handle_Message$msg$
18:    if $msg.type$ ==CODE_DATA then
19:      $adev \leftarrow$ Find_Adev_From_Did($msg.did$)
20:      if !$adev$ then
21:        return
22:      end if
23:      $len \leftarrow msg.sizeInfo$
24:      if $len > dev.bsize$ then
25:        return
26:      end if
27:      $data \leftarrow$ Get_Data($msg.socket, len$)
28:      Copy $msg$ and $data$ to $adev.dev.buf$
29:    end if
30:  end Function
```

connection to a VM, it sends several `CODE_REG` messages to the VM backend for follow-up. VM backend stores this message to its master ambient device buffer, and the VM guest can retrieve the information of ambient devices by reading the master device buffer. After attaching the virtio device to the VM, frontend loads the virtio driver, and new device file `/dev/vdX` emerges for the device. Note that the guest VM does not need to know how to retrieve the data from the physical remote device, what information is needed for connection, and where it is attached. This is because Ahub logic faithfully follows the concept of ambient access, but `CODE_REG` sometimes has to contain more information of the devices for better utility such as data bandwidth, status of device, location of physical device, and so on. We leave the informative design of message format to future work.

3) Device Read Process: Algorithm 2 shows the flow of device read operation process. Ahub periodically updates the device data and saves to its own buffer. The content of data could be various such as camera image, printer status, and so on. Since the connection between the VM and the Ahub is in the master ambient device, the master finds a proper buffer to store the data of this device and copies the data with header information. Read operations from VM guests follow the same procedure as master ambient device. Frontend driver `virtio-blk` performs the same process of existing `virtio` protocol, which includes shared memory allocation, and vring control, and notification signal transmission. `virtio-ambient-pci` catches the signal and copies the readable data that the master

Algorithm 3: Device write process.

```
 1: /* Ahub */
 2: function Handle_Message$msg$
 3:   if $msg.type$ ==CODE_DATA then
 4:     $adev \leftarrow$ Find_Adev_From_Did($msg.did$)
 5:     if !$adev$ then
 6:       return
 7:     end if
 8:     $len \leftarrow msg.sizeInfo$
 9:     if $len > dev.bsize$ then
10:       return
11:     end if
12:     $data \leftarrow$ Get_Data($msg.socket, len$)
13:     $adev.fd$.write($data$)
14:   end if
15: end Function
17: /* VM backend */
18: function Virtio_Ambient_Write$buf$
19:   $adev \leftarrow$ Find_Adev_From_Buf($buf$)
20:   $data \leftarrow$ Copy_Data($buf$)
21:   $len \leftarrow$ Get_ByteLength($data$)
22:   $msg \leftarrow$ Make_Header(CODE_DATA, $adev$, $len$)
23:   Send $msg$ to connected Ahub
24: end Function
```

Algorithm 4: Device unplug process.

```
 1: /* Ahub */
 2: function Unplug_Adev$adev$
 3:   $adev.fd$. close()
 4:   $msg \leftarrow$ Make_Header(CODE_UNPLUG, $adev$)
 5:   Send $msg$ to the nodes
 6:   Remove $adev$
 7: end Function
 9: /* VM backend */
10: function Handle_Message$msg$
11:   if $msg.type$ ==CODE_UNPLUG then
12:     $adev \leftarrow$ Find_Adev_From_Did($msg.did$)
13:     /* QEMU implementation */
14:     qdev_unplug($adev.dev$)
15:
16:     Remove $adev$
17:   end if
18: end Function
```

ambient device secured before, and sends notification signal to the guest. Note that the processes between physical device and Ahub, Ahub and VM backend, and VM backend and VM frontend are asynchronous. Depending on the characteristics of the device, the cycle of Ahub to VM message transaction could vary. For camera devices, certain frames per second (FPS) can be important for quality of service. In our implementation, Ahub determines the read cycle and data transfer cycle, respectively, while generating the ambient device object.

4) Device Write Process: In the case of write operation described in Algorithm 3, VM-side sends `CODE_DATA` message to the Ahub. VM guest first notifies the backend device with address of `iovec`-type buffer. Then, the driver reads the buffer and sends `CODE_DATA` message with the data to its connected Ahub. The Ahub checks if the operation is for the device that the Ahub locally accesses or not. If Ahub can directly access the device, it performs write operation with the device interface it has. Otherwise, the Ahub forwards the message to the Ahub that can physically access the targeting device.

