Version of Record: https://www.sciencedirect.com/science/article/pii/S1389128621002917
Manuscript_e141456634863091344 4b0c7ba0585dc

User Input Enrichment via Sensing Devices

Yutao Tang*, Yue Li†, Qun Li†, Kun Sun‡, Haining Wang§, Zhengrui Qin¶
*Sam's Club Lab
Yutao.Tang@walmart.com
†Department of Computer Science, College of William and Mary
{yli,liqun}@cs.wm.edu
‡Department of Information Sciences and Technology, George Mason University
ksun3@gmu.edu
§Department of Electrical and Computer Engineering, Virginia Tech
hnw@vt.edu
¶School of Computer Science and Information Systems, Northwest Missouri State University
zqin@nwmissouri.edu

***Abstract*—Nowadays a user may have many electronic devices. However, these devices suffer from different resource and usage constraints. On one hand, mobile devices, such as smartphones, have short battery life, limited computing power, and small-sized display. On the other hand, more powerful devices such as desktops or smart TVs are bulky and lack motion-related input data. In this paper, we aim to integrate these two types of devices into one computing platform when both devices are accessible to a user, in which their individual advantages are combined for users' convenience. To this end, we develop a new user-centric paradigm on computing systems, called Application Execution with Sensing Input (AESIP), which can transparently inject sensing data to a powerful yet stationary device using an auxiliary mobile device. Not requiring any modification on the mobile device's OS and applications, AESIP supports all mobile-specific input data sources, such as touchscreen, gyroscope, and accelerometer. As one design goal of AESIP is to maximize the user's Quality of Experience (QoE), we tackle several usability challenges and enable richer functionality. We implement a prototype of AESIP on a Nexus 5, a Raspberry Pi and a desktop machine. Our performance evaluation shows that AESIP induces little latency and negligible bandwidth usage, and it significantly increases the battery life of a mobile device. We further conduct a user study to evaluate the usability of AESIP.**

I. Introduction

Digital electronic devices have significantly changed people's life for decades. These devices are designed for various purposes, and different traits bring them different advantages under certain circumstances. For instance, personal computers, tablets, and smart TVs are usually powerful with faster CPU and wider screen. However, these devices are also more stationary, such that the functionality is limited by the lack of many input sources, such as motion-related sensors including accelerometer and gravity sensor. Meanwhile, users also have more handy and portable devices, for example, mobile phones and smart watches, which are usually equipped with many sensors and support convenient input operations via the touchscreen. As a result, applications that leverage more input data are developed to provide richer functions. However, these mobile devices have short battery life, limited computing power, and smaller display size and resolution.

Obviously, the two different types of devices offer unique computing and communication services to users, but either of them has its own limitations. It would be an ideal scenario to seamlessly integrate them into one computing platform, as shown in Figure 1, where they are able to complement each other and deliver a significantly improved Quality of Experience (QoE) to users. People have developed many techniques to support hardware sharing. One common example is screen mirroring [1], [2], which allows users to freely run their mobile phone applications on a wider screen. However, this mechanism is battery hungry and demands persistent high bandwidth. Application-level screen sharing, such as Google Chromecast [3], enables mobile devices for controlling purpose only. However, they are only specific to their own applications, and thus cannot be applied to other apps. Some other techniques, such as VNC [4], [5], attempt to move the computation to a remote device and use another device to serve I/Os. However, VNC on a mobile platform usually does not support sensor data sharing. Besides, it introduces long latency and heavy dependence on the network conditions since screen frames are usually transmitted. RIO [6] is a systematic solution enabling the sharing of all the I/O between two mobile devices. However, it requires extensive modifications on the OSes of the two parties, and the use of Distributed Shared Memory (DSM) [7], [8] consumes much network resource.

In this paper, we present **A**pplication **E**xecution with **S**ensing **I**n**P**ut (AESIP), which provides a fully functional input sharing between two types of devices running Android without losing any sensing traits. AESIP aims to bind the two devices together to feature both powerful computation/display and handy maneuvers/operations. It usually runs the application on a relatively stationary but powerful machine (denote as the execution-device), such as a desktop, and meanwhile accepts input data from a portable gadget (denote as the sensing-device) like a smartphone or a smart watch. In other words, the sensing-device is used as an auxiliary input generator and the execution-device performs computation, output, and display tasks.

A major challenge of AESIP is how to transparently

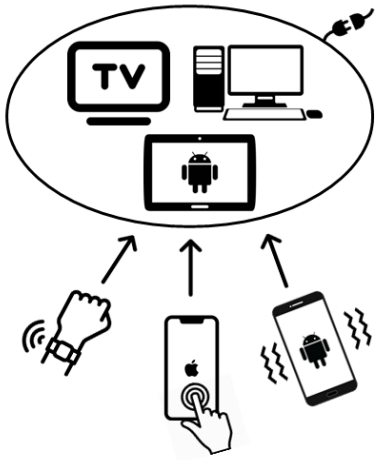


Fig. 1: Integrate two types of devices into one computing platform

decouple the input from the application execution without modifying the application. One of our design goals is that no modification should be made on existing applications. Such a feature significantly enhances the usability and applicability of AESIP, since it is impractical to assume all application developers are motivated to update their applications. To this end, we leverage the Android input subsystem and low-level Hardware Abstraction Layer (HAL) [9] of Android systems to provide virtual hardware components on the execution-device and feed the data from the sensing-device as if the data is locally generated. Note that our design does not need to modify the OS of the sensing-device. A user only needs to install a client application on his/her device as a normal application. Furthermore, virtual buttons are introduced in some applications to provide customized controlling. When virtual buttons are used, it is challenging for users to accurately locate the buttons on their not-displaying screen. We address the issue by statically mapping the virtual buttons back to the sensing-device, such that the buttons appear on the screen of the sensing-device.

The benefits of AESIP are four-fold. **First, AESIP is cost effective. It supports a large range of devices; the sensing-device can be any smartphone and smartwatch, and the execution-device can be both ARM-based and x86-based devices running Android system. This allows users to use their existing hardware devices and software apps while enjoying similar user experience provided by dedicated devices like Xbox Kinect.** Second, AESIP is application transparent. To run with AESIP, no modification is needed for apps, and a user just need to directly download apps from popular app store such as Google Play as usual. Third, AESIP combines the advantages of both sensing-device and execution-device while circumvents their disadvantages. For examples, users can play graphically demanding games with large-screen smart TVs on low-end resource-limited smartphones. Finally, AESIP bridges the gap between different OSes with different versions. Its loosely coupled interface between sensing-device and execution-device makes it easy to be applied to different devices.

We have implemented a prototype of AESIP, in which sensing-device runs on a Google Nexus 5 and execution-device is deployed and tested on both Raspberry Pi 4 (ARM-based) and a desktop (x86-based) [10] that run Android systems. The sensing-device supports four types of input, including touchpad, touchscreen, keyboard, and sensing data from the Nexus device. We conduct a comprehensive performance evaluation and demonstrate that AESIP has low latency, negligible network usage, and improved battery performance on the sensing-device. To evaluate the usability of AESIP, we conduct a user study with 32 participants. The participants are asked to use three different applications through AESIP and complete a questionnaire afterwards. The results show that AESIP generally has good usability.