5) Device Unplug Process: Algorithm 4 describes the device unplug process. If the device is inaccessible due to physical abnormality or request of device owner, the Ahub directly attached to the device deletes the ambient device object and reports `CODE_UNPLUG` to the other Ahubs and attached VMs. The other Ahubs deletes the entry of the device from its device list and alerts to their connected VMs. Each VM that receives `CODE_UNPLUG` message performs an unplug process following standard `virtio` implementation of backend and frontend unplug processes, defined at each side: standard version of QEMU and Linux kernel. The unplug processes are also invoked on the disconnection between the VM and Ahub, for avoiding the wrong process of VM applications. Reminding the collection of message formats presented in Fig. 5, we targeted any shapes of network desiring to provide the ambient experience to VM user and application. Multiple Ahubs can share the information of devices and each can provide the device data to connected VMs. Onto this N-to-N architecture, one can adopt various methodologies to efficiently utilize the network resource. In Section IV, we present multiple empirical scenarios with single or multiple VMs and hosts to show the flexibility of the network structure. Including the demonstrated cases, Ahub and `virtio-ambient` can hold more complex and scalable network by adjusting the `Device ID` field of the message format.

We analyzed the computational overhead of data read and write processes in terms of time complexity. As shown in Algorithms 2 and 3, Ahub and `virtio-ambient` directly copy the device data into the memory without nested loop. Thus, the time complexity of each process can be normally presented as $O(MN)$, where M and N refer to the data size and the number of readable/writable machines, respectively. To the best of our knowledge, even though there is no publications addressing the time complexity of our benchmark system robot operating system (ROS), we argue that the complexity is equal or similar to the proposed system due to the simplicity of the purpose. However, the structural overhead of ROS architecture results in the difference of the performance and the resource cost, as shown in Section IV.

In addition, the above routines of ambient device operations indicate the high flexibility of the shape of ambient devices. With `read/write` implementation, Ahub can generate a virtual device and publish to the network. So, in the same context, *VM* can create a virtual device that performs `read/write` operations with a certain action. We address the concept of VM-side device creation in the following section.

Algorithm 5: Metadevice generation process.

```
/* VM backend */
function Virtio_Ambient_Write$buf$
  $adev \leftarrow$ Find_Adev_From_Buf($buf$)
  if $adev == master\_adev$ then
    $devname \leftarrow$ Copy_Data($buf$)
    $bsize \leftarrow$ Set_Buffer_Size()

    /* QEMU implementation */
    $opts \leftarrow$ qemu_opt_create()
    $dev \leftarrow$ qdev_device_add($opts$)
    drain_call_rcu()

    $adev \leftarrow$ New_Adev ($devname$, $did$, $bsize$, $dev$)
    $msg \leftarrow$ Make_Header(CODE_REG, $adev$)
    Send $msg$ to connected Ahub
  else
    Do Algorithm 3
  end if
end Function
/* Ahub */
function Handle_Messages$msg$
  if $msg.type$ ==CODE_REG then
    $did \leftarrow$ Get_New_DeviceId()
    $adev \leftarrow$ New_Adev($msg$, $did$)
    $adev.meta \leftarrow true$
    Forward $msg$ to neighbor nodes
  else if $msg.type$ ==CODE_DATA then
    $adev \leftarrow$ Find_Adev_From_Did($msg.did$)
    if $adev.meta$ then
      $adev.buf \leftarrow$ Get_Data($msg.socket$, $len$)
    else
      Do Algorithm 3
    end if
  end if
end Function
```

C. Metadevice

`virtio-ambient` provides the ability of virtual device creation not only to Ahub but also to VM guests. Virtually defined device called *metadevice* is a novel concept in our proposed system. The Ahub wraps the functionalities of devices into `read`/`write` operations. Thus, in the context of our system, metadevice can be any form of software or cyber–physical system such as drone(s), simulator, or neural network. Furthermore, a metadevice can read or write other metadevices or ambient devices to generate higher level information.