The major contributions of this paper are summarized below.

- We propose AESIP, a user-centric paradigm that can share sensing input data from a sensing-device with an execution-device. AESIP adopts a light-weight and effective approach to transparently inject input data to the execution-device. Meanwhile, many usability issues are addressed for improving QoE.
- We implement a prototype of AESIP. Specifically, sensing-device runs on an off-the-shelf Nexus 5 mobile device and execution-device is tested on both ARM-based Raspberry Pi 4 and a X86-based desktop. Besides the core functions, we also implement additional functions such as a static input mapping mechanism.
- We evaluate the performance of AESIP, which shows low latency, low network bandwidth requirement, and reduced battery consumption.
- We evaluate the usability of AESIP through a user study, in which most participants have very good user experience.

The rest of the paper is organized as follows. Section II introduces the design goals and a high level view of the system architecture of AESIP. Section III details the design and implementation of AESIP, as well as enhanced functionality, and presents a prototype of AESIP on Android system. Section IV evaluates the performance of AESIP, including latency, data volume,and power consumption. Section V measures the usability of AESIP through a user study. Section VII surveys related works, and finally, Section VIII concludes the paper.

II. Scheme Overview

In this section, we first set our design goals when developing AESIP. These design goals are essential to build an effective and usable system, and help us make better design and implementation decisions. Afterwards, we illustrate a high-level view of the system architecture. Finally, we introduce several use cases that AESIP can improve user experience.

A. Design Goals

We first list the design goals of AESIP in the following.

Coverage. AESIP should provide abundant input methods to satisfy most applications' requirements. Users are able to easily interact with the applications running on the execution-device.

No modification on applications. The modification requirement on existing applications may largely cripple the adoption of AESIP as it is not practical to expect all developers to abide. AESIP does not require any modification on applications. In fact, all input signals are generated as if they are from the local device, such that the running applications retain full functionality.

High QoE. There could be many usability issues when using a mobile device as the input device. These issues have to be considered for achieving high QoE. We carefully address them under different scenarios.

Low overhead. Many applications are time sensitive, especially those involving intensive user interactions. A high overhead of AESIP may render the applications unusable. Therefore, AESIP should be responsive in a timely manner.

Easy deployment. AESIP can be easily deployed on user's devices (e.g., smartphone and smartwear). AESIP does not need any special privilege, such as root privilege. In addition, AESIP supports various types of input devices.

B. System Architecture Overview

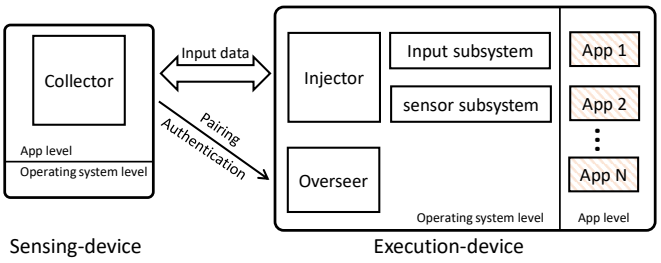


Fig. 2: A high level view of AESIP

Given the design goals above, we now present the architecture of AESIP. Figure 2 illustrates its outline. AESIP fundamentally involves two parties. One is the sensing-device and the other is the execution-device. The two parties communicate through network channels.

The collector is responsible for input data collection and is usually installed in a sensor-rich handy device (the sensing-device), such as a mobile phone. It provides a user-friendly interface to control the execution-device. The collector is installed in the form of a normal application. Namely, the collector does not need any special privilege so that any device can be used without modification. After authentication and pairing, a sensing-device establishes a connection to the execution-device and starts generating input data. The input data includes data from the touchscreen, touchpad, keyboard, and sensors. In certain scenarios, it also receives data from the execution-device. For example, in the case of sensing data transmission, the execution-device needs to first notify the sensing-device that a certain type of sensor data is needed.

The execution-device hosts the applications and provides the user with visual or aural feedback, which is usually an empowered but relatively stationary device, such as a personal computer and an Android TV box. The execution-device embeds an injector exercising as the tube transmitting data between the applications and the collector. It accepts input data from collector via the network, and then transparently injects the input data to the local OS as if the data is natively generated. The data injection is done by reconstructing system motion events and feeding them to the Android input subsystem, as well as leveraging HAL to deliver sensor data. The injector is able to simultaneously accept data from multiple collectors, allowing collective controlling from several users. More details are elaborated in Section III

C. Use Cases

In the following, we provide several typical use cases, in which AESIP can improve user's QoE.

Sensing-demanding games on Large screen: A larger screen is going to help users see more clearly and provide an immersive experience for games.AESIP can easily bring these features to reality. With AESIP, a user could play sensing-demanding games on an Android TV while controlling the game characters through the accelerometer data from his/her mobile phone.

Stable displaying: Today many people use E-health apps on their mobile devices for body training, where the apps provide guidance and count the movement using sensor data. It is hard for a user to focus on both the guidance on the screen and exercising with the device (to generate the sensing data for counting). With AESIP, the user could simply mount the sensing-device on her arm and look at the execution-device for instructions.

Multiple player. Given the fact that all inputs are aggregated and processed in the execution-device, AESIP supports using multiple sensing-devices to generate input data. Thus, AESIP allows multiple users to corporately control an application, which opens up a new control mechanism on which future applications, such as corporative gaming, may be built to provide higher QoE.

III. Implementation

Although straightforward in principle, the detailed design and implementation of AESIP are non-trivial. We build a prototype of AESIP on Android systems. Namely, both of the sensing-device and the execution-device are running Android as their OSes. There are two reasons we choose to do so. First, it is the most popular mobile OS world wide. Nowadays Android has been ported to many devices, such as Android TV, Tablets, and Android for x86 on Desktops or Laptops. More devices will soon start to adopt Android as their OS. Second, Android is open-sourced, allowing us to modify the OS at the execution-device side.

It is worth to mention that though the sensing-device implementation is on Android, it is easy to migrate to other OSes, such as the iOS and Windows Phone OS. This is because the interfaces between collector and injector are well defined with protocol buffer [11], which can serialize structured data regardless of the programming language and the platform. As long as the communication protocol between the sensing-device and the execution-device is followed, any mobile device can be used as a sensing-device.

In this section, we will elaborate on the detailed design and implementation of AESIP. We focus on how to handle each type of the input data since it is the key function of AESIP.

A. Authentication and Pairing

The very first step is to handshake. The sensing-device should be able to find the execution-device, but the execution-device should not allow any random sensing-device to connect. Thus, pairing the devices with authentication is needed to for the security purposes. Secure pairing with authentication is a well-studied field and can be implemented in many ways. We use a simple scheme that supports the management of multiple sensing-devices with the aid of an overseer.

Figure 2 illustrates this process. (1) The overseer on the execution-device is started as a daemon service, which listens to the connection from collector. (2) The collector authenticates itself to the overseer by connecting to the IP address and port of the overseer with a pre-set passcode. (3) If the authentication is successful, the overseer will then generate a session key, and send it to both the collector and the injector. (4) The collector and the injector uses the shared session key to encrypt/decrypt the data and communicate with each other. We use the platform-independent gRPC [12] to transmit the data.

B. Data Tubing

In order to provide the best user experience, selecting and properly handling the input data is critical to our system. We find four types of input can cover most of the demands. They are touchscreen, touchpad, keyboard, and sensors. Among the four types, three of them (except for the sensors) need user interactions, which means a user interface is needed. We set each of the types in a tab in the collector. Whenever the user needs a certain type of input, she switches to the tab and operates accordingly. In the following, we introduce each type of the input data, including why we need them and how to handle them.

1) Touchscreen: The touchscreen is the most natural and frequent input platform for applications. AESIP provides a user interface built on the Android *SurfaceView* class. When a user interacts with the *SurfaceView* object, the underlying Android service will detect the action on the screen and encapsulate it into a *MotionEvent* data structure. The data structure will be sent back to the *SurfaceView* object. In this step, the collector captures the user's interaction on the touchscreen.

To serialize the data in order to support devices with different OSes, the collector should collect 14 necessary data fields. Examples are *downTime* (the time when the user originally pressed down), *eventTime* (the time when this event happens), *action* (the action performed, such as ACTION_DOWN), etc. Note that the coordinates recorded are further scaled to reflect an accurate match between the SurfaceView instance and the screen of the execution-device.

When the data is collected from the sensing-device, the collector extracts the fields from the *MotionEvent* and sends them to the execution-device through wireless network. Upon receiving the data, the execution-device will construct a new *MotionEvent* object from the received data, and then inject the object to the Android input subsystem. Specifically, the execution-device gets an instance of the *InputManger* class, and then calls the *InputManager.injectInputEvent()* procedure and passes in the *MotionEvent* instance.

2) Touchpad: Featuring a usable touchscreen incurs a usability issue because the sensing-device merely generates input data without receiving any feedback. Consider the scenario when a user wishes to tap on an application icon shown on the execution-device, she may find it difficult to do so since the screen of the sensing-device does not show the icon and finding the correct position is challenging.

In order to tackle the problem, we adopt a solution in which we emulate a touchpad. The touchpad consists of a touching area and a button, just like the touchpad on a laptop, except that the button is located under the touching area for better usability. The touchpad input tab is shown in Figure 3. It controls a cursor that navigates on the display of the execution-device. This approach is practical since the touchpad cursor moves with relative coordinates and is much easier for a user to keep track of. For example, if a user wishes to launch an application, she needs to first move the touchpad cursor to the application icon and then tap on the button. Android for X86 has demonstrated that a mouse can handle most of the touch screen input. Our approach is equally powerful.

To implement the touchpad, the collector features another tab hosting a *SurfaceView* interface. The cursor is drawn as a *FloatingView* object on the screen of the execution-device. This object is on top of any other View objects and is controlled by the injector. The cursor does not interact with other objects, but only indicates a position that is under operation.

Note that the touchpad allows two types of operations: interactive and non-interactive. Interactive operations imply any operations that have a real impact on the execution-device, such as tapping, swiping, pressing, and dragging, while non-interactive operations simply move the cursor. The key difference is that interactive operations intend to operate on the screen of the execution-device and expect certain responses. Fundamentally, an interactive operation injects *MotionEvent* events to the OS of the execution-device while a non-interactive operation does not. The two types of operations can be easily differentiated by checking whether the button is pressed/tapped. For example, if a user presses the button

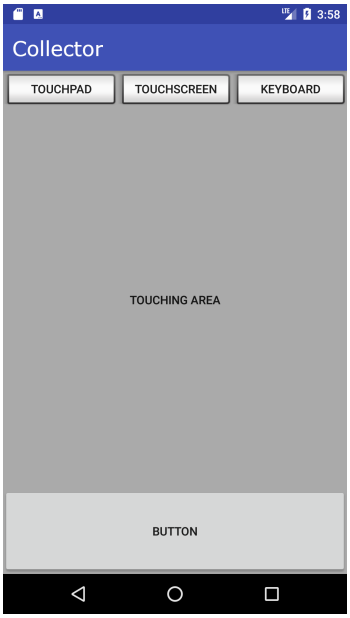


Fig. 3: Touchpad Input

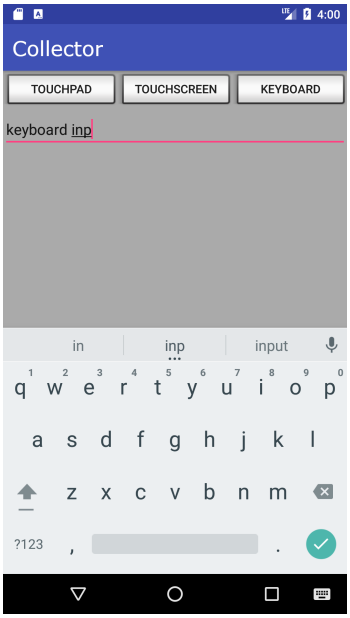


Fig. 4: Keyboard Input

while moving the cursor via sliding on the touching area of the touchpad, it represents a drag operation. All MotionEvents generated at the collector will be delivered to the injector, which then adjusts and injects these events to the local input subsystem. The adjustment is important since the coordinates of the MotionEvents should reference to the cursor's position. For instance, if the cursor is located at (x, y) at the display of the execution-device and the user performs a drag by pressing the button and sliding on the touching area at position (a, b) on the sensing-device's display. The collector will capture MotionEvent starting from (a, b) and a batch of following movements $(M = \{m_{1,2,\ldots,n}\}$, and $m_i = (m_{ix}, m_{iy}))$ from the sensing-device. The injector will adjust all points such that the operation is performed starting from (x, y) and the following movement is adjusted as $m'_i = (m_{ix} + y - a, m_{iy} + y - b)$.

3) Keyboard: Most mobile OSes (e.g., Android, iOS, and Windows Phone) provide software keyboard because they are touchscreen-based. Even with our touchscreen implementation, inputting on the execution-device is still hard and countering a user's input habit, since the user needs to manually move the cursor to each character and tap on the collector. Besides, it raises security concerns if the user inputs credentials on the execution-device, which is more likely to leak the credentials. For example, a guest may be able to peek the YouTube password of the user from the smart TV screen. For the sake of user experience and security, we enable a sensing-device side keyboard input generation channel, instead of simply relying on the built-in keyboard of the Android system on the execution-device. Specifically, we integrate another tab on the collector that has a built-in Android keyboard. The user switches to the tab and then input just like she normally does on a mobile device whenever she has the need.

To implement the keyboard, the collector provides an *EditText* View for the user. The user can leverage the soft keyboard on it to input data. Figure 4 shows a screenshot of the keyboard input tab. Every character keyed in will be captured by the collector, which in turn encapsulates it and directly sends it to the injector. The three necessary fields fed to the Protocol Buffer are time, character, and deviceID. Upon receiving the encapsulated data, the injector reconstructs the *KeyEvent* and injects it to the Android input subsystem, such that the applications are able to receive the event. Similarly, this *KeyEvent* will be injected by the *InputManager* Class of Android. Note that copy and paste operations on the soft keyboard are handled as the pasted content is typed in per character.