Algorithm 5 shows the message flow of metadevice generation, starting from a VM guest. Similar to attaching a device, the guest writes the new device name of the metadevice to the master ambient device, and the VM backend creates the backend ambient device. Opposite to the registration process, VM backend sends `CODE_REG` message to the Ahub, and the Ahub propagates this message to the other Ahubs and connected VMs. The other nodes (Ahubs and VMs) notice the creation of new metadevice from this message and update the device list of themselves. One may argue that the concept of metadevice has no difference with the previous networking tools, such as topic pub-sub mechanism of ROS. However, there are clear novelties of our `virtio`-based system as the following:

1) *Lightweight VM:* For ROS, designing a multinode system requires ROS installation of all VMs. Furthermore, each VM should initialize the ROS node, define subscribers and publishers, and run event loop for catching the data reception event and operating callback routine. However, `virtio-ambient` requires VM guests to do a set of simple system calls, device open, read, and write. This difference not only simplifies the VM-side application design but also reduces the overhead which occurs in VM process.
2) *Secured VM:* VM guests do not have networking information of ambient devices, such as their IP addresses. This indicates that the VM backend has already buffered data; so guests *do not need any networking*. From this feature, the user can generate highly secured environment in VM where the host machine vouches the network security through its resource.
3) *Software reusability:*The main advantage of the proposed system is that the VM guest can access the ambient device by the same way of local device. Thus, the VM user can utilize the software products implemented for local devices while using ambient devices.

Note that Ahub can freely generate and issue the metadevices since the VMs and other Ahubs do not care about how the devices are actually operated. However, for metadevices a VM created, the creator can do `write` operations. The creator sends `CODE_DATA` to the Ahub by `write` operation, and the connected Ahub stores the data into the device buffer. Other VMs can `read` the metadevice data through `CODE_DATA` message sent from the Ahub. Broadly speaking, Ahub should have criteria to determine whether or not to perform the ambient device operation for each request. We address how to grant the accessibility of ambient device in the next section.

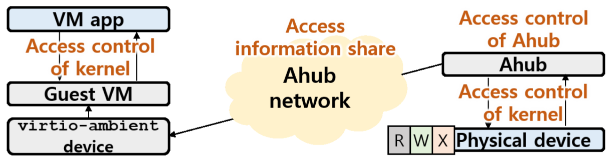


Fig. 6. Access control stages of ambient devices.

D. Access Control

Including metadevice, all ambient devices contain the information of control right. If a group of Ahub operates as a platform, the Ahubs should manage the availability to read or write the device data for security and extensibility. Fig. 6 shows the access control stages of the ambient devices. Linux and other operating systems have their own access control schemes; so the proposed system should consider working with them. In the case of Linux kernel, opening block device file basically requires the user to

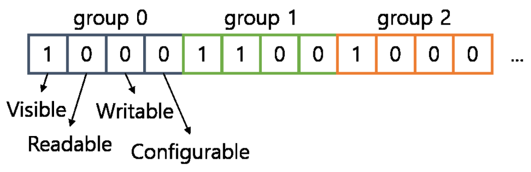


Fig. 7. Access configuration field layout.

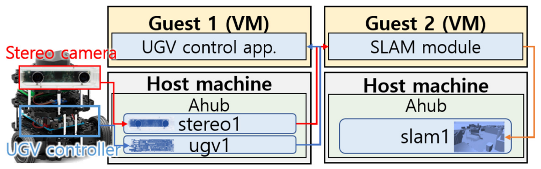


Fig. 8. `virtio-ambient` performance evaluation experiment.

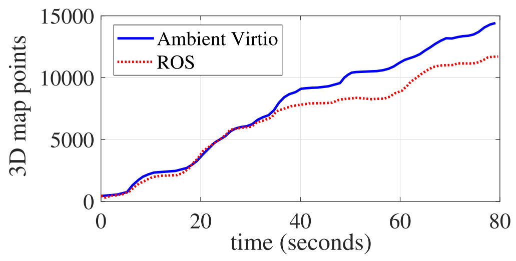


Fig. 9. Number of 3-D features in `virtio-ambient` performance evaluation experiment, compared with ROS.

be at the `block` user group or super user authority. If the guest user has permission to read and write all block device files, all ambient devices can be fully exposed to the user. Thus, the Ahub should intervene the accessibility of ambient devices. Utilizing the Ahub routines and mechanism, Ahub mainly controls the device data exchange by grouping VMs. We use Field 2 of `CODE_REG` header, as shown in Fig. 7. Each group of VMs indicates the level of the device accessibility, presented by 4 digits. Each digit has the following meaning in order.