4) Sensor data: Contemporary mobile devices are featured with various sensors, including the accelerometer, gyroscope, and gravity sensor. The data generated by these sensors are widely used in many applications, such as exercising or gaming applications. However, motion-related sensor data cannot be easily generated by the execution-device due to its stationary nature. As such, using the sensing-device to generate sensor data for the execution-device is necessary.

In our implementation, all the sensor data is provided by the sensing-device. If the application running on the execution-device needs to use certain types of sensor data, it will first register to the local sensor service. The sensor service will notify the injector that forwards the request to collector. The collector then starts to collect sensor data on the local device by registering the same type of a listener on the sensing-device, and forwards the data back to the injector. Unlike other actions that could leverage existing Android APIs to inject data to the Android subsystem, there is no API to inject a sensor data structure. We manage to inject the data by leveraging the Hardware Abstraction Layer (HAL). Just as

the name suggests, HAL abstracts hardware and isolates the Android framework from the Linux kernel and the underlying hardware. In a typical Android system, the Linux kernel driver reads the input data from registers, and then sends the input data to HAL, which packages the data and further delivers it to the upper layer Android framework and applications. The definition of each type of input data should be pre-registered in HAL such that HAL knows how to handle the data. Such a system opportunity enables the injector to directly inject the input data to HAL as if the data is locally generated. As such, the applications can function normally without any modifications. AESIP leverages such a technique to achieve transparency.

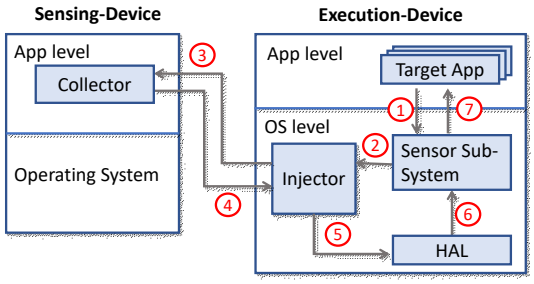


Fig. 5: Sensor data flow

Figure 5 illustrates this procedure. The application registers a listener to the sensor manager [13] when it needs to access a certain sensor. We modify the sensor manager such that whenever a new listener is registered, it sends a notification to the injector, which in turn signals the collector through networks. After receiving the signal, the collector itself registers a listener of the same sensor type to its local sensor manager and starts to collect data and send them to the execution-device. To feed the data to the application, we disable the original sensor HAL in the execution-device, which is unaware of the existence of the injector. Instead, we create our specially tailored HAL to facilitate the data flow. At system startup, our HAL will create a thread, which listens to the injector, and decode the sensor data inside. Then, the thread will wrap the raw sensor data to a data structure that conforms to the HAL sensor data definition and forward the data structure to the upper layer Android framework. As such, the application can fetch the sensor data without any modifications.

C. Functionality Enhancement

Apart from the main framework of AESIP, we apply enhancements to the system for better usability or possible richer functionality.

1) Virtual buttons: Nowadays, there are many mobile applications that provide customized input for application-specific operations. A typical use case is the virtual buttons in an application. While it is easy to operate on these virtual buttons when running on the mobile device, it incurs similar usability shortcoming with AESIP just like the touchscreen. Specifically, users may find it difficult to accurately locate the positions of these buttons on the mobile device screen since the visuals are only shown on the screen of the execution-device. Using a cursor is not ideal under this scenario for two reasons. First, multiple virtual buttons are frequently combined into more sophisticated actions, which is not achievable by using a cursor. Second, it also requires too much user effort to maneuver the cursor, especially in many real-time applications when prompt or combinations of operations are needed.

In order to resolve the problem, we develop a static input mapping mechanism to help the users locate these virtual buttons. Furthermore, it allows the users to freely layout these buttons on the mobile device such that the user experience may become even better. This is because virtual buttons are usually small and corner-positioned as the buttons should not take too much space and cover other content. We describe the mapping mechanism as follows.

Fig. 6: A virtual button example

The Touchscreen tab in the collector has a button specific to this function. Whenever the user switches to this tab and taps on the button, it sends a notification to the injector, which then takes a screen shot of the display of the execution-device. Afterwards, the injector transmits the captured picture back to the collector, which then displays this frame on the screen. The virtual buttons usually do not change their locations in an application, and thus, the users are able to locate the virtual buttons even though the displayed screen is static. Furthermore, due to the fact that the collector only needs to serve the virtual buttons, we further allow the user to customize the screen area to better fit her needs. Specifically, the user can keep the useful areas (where the virtual buttons are), and cut off all other content by cropping the screenshot. The saved space can then be used to better accommodate the buttons. In a full-fledged AESIP, some feedback features like pressing sound or vibration may also be added in order to prevent users from constantly losing focus on the buttons. Figure 6

shows an example of "Virtual button". In this example, we install "Streets of Rage 2" app on the execution-device. As we can see, the user sets up the "Virtual buttons" on the sensing-device, and use them to control the game running on execution-device.

2) Multiple Collectors: With our design, AESIP is able to support multiple input sources. For example, two users may cooperate on two sensing-devices to control a single application. This can be achieved by using the injector to accept data from multiple collectors. The injector maintains a data queue such that data from any collector is added to the queue in the order of time. Since different collectors have different IDs in the *MotionEvent* injected, the application can recognize the events from different operations. However, the application is not aware that the operations are from multiple devices. Instead, it simply considers that all operations are from the same device. For example, if two users slide on their touchscreens in different directions, the application may consider it a zooming operation.

One particular use case is to use the static input mapping mechanism to achieve cooperative gaming. The control can be extended to multiple sensing-devices when there are multiple virtual buttons on the screen. As such, these virtual buttons can be divided into multiple displays and operated by multiple users without interfering each other. One main advantage of this approach is that it does not need application awareness. On the other hand, the applicability could be limited. First, an application may not always have virtual buttons. The other types of input data are not supported. Second, most current applications are only designed to be used by a single user, so having two persons operating on the application may not be very useful at the current stage. However, this feature enriches the functionality of AESIP and future multi-user-aware applications may be developed to provide even richer functionality.

IV. Evaluation

We deploy a prototype of AESIP to evaluate its performance and overhead. The collector is installed on a Nexus 5 mobile phone (the sensing-device), and **the execution-device is deployed on a x86 virtual machine running on top of a desktop and a Raspberry Pi 4 platform, respectively.** The router we used provides a 300Mbps bandwidth. Note that since AESIP is intended to run in a Local Area Network (LAN) environment, 300Mbps is considered as a common bandwidth. We evaluate AESIP in terms of the input data transmission latency, data volume, and the power consumption on the sensing-device.

A. Latency

The latency of input data transmission is very important since high latency significantly degrades the usability. The latency mainly consists of two parts. The first part is the network delay. We record the time of an event being received by the collector, which is denoted as t_1. On the injector, we record the time that the same event is received from the network, which is denoted as t_2. Here $t_2 - t_1$ represents the delay of the network transmission. The second part is the delay induced by injecting the data to the application. To measure this delay, we record the time when the application receives the event (denoted as t_3). This is done by using a dummy application developed by us. Thus, $t_3 - t_2$ implies the delay caused by injecting the data event.

We conduct two independent experiments which use a Raspberry Pi 4 (ARM-based) and a Virtual machine on desktop (x86-based) as the execution-device, respectively. In each experiment, we send 100 events per class of touchscreen, touchpad, and keyboard, as well as 500 sensor events, from the collector to the target application that is running on the execution-device. The results of network latency and injection latency are shown in Figure 7 and Figure 8, respectively. As we can see from the figures, **the ARM-based execution-device has slightly shorter latency than the x86-based execution-device. The reason is that the ARM-based execution-device runs directly on the hardware and it can receive messages from sensing-device immediately while the x86-based execution-device runs on the virtual machine provided by Host OS which needs to forward messages to the virtual machine. In the two experiments,** the four types of input have small network latency, which is in average around 20-40ms. Besides, these input do not differ much. Similarly, the injection latency is also fairly small, around 40-50ms for touchscreen, touchpad, and keyboard events. The sensor events have significantly lower latency. This may be because the injection methods are different. The former three types of events are injected through the Android input subsystem. By contrast, sensor events are injected directly through HAL. The Android system may handle input from the two channels differently.

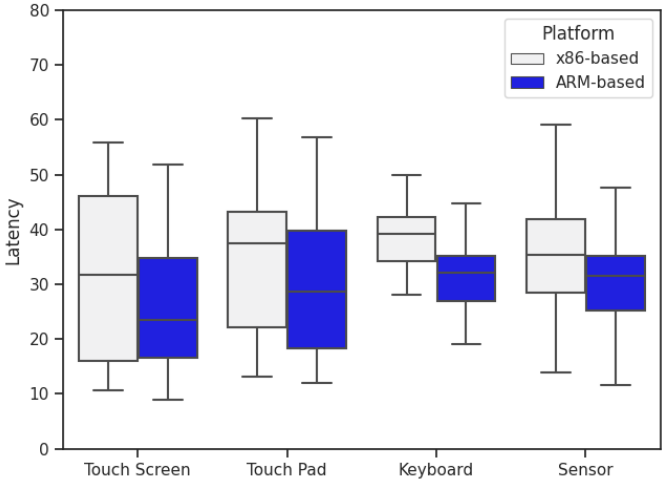


Fig. 7: Network Latency

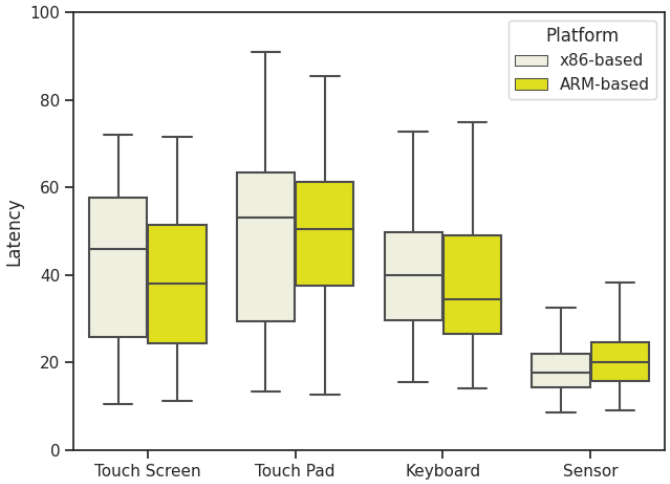


Fig. 8: Injection Latency

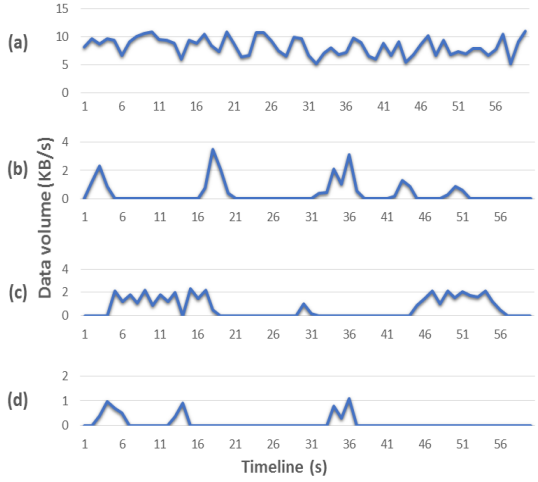


Fig. 9: Data Volume

(a) sensor data (b) touchscreen (c) touchpad (d) keyboard

B. Data Volume

With AESIP, the input data that is transmitted should not be too large; otherwise, the network may not be able to consume the data faster than data generation. In this case, the input data will suffer from much larger transmission latency. Even worse, the network may start to drop packets, making the application unresponsive. We show that AESIP has very small data volume, even if the sensor data is extensively read. We casually use the touchscreen, touchpad, and keyboard to control the execution-device within a 1-minute window. As for the sensors, we request data from four types of sensors, including the accelerometer, gyroscope, gravity sensor, and the light sensor. **Note that we also conduct Data Volume evaluation on x86 and ARM platforms, but the results on different platforms do not show a noticeable difference. Therefore, we only show the results on x86-based platform**

Figure 9 shows the data volume produced by the four input types. The spikes in the figures represent active operations. As the sensors are constantly being requested, there is always data in transmission. However, even though we have activated four sensors in the experiment, the data generated is at most around 10KBps. Besides sensors, the touchpad has more active operations, as shown in Figure 9c, because a user usually lies her finger on the screen to control the cursor. It can be seen that none of the other three input types produce more than 4KBps data. Compared to today's network bandwidth, such a small amount of data is negligible. Thus, we prove that AESIP has a very low requirement on network conditions as it consumes very small bandwidth.

C. Power consumption

One advantage of AESIP is its power efficiency on the sensing-device, which usually has a limited battery capacity. Since the computing and displaying tasks are offloaded to the execution-device, the sensing-device is largely released from battery draining. On the other hand, the execution-device is usually stationary and connected to a power outlet, such as the desktop we used, so the power is less of a concern. We measure the power consumption of the mobile device with and without AESIP when using two applications. Specifically, we record the power usage in a one-hour period when the device is idle, watching a Youtube clip, watching the same clip with AESIP, playing Doodle Jump, and playing Doodle Jump with AESIP. Doodle Jump is a leisure game where the main character keeps bouncing up against boards and avoids obstacles by moving left or right. The principle behind this game is that a user tilts the mobile phone to control the direction of the character and the tilting is reflected on the accelerometer readings.

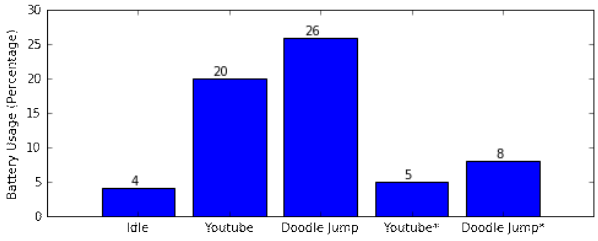


Fig. 10: Battery Consumption

"*" indicate that the application is used via AESIP

The results of the power consumption are shown in Figure 10. It is intuitive that the power consumption is the smallest when the mobile phone is idle. In the case of watching a Youtube video using AESIP, there is not much difference (only 1%) from the idle state, since the heavy operations are offloaded to the execution-device. Unlike playing Youtube video, in the case of Doodle Jump, the mobile phone still remains in a busy state, even with AESIP, due to the fact that it is actively retrieving data from the accelerometer. However, the power usage is still 69% less than playing it locally. **The reason is that some computation has been offloaded**

to the execution-device, and the power consumption of the limited data transfer overhead (less than 15k/s) is smaller than the power saving of computation offloading. Since users are much more sensitive to the power level of the sensing-device, using AESIP greatly alleviates the frustration caused by the quick battery drain.