1) The **Visible** digit indicates whether the device is present or not to a group of VMs. In device registration process, `CODE_REG` message is only sent to the group of VMs whose Visible digit is set to 1.
2) The **Readable** digit indicates whether the VM group can read the device data. Ahub sends `CODE_DATA` message only to the group of VMs whose Readable digit is set to 1.
3) The **Writable** digit indicates whether the group of VMs can perform the write operation. Ahub processes the write operation of the physical device only when the operation request is sent by the group of VMs whose Writable digit is set to 1. In the metadevice case, Ahub updates the device buffer if the device creator sent the write operation.
4) The **Configurable** digit indicates whether the VM group can operate the specific configuration command through the write operation (e.g., changing camera resolution). Ahub handles the configuration operation sent from the group of VMs whose Configurable digit is set to 1.

By using 4-b field, an ambient device can set the access right of eight groups. The number of groups can vary by changing the access configuration field. The hypervisor set the group number of a VM at startup. Through this access control, Ahubs can operate a large-scale device platform for ambient computing service.

IV. Performance Evaluation

We implemented the proposed system to validate and evaluate the system. As mentioned earlier, we modified the latest version of QEMU written in C for declaring `virtio-ambient` module and `virtio-ambient-pci` device. Additionally, we implemented Ahub and a simple guest-side wrapping functions called `alib` by Python. We enabled `kvm` as a host-kernel hypervisor and used Ubuntu 20.04 LTS image for guest VM which runs Linux kernel 5.4. We run the hosts and the VMs at a desktop equipped with Intel Core i5-6600 3.30 GHz and 16 GB RAM, and a laptop equipped with Intel Core i7-4500 U 1.80 GHz and 8 GB RAM. Each machine accessed to the Wi-Fi network by external network interface card named TL-WN722 N made by TPLink, and a device named KCX-017 is used for power measurement. For image acquisition, we used USB web camera devices named S604HD. Finally, we adopted Robotis Turtlebot3 Burger [16] for mobility. To prove the efficiency of the system, we compare the analytic figures of the demonstration scenario with the ROS. The reason for adopting ROS for the comparison study is the similarity of the role and the motivation with the proposed system. Both `virtio-ambient` and ROS provide a unified framework to the applications that require the access to the local or remote device. To let a virtual machine access the remote camera and obtain the stream of image, a developer should determine how to implement the application: using ROS, `virtio-ambient`, or using another platform made by others or himself. We determined to show the performance of the proposed system with ROS implementation since ROS is open-source and widely used for data access and sharing.

A. *Comparison With ROS*

As shown in Fig. 8, we designed a test bed. A test bed was designed using an unmanned ground vehicle (UGV) and two VMs. We equipped a stereo camera on the UGV. Then, we virtualized the camera and the UGV controller and plugged to VMs with the name of `stereo` and `ugv1` so that a VM controls the UGV and the other VM performs simultaneous localization and mapping (SLAM) operation. `stereo` periodically reads the stereo camera and transmits a concatenated stereo image in 60 frames per second. Meanwhile, a VM running SLAM generated a metadevice `slam1` to write the resulting 3-D map to the Ahub. Furthermore, we implemented the same test bed while using the ROS interface for comparison with the existing system. We launched `roscore`, `turtlebot_bringup`, and `uvc_camera` nodes at UGV 1. UGV 2 subscribes the stereo image published by `uvc_camera` node and performs SLAM while publishing the control commands to the `turtlebot_bringup`. In both test beds, we launched `rtabmap` [17] for SLAM operation. We moved the UGVs for 80 s in circular trajectory and collected the number of the 3-D map points in each case. Fig. 9 shows the results of two experiments. As shown in the figure, the case of `virtio-ambient` resulted about 3000

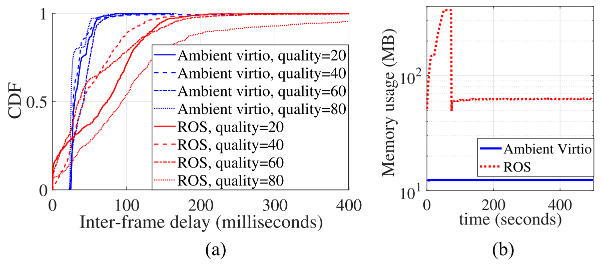


Fig. 10. Performance comparison of stereo image transport with respect to interframe delay (left) and memory usage (right). (a) Interframe delay. (b) Memory usage.

more 3-D points than the case of ROS. The main reason of this result is the extra overhead of the ROS module running in the guest VM of UGV 2. ROS-based system processes image data at the application layer of guest VMs, which may incur poor performance despite the assistance of `KVM`. Furthermore, ROS consequently uses `virtio-net` for image transport; so the performance difference could be larger if another heavy traffic is located. In contrast, `virtio-ambient` uses separated device buffer for each device; so the bottleneck of data exchange could be less than the case of ROS. This experiment showed that the proposed system works as intended with less overhead in VM. Since the same SLAM operation ran in both test beds, we intensively measured the performance of image transport part of the scenario. Fig. 10 shows the interframe delay and memory usage of image reception and publish (for `rtabmap`) process in each case. We varied the quality of compressed image from 20 to 80 and collected the interval of the arrivals of the images. Since ROS transmits the left and the right images separately, we measured the delay at every left image reception. As shown in Fig. 10(a), the delays of our system are less than 100 ms in 99% of the cases, while the delays of ROS are spread at the range of 0–120 ms in most cases. The average delays of Ambient Virtio and ROS are 35.58 and 79.94 ms, respectively, where our system performs 2.24× faster image transfer. In addition, we observed the collective image transmission of ROS case due to the bottleneck in the `virtio-net`, which decreases the quality of experience (QoE) of the system. In the case of quality= 80, long tail of interframe delay emerged because of the large-sized image data. The difference of performance comes from the difference of the structural overhead of the systems. ROS reactively catches the image subscription event and unpacks the data while operating ROS interface, but `virtio-ambient` copies the data from the device buffer where the backend process proactively receives and stores the image data. To clearly show the advance in the viewpoint of memory, we plotted the memory usage of the *raw* image reception process of each case, as shown in Fig. 10(b). For compressed image transfer, regardless of quality, our proposed system shows less memory usage than ROS by 29.0 MB. However, at the raw image transfer of ROS, the process instantly raises the memory usage to over 120 MB at the interval of 14–75. Whereas, `virtio-ambient` shows consistent usage of memory since the overhead opening the device file is unnoticeable. In the steady state, Ambient Virtio and ROS use 12.38 and 62.68 MB of memory, respectively, where our system achieves 80.25% memory savings. Finally, we measured

TABLE I
AVERAGE POWER CONSUMPTION COMPARISON OF STEREO IMAGE

	Average power consumption
`virtio-ambient`	16.4 mAh
ROS	18.2 mAh

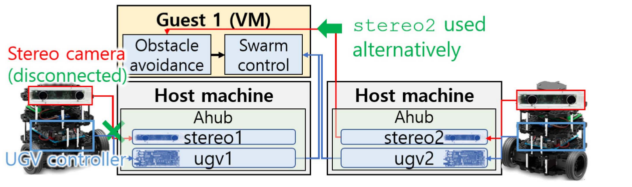


Fig. 11. Device switching experiment scenario.

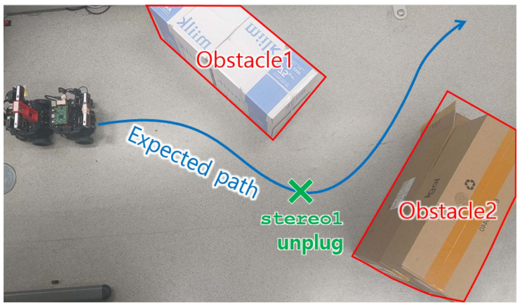


Fig. 12. Device switching experiment setup.

the power consumption of network interface card of each case, while sustaining the image transmission for 10 min. The result is shown in Table I. The case of `virtio-ambient` consumes about 16.4 mAh of power, while ROS consumes about 18.2 mAh. The reduced power consumption of `virtio-ambient` comes from the less structural overhead of Ahub message routines. Through the evaluation of image transmission, we verified that `virtio-ambient` makes the VM access remote device data with much less overhead.