In summary, the evaluation results show that: (1) AESIP has a very low network latency and injection latency on both x86-based and ARM-based platforms; (2) AESIP requires very small bandwidth; and (3) AESIP consumes less power than running the Apps directly on sensing-device.

V. User Study

Usability is a main consideration of AESIP. In order to evaluate how easy it is to use, we conduct a user study to gather feedback and comments from normal users. Toward this end, we recruit a total of 32 volunteers using AESIP. We set up a lab computer that runs Android for x86 with a 27-inch display, which serves as the execution-device. The sensing-device is a Nexus 5 mobile phone that runs Android 6. We pre-installed all the applications that are used in this study. Besides, The sensing-device and execution-device are paired prior to the study.

The volunteers consist of 14 females and 18 males. Since the user study is done in a school scale, most of our volunteers age 20-30 years old and have a bachelor's degree. We try to diversify the background of our volunteers by recruiting them from eight different fields of study, including mathematics, law, finance, economy, computer science, etc. The participants are asked to use three applications with AESIP, which are carefully selected and include all the input features that are provided. Each application takes several steps to use, and we describe them as follows.

Youtube. The first task is to use AESIP to watch a video clip on Youtube. Each participant is asked to open the Youtube application on the execution-device, log in using a test account managed by us, and randomly search a short video to watch.

Doodle Jump. Each participant is asked to play the Doodle Jump game. The participant needs to tilt the sensing-device to control the game character for jumping up.

Monster Truck. Monster Truck is another game that is controlled by the virtual buttons on the display of the execution-device. The game needs two hands to tap the virtual buttons on both sides of the mobile phone. We ask each participant to map the virtual buttons to the screen of the sensing-device and play the game.

TABLE I: User experience questionnaire on AESIP.

Q1. Rate your experience of using YouTube
Q2. Rate your experience of using Doodle Jump
Q3. Rate your experience of using Monster Truck
Q4. Is latency acceptable?
Q5. Are input tabs easy to use?
Q6. Rate the overall usability

We ask the participants to use each of three applications for five minutes. Afterwards, they are presented a short post-study questionnaire regarding the usability of AESIP. As shown in Table I, the questionnaire is anonymous and consists of six 6-point scale questions, where 6-point indicates strongly agree and 1-point indicates strongly disagree. The first three questions are about the experience using each specific application, and the latter three questions are more general. All participants successfully complete the study within 30 minutes.

Figure 11 shows the results of the questionnaire, from which we can see that different applications have different ratings of usability. Using Youtube receives the highest rating (more than 80% of participants agree that they have a better user experience on AESIP than on a mobile phone), since it does not need extensive user interaction and offers desirable high-resolution display experience. Playing Doodle Jump is comparably favored on AESIP. Several participants mention that tilting the mobile phone no longer affects the view point of the display, which is the reason why playing on AESIP has a better user experience. On the other hand, playing Monster Truck on AESIP is not as favored as the other two applications, though it is still considered usable. As mentioned by the participants, they sometimes lose the focus of their fingers on the virtual buttons. In this scenario, they need to keep switching their sights between the monitor and mobile phone. Meanwhile, four participants explicitly mention that the virtual buttons work well as long as you crop it correctly. As virtual buttons were associated with more feedback such as vibration or beeping sounds, the usability would be further improved.

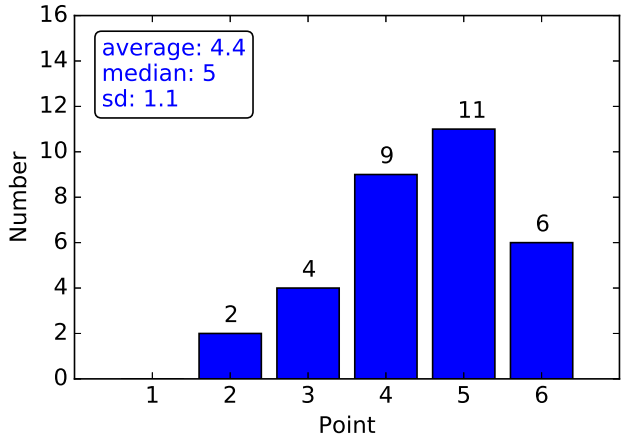
When asked if the latency of AESIP is acceptable, most participants (75%) agree or strongly agree. This is consistent with our performance evaluation result that AESIP introduces small latency. When asked about how easy the input tabs can be used, some participants have concerns when multi-finger operations or prompt consecutive operations are needed. In these cases, a touchpad may not be ideal. This is also the main reason why some users do not consider that the input is easy to use.

In summary, AESIP enables users with better user experience when running some applications, such as the less interactive application Youtube. While there is a slight degradation in usability in certain application scenarios, the overall usability of AESIP is still considered high by most of our participants.

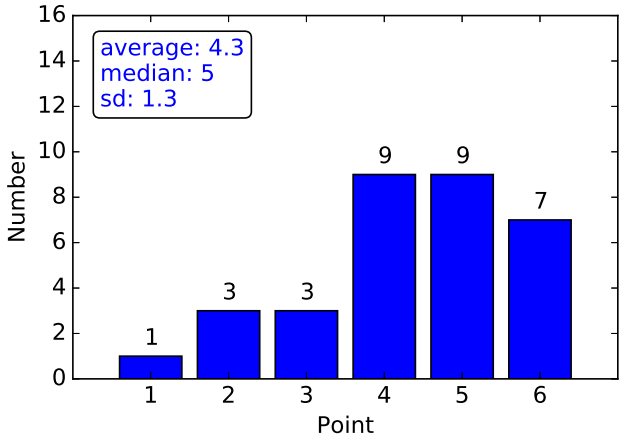
VI. Limitations

AESIP is not perfect so far, and it has several limitations that could be mitigated or removed in our future work.

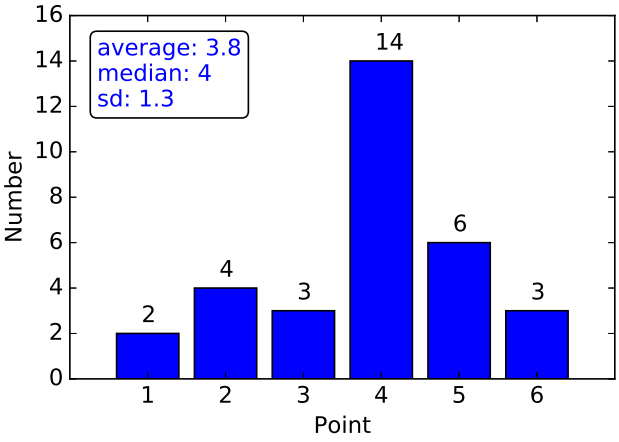
First, compared to other existing system like Xbox Kinect and Rio [6], AESIP is not very suitable for handling apps that require to frequently transfer large volume of data, e.g. video and audio, between the sensing-device and the execution-device, since large data transmission will drain more energy of the sensing-device and introduce larger latency, which may degrade user experience.