B. *Device Switching Experiment*

Utilizing `virtio-ambient` and Ahub functionalities, we designed a device redundancy scenario and verified the effectiveness of the system. Since the `virtio-ambient` based applications access remote devices the same as local ones, the VM can simply switch which device to use by changing the device while reusing the other parts of the software. Fig. 11 describes the device switching experiment scenario with two UGVs and one VM. Each Ahub running in the companion board of each UGV virtualized the stereo camera and the controller. We plugged all four ambient devices to VM. VM generates disparity image from one of the ambient stereo cameras and determines the directions of the UGVs to avoid the obstacle collision. We previously input the relative position between UGVs for swarm control. Fig. 12 shows the experiment environment setup. Two obstacles located at the UGVs path, and blue solid line in the figure indicate the expected path of UGVs. After detouring Obstacle 1, we disconnected a stereo camera equipped in UGV 1. Then, Ahubs removed `stereo1`, and the VM replaced the image source to

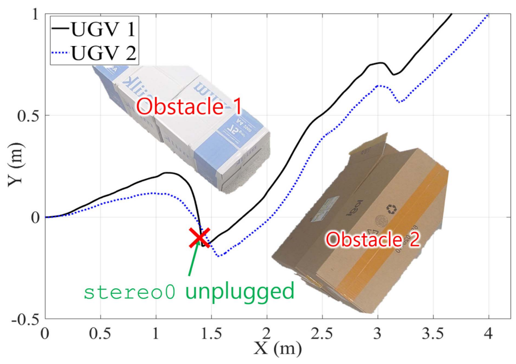


Fig. 13. UGVs trajectories of device switching experiment.

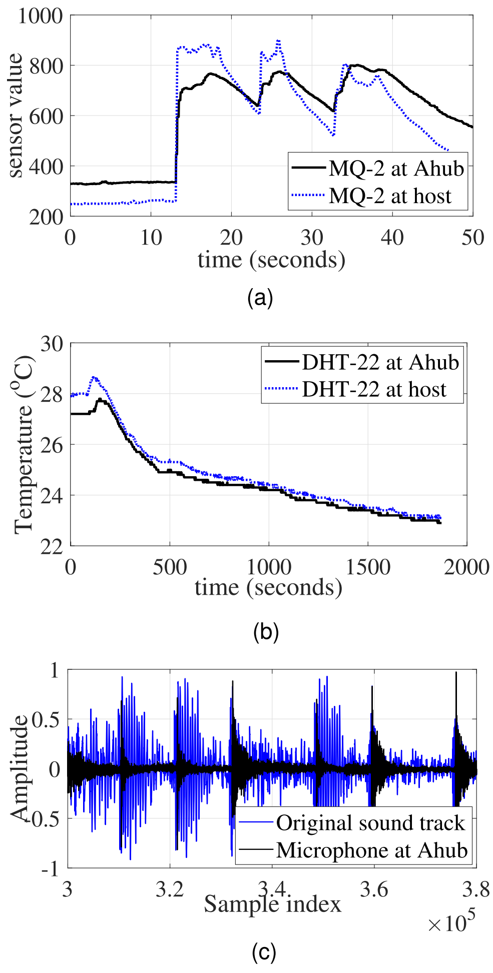


Fig. 14. Miscellaneous device demonstration. (a) Gas sensor experiment. (b) Temperature sensor experiment. (c) Audio input experiment.

stereo2. The VM can proceed the obstacle avoidance control because we gave the relative position of UGVs. Without our system, the VM would have to either reconnect to the other stereo of UGV 2 via an appropriate network stack or stop running to install a new camera on UGV 1. However, since we spread virtio-ambient camera through Ahubs before, the switching instantly occurs and VM can process the UGV control.

Fig. 13 shows the experiment results with experiment setup. The video of the demonstration can be found at [18].[3] As shown in the figure, the UGVs detoured the obstacles from the distance information obtained by stereo1. Then, when UGV 1 reached near the position of (0.7 m, 1.0 m), Ahub triggers the unplugging operation of stereo1. The VM immediately detected the removal of stereo1, searched for a replacement by device name, and switched the source of disparity image to stereo2 whose physical device was at UGV 2. After a few staggering, UGVs found the best direction and directly moved forward. From this demonstration, we verified that our proposed system is available in the circumstances where VM requires the redundancy of data sources.

C. Miscellaneous Devices

We conducted various demonstrations with various devices to confirm the extensibility of Ambient Virtio system. We investigated multiple types of peripheral devices and implemented as ambient devices while configuring from the hardware connections to the proper software interface of each. Considering the page limit of the article, we chose the devices that effectively show the usability of our system.

1) *Atmospheric sensors:* We defined virtual gas sensors with a gas sensor MQ2 and a thermometer DHT-22, respectively. Ahub created ambient device objects called mq0 and dht0 and transmitted the sensor data to VM. To validate the system, we connected another set of sensors at host machine and concurrently collected the data while emitting butane gas for three times and lowering the air temperature. Note that we synced the clock with -rtc clock=host option of QEMU and used the same code at the host and the VM with only difference being the device file name (/dev/ttyACM0 and /dev/vdX). Fig. 14(a) and (b) shows the sensor data with respect to the time. In the figures, there are small differences in the sensor values due to the difference of sensor position and hardware calibration. However, both results show the clear reaction of the gas emission and temperature changes. The results imply that the VM can ambiently access the virtualized sensors just as a physical machine accesses physical sensors.
2) *Acoustic device:* We defined an audio input device as a virtual device. With the same process of sensors and cameras, we connected a microphone at Ahub and created mic0 ambient device. Ahub stored the audio stream data into fixed size buffer and transferred to VM with global sample index. We played a simple sound track nearby the device, and VM reads the sound data from the ambient device. Fig. 14(c) compares a subset of the samples collected at the VM to the original sound file. The difference of waveforms comes from the performance of the audio input device, since we verified that there is no loss from the Ahub to VM transmission. From the experiment, we verified that Ahub can virtualize media device and serve the VM to access in ambient environment.

[3]Note that we followed double-blind policy while uploading the video. We removed any information that leads readers to infer the authors' information.

V. Conclusion

In this article, we proposed a novel device virtualization scheme called *Ambient Virtio* for ambient computing environment. Existing virtualization enables ambient access to the core resource of computing platform rather than peripheral devices. By augmenting the commonly used virtualization technique, we enabled for VMs to interact with the ambient devices, as the same way of local devices. We summarized the contents of this article as follows:

1) We implemented `virtio-ambient` device following `virtio` protocol and *Ahub* that virtualizes the physical device and enables ambient access of VMs. We specified the routines for the system that cover the lifespan of ambient device with sample message format.
2) We devised a novel concept of virtually defined device named *metadevice*. Any components including VMs and Ahubs can define a metadevice and publish to the ambient use. We indicated the possibility of the evolution of device system through metadevice design.
3) We constructed an access control scheme for ambient device platform. We designed four-staged control rights reflecting the features of the ambient device system.
4) We demonstrated our proposed system with empirical devices. From demonstration, we verified the performance improvements in VM-side process and the broad availability of the system.

To improve the proposed system, we set the following future research plans and continue our in-depth research.

a) *Data traffic optimization:* Due to the diversity of the role of device, the network traffic in terms of bandwidth and the required round trip time (RTT) also varies. We can further deepen the ambient device management process to elevate the quality of ambient computing platform.
b) *Ambient device data encryption:* In addition to the access control, Ahub or VM can encrypt the device data for confidentiality. Then, the access control logic will include the key generation and distribution process.
c) *Portability:* The current design requires `virtio` module to VM. By investigating the other structures used in popular, we plan to support as many VMs as possible.

The ultimate goal of our research is to advance the ambient computing concepts in which users can access and utilize all the capabilities of ambient devices while securing minimal equipment such as communication modules and low-cost computing modules. We believe that our proposed system is a feasible solution for engaging smart things in the ambient computing ecosystem.

References

[1] S. K. Lee, M. Bae, and H. Kim, “Future of iot networks: A survey,” *Appl. Sci.*, vol. 7, no. 10, 2017, Art. no. 1072.
[2] M. Aghaei, F. Grimaccia, C. A. Gonano, and S. Leva, “Innovative automated control system for PV fields inspection and remote control,” *IEEE Trans. Ind. Electron.*, vol. 62, no. 11, pp. 7287–7296, Nov. 2015.
[3] A. Rostami, “Introduction of team viewer software,” *Interdiscipl. J. Virtual Learn. Med. Sci.*, vol. 3, no. 1, pp. 70–70, 2020.