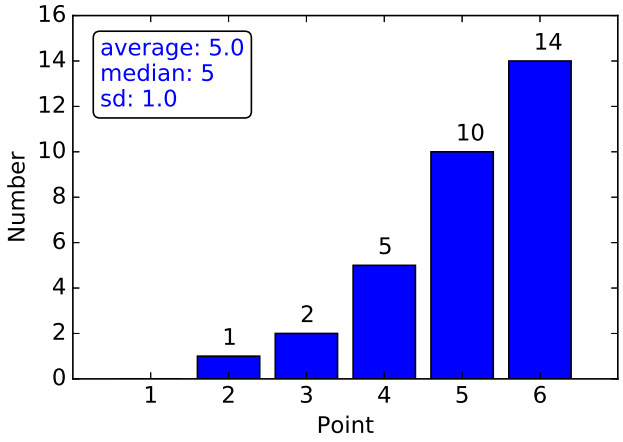
(a) Rate using Youtube



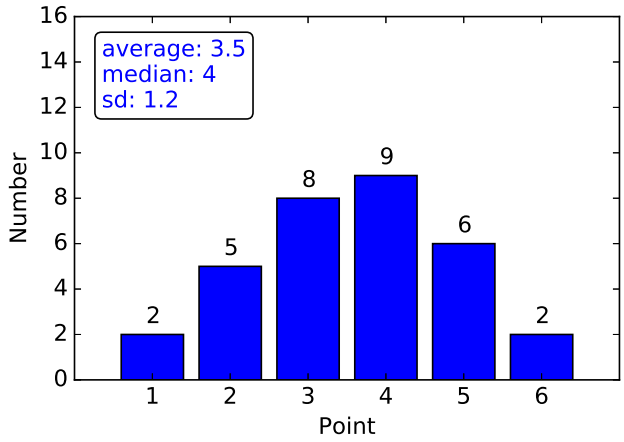
(b) Rate using Doodle Jump



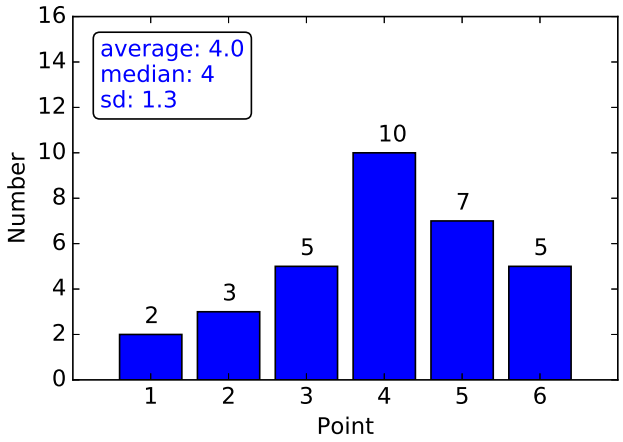
(c) Rate using Monster Truck



(d) Latency is Acceptable



(e) Input Tabs are Easy to Use



(f) Rate Overall Usability

Fig. 11: Survey Results

Second, AESIP does not provide good support for animated "Virtual button". All "Virtual buttons" displayed on the sensing-device are static, and the users need to redo the mapping process if the button changes. Even thought keeping the original mapping usually does not affect the operation for most cases, the different displays on sensing-device and execution-device may confuse users. Fortunately, this issue can be solved by monitoring the source animated "Virtual button" on the execution-device in real time, Once AESIP detects the content in the source area changes, it would update

the destination area on the sensing-device accordingly, we will leave this feature as our future work.

Third, most Android apps, especially games, are designed to play on smartphone, whose aspect ratio and resolution are usually different from those on execution-device. Therefore, when we run these apps on execution-device, they do not fully occupy the screen, leaving the unoccupied part black. This situation is even worse when apps only support portrait display.

Finally, in our current prototype implementation, the sensing-device is only tested on Android smartphone. We will test more devices, such as smart watch and iOS based system in the future.

VII. Related Work

Hardware sharing is an important topic that has been long studied in different levels. Many techniques have been developed to share resources, such as remote file systems [14], [15], memory sharing [16], [17], and network USB [18], [19], [20], [21]. Specifically, M2 [20] presents a data-centric solution that utilizes high-level device data to support I/O sharing between heterogeneous devices. FLUID [21] allows users to migrate or replicate individual user interfaces of a single app on multiple devices. All these methods enable remote resource access to build more advanced resource sharing tools.

Similar to AESIP, some other tools are used for controlling purposes, such as VNC [4], [5], [22], THINC [23], X server [24], and [25]. However, existing tools have limited usage on a mobile platform, such as inability to handle sensor data. For example, [25] designed a mobile app to use touch gestures to control the smart TV such as search and pick a movie, however it cannot handle sensor data. In order to address this problem, Rio [6] proposes a more low-level and systematic solution by using Distributed Shared Memory [7], [8]. What differentiates AESIP from Rio is that AESIP focuses more on application controlling and usability. The implementation of AESIP is also more light-weight and involves fewer modifications on the devices. Specifically, AESIP does not require any system modifications on the sensing-device while Rio needs to modify both parties.

Remote computation offloading for mobile devices nowadays have become a quite common practice [26]. Many solutions leverage a server or a cloud service to do the computation. Examples are Cloudlets [27], Clonecloud [28], MAUI [29], COMET [30], and others like [31], [32], [33], [34], [35], [36]. These techniques all attempt to move the heavy computation tasks to a more powerful platform, such as a remote server or a cloud service. There are also other remote computation schemes aiming at providing better security support [37], [38], [39], [40], [41]. For instance, SessionMagnifier [37] shares a browser session bettwen an untrusted computer and a mobile phone, where users can perform sensitive activities on the phone while normally browsing on the computer. TinMan [40] stores confidential records on a remote server and removes the sensitive operations on these records from the local device. CleanOS [39] and Keypad [38] secure a key by either evicting it or making the key retrieval auditable. Leveraging mobile devices for more secure user authentication on a computer is also frequently discussed [42], [43], [44], [45], where mobile devices support a second data channel. Furthermore, efficient communication among devices are also studied in an inter-connected environment [46].

VIII. Conclusion

Nowadays, many people have multiple electronic devices for different purposes, such as smart TVs, desktops, smart phones, smart watches, and so on. These devices are very different with respect to CPU power, power life, screen size, portability, and richness of input. It is of great interest to integrate or bridge them into one seamless computing platform such that better user experience can be offered to human users by overcoming the constraints of each individual device. In this paper, we propose AESIP, a new user-centric paradigm that allows a sensing-device to supply input data to a more powerful execution-device, such that different-purposed devices can complement each other with their own traits and advantages to provide much improved QoE (Quality of Experience) for users. AESIP supports the transparent transmission of four popular types of input data with user QoE considerations, namely touchscreen, touchpad, keyboard and sensor. We implement a prototype of AESIP on the Android platform and evaluate its performance in terms of data transmission latency, data volume, and power consumption on the sensing-device. Our results show that AESIP incurs very small overhead and significantly reduces the power drain on the sensing-device. We also evaluate its usability by conducting a user study involving 32 participants, and the results show that AESIP is considered to have high usability by most participants.

References

[1] mobzapp, "Screen stream mirroring," http://mobzapp.com/mirroring/index.html, 2016.

[2] M. McGill, J. Williamson, and S. A. Brewster, "Mirror, mirror, on the wall: collaborative screen-mirroring for small groups," in *Proceedings of the ACM International Conference on Interactive Experiences for TV and Online Video*. ACM, 2014.

[3] "Google chromecast," https://store.google.com/product/chromecast_ 2015.

[4] Wikipedia, "Virtual network computing," https://en.wikipedia.org/wiki/Virtual_Network_Computing, 2014.

[5] TightVNC, "Tightvnc software," http://www.tightvnc.com/, 2017.

[6] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A system solution for sharing i/o between mobile systems," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2014, pp. 259–272.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, *Implementation and performance of Munin*. ACM, 1991, vol. 25, no. 5.

[8] S. Zhou, M. Stumm, K. Li, and D. Wortman, "Heterogeneous distributed shared memory," *IEEE Transactions on parallel and distributed systems*, vol. 3, no. 5, pp. 540–554, 1992.

[9] "Android hardware abstraction layer documentation," https://source.android.com/devices/halref/index.html, Feburary 2015.

[10] C.-W. Huang, "Android-x86 project," http://www.android-x86.org/, November 2014.

[11] Google, "Google protocol buffer," https://developers.google.com/protocol-buffers/, November 2016.

[12] ——, "Introduction to grpc," https://grpc.io/docs/what-is-grpc/introduction/, November 2018.

[13] "Sensormanager," http://developer.android.com/reference/android/hardware/SensorManager.html, Feburary 2015.

[14] P. J. Leach and D. Naik, "A common internet file system (cifs/1.0) protocol," Internet-Draft, IETF, Tech. Rep., 1997.

[15] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 174–187.

[16] P. J. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems." in *USENIX Winter*, vol. 1994, 1994, pp. 23–36.

[17] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 4, no. 2, pp. 63–71, 1996.

[18] A. Hari, M. Jaitly, Y.-J. Chang, and A. Francini, "The swiss army smartphone: Cloud-based delivery of usb services," in *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*. ACM, 2011, p. 5.

[19] FabulaTech, "Usb over network," http://www.usb-over-network.com/, 2017.

[20] N. AlDuaij, A. Van't Hof, and J. Nieh, "Heterogeneous multi-mobile computing," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019, pp. 494–507.

[21] S. Oh, A. Kim, S. Lee, K. Lee, D. R. Jeong, S. Y. Ko, and I. Shin, "Fluid: Flexible user interface distribution for ubiquitous multi-device interaction," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.

[22] Z. Zhang, H. Cao, S. Su, and W. Li, "Energy aware virtual network migration," *IEEE Transactions on Cloud Computing*, 2020.

[23] R. A. Baratto, L. N. Kim, and J. Nieh, "Thinc: a virtual display architecture for thin-client computing," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 277–290.

[24] Xserver, "xserver," https://www.x.org/archive/current/doc/man/man1/Xserver.1.xhtml, 1987.

[25] J. Sun, Y. Li, L. Wang, X. Li, X. Ma, J. Xu, and G. Chen, "Controlling smart tvs using touch gestures on mobile devices," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE, 2015, pp. 1222–1229.

[26] Y. Geng, Y. Yang, and G. Cao, "Energy-efficient computation offloading for multicore-based mobile devices," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 46–54.

[27] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1109/MPRV.2009.82

[28] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[29] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *MobiSys '10*, 2010.

[30] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: code offload by migrating execution transparently," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 93–106.

[31] D. Chatzopoulos, C. B. Fernandez, S. Kosta, and P. Hui, "Offloading computations to mobile devices and cloudlets via an upgraded nfc communication protocol," *IEEE Transactions on Mobile Computing*, 2019.

[32] X. Meng, W. Wang, Y. Wang, V. K. Lau, and Z. Zhang, "Closed-form delay-optimal computation offloading in mobile edge computing systems," *arXiv preprint arXiv:1906.09762*, 2019.

[33] L. N. Huynh, Q.-V. Pham, X.-Q. Pham, T. D. Nguyen, M. D. Hossain, and E.-N. Huh, "Efficient computation offloading in multi-tier multi-access edge computing systems: A particle swarm optimization approach," *Applied Sciences*, vol. 10, no. 1, p. 203, 2020.

[34] S. Yu, X. Chen, L. Yang, D. Wu, M. Bennis, and J. Zhang, "Intelligent edge: Leveraging deep imitation learning for mobile edge computation offloading," *IEEE Wireless Communications*, vol. 27, no. 1, pp. 92–99, 2020.

[35] U. Saleem, Y. Liu, S. Jangsher, X. Tao, and Y. Li, "Latency minimization for d2d-enabled partial computation offloading in mobile edge computing," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 4, pp. 4472–4486, 2020.

[36] M. Du, Y. Wang, K. Ye, and C.-Z. Xu, "Algorithmics of cost-driven computation offloading in the edge-cloud environment," *IEEE Transactions on Computers*, 2020.

[37] C. Yue and H. Wang, "Sessionmagnifier: A simple approach to secure and convenient kiosk browsing," in *Proceedings of the 11th international conference on Ubiquitous computing*. ACM, 2009, pp. 125–134.

[38] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy, "Keypad: an auditing file system for theft-prone devices," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 1–16.

[39] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: Limiting mobile data exposure with idle eviction." in *OSDI*, vol. 12, 2012, pp. 77–91.

[40] Y. Xia, Y. Liu, C. Tan, M. Ma, H. Guan, B. Zang, and H. Chen, "Tinman: Eliminating confidential mobile data exposure with security oriented offloading," in *EuroSys '15*. New York, NY, USA: ACM, 2015, pp. 27:1–27:16.

[41] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 347–356.

[42] D. McCarney, D. Barrera, J. Clark, S. Chiasson, and P. C. van Oorschot, "Tapas: design, implementation, and usability evaluation of a password manager," in *ACSAC*. ACM, 2012.

[43] C. S. Nikolaos Karapanos, Claudio Marforio and S. Capkun, "Soundproof: Usable two-factor authentication based on ambient sound," in *Proc. USENIX Security*, 2015.

[44] L. Wang, Y. Li, and K. Sun, "Amnesia: A bilateral generative password manager," in *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE, 2016, pp. 313–322.

[45] Y. Li, H. Wang, and K. Sun, "Bluepass: A secure hand-free password manager," in *Securecomm*, 2017.

[46] R. Yu, G. Xue, and X. Zhang, "Application provisioning in fog computing-enabled internet-of-things: A network perspective," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 783–791.