[4] D. Kandris, C. Nakas, D. Vomvas, and G. Koulouras, “Applications of wireless sensor networks: An up-to-date survey,” *Appl. Syst. Innov.*, vol. 3, no. 1, pp. 14–14, 2020.
[5] M. Chae, H. Lee, and K. Lee, “A performance comparison of Linux containers and virtual machines using docker and KVM,” *Cluster Comput.*, vol. 22, no. 1, pp. 1765–1775, 2019.
[6] C. Akasiadis, V. Pitsilis, and C. D. Spyropoulos, “A multi-protocol IoT platform based on open-source frameworks,” *Sensors*, vol. 19, no. 19, pp. 4217–4217, 2019.
[7] H. Harman, K. Chintamani, and P. Simoens, “Robot assistance in dynamic smart environments-a hierarchical continual planning in the now framework,” *Sensors*, vol. 19, no. 22, 2019, Art. no. 4856.
[8] J. He, Y. Zhang, J. Lu, M. Wu, and F. Huang, “Block-stream as a service: A more secure, nimble, and dynamically balanced cloud service model for ambient computing,” *IEEE Netw.*, vol. 32, no. 1, pp. 126–132, Jan./Feb. 2018.
[9] L. Wang, Z. Yu, D. Zhang, B. Guo, and C. H. Liu, “Heterogeneous multitask assignment in mobile crowdsensing using spatiotemporal correlation,” *IEEE Trans. Mobile Comput.*, vol. 18, no. 1, pp. 84–97, Jan. 2019.
[10] R. D. Pietro and F. Lombardi, “Virtualization technologies and cloud security: Advantages, issues, and perspectives,” in *From Database to Cyber Security*. Berlin, Germany: Springer, 2018, pp. 166–185.
[11] W. Huang, “Introduction of AMD advanced virtual interrupt controller,” in *XenSummit. AMD*, San Diego, CA, USA, 2012.
[12] J. Gandhi, M. D. Hill, and M. M. Swift, “Agile paging for efficient memory virtualization,” *IEEE Micro*, vol. 37, no. 3, pp. 80–86, Jun. 14, 2017.
[13] L. Abeni and D. Faggioli, “Using Xen and KVM as real-time hypervisors,” *J. Syst. Architecture*, vol. 106, 2020, Art. no. 101709.
[14] B. Guo et al., “Timeslot switching-based optical bypass in data center for intrarack elephant flow with an ultrafast DPDK-enabled timeslot allocator,” *J. Lightw. Technol.*, vol. 37, no. 10, pp. 2253–2260, May 2019.
[15] Z. Yang et al., “SPDK: A development kit to build high performance storage applications,” in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, 2017, pp. 154–161.
[16] R. Amsters and P. Slaets, “Turtlebot 3 as a robotics education platform,” in *Proc. Int. Conf. Robot. Educ. (RiE)*, Springer, 2019, pp. 170–181.
[17] M. Labbé and F. Michaud, “RTAB-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation,” *J. Field Robot.*, vol. 36, no. 2, pp. 416–446, 2019.
[18] “A video of IEEE systems journal,” [Video]. Accessed: Aug. 2022. [Online]. Available: https://youtu.be/COiO_ZGzLOE

Seongjoon Park received the BSE degree in electrical engineering from Korea University, Seoul, Korea, in 2015, and the Ph.D. degree in electrical and computer engineering from Korea University, Seoul, Korea, in 2022.

His current research interests include community wireless networks, network modeling and simulations, and virtualization techniques.

Kiseok Kim received the BSE degree in electronics engineering from Inha University, Incheon, Korea, in 2020. He is currently working toward the Ph.D. degree in electrical and computer engineering, Korea University, Seoul, Korea.

His current research interests include embedded Linux computing and Android platform.

Hwangnam Kim (Member, IEEE) received the BSE degree in computer engineering from the Pusan National University, Busan, Korea, in 1992, the MSE degree in computer engineering from Seoul National University, Seoul, Korea, in 1994, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Champaign, IL, USA, in 2004.

He is currently a Professor with the School of Electrical Engineering, Korea University, Seoul, Korea. His research interests include wireless networks, unmanned aerial systems (UAS), UAS traffic management, counter UAS systems, and Internet of Things.