FLUID-XP: Flexible User Interface Distribution for Cross-Platform Experience

Sunjae Lee[1]*, Hayeon Lee[1], Hoyoung Kim[1], Sangmin Lee[1], Jeong Woon Choi[1], Yuseung Lee[1]
Seono Lee[1], Ahyeon Kim[1], Jean Young Song[2], Sangeun Oh[3], Steven Y. Ko[4], Insik Shin[1]*
[1]KAIST, S. Korea [2]DGIST, S. Korea [3]Ajou University, S. Korea [4]Simon Fraser University, Burnaby, Canada
*{sunjae1294,insik.shin}@kaist.ac.kr

ABSTRACT

Being able to use a single app across multiple devices can bring novel experiences to the users in various domains including entertainment and productivity. For instance, a user of a video editing app would be able to use a smart pad as a canvas and a smartphone as a remote toolbox so that the toolbox does not occlude the canvas during editing. However, existing approaches do not properly support the single-app multi-device execution due to several limitations, including high development cost, device heterogeneity, and high performance requirement. In this paper, we introduce FLUID-XP, a novel cross-platform multi-device system that enables UIs of a single app to be executed across heterogeneous platforms, while overcoming the limitations of previous approaches. FLUID-XP provides flexible, efficient, and seamless interactions by addressing three main challenges: *i)* how to transparently enable a single-display app to use multiple displays, *ii)* how to distribute UIs across heterogeneous devices with minimal network traffic, and *iii)* how to optimize the UI distribution process when multiple UIs have different distribution requirements. Our experiments with a working prototype of FLUID-XP on Android confirm that FLUID-XP successfully supports a variety of unmodified real-world apps across heterogeneous platforms (Android, iOS, and Linux). We also conduct a lab study with 25 participants to demonstrate the effectiveness of FLUID-XP with real users.

CCS CONCEPTS

• **Human-centered computing → Interaction design**; **Ubiquitous and mobile computing**; **Interaction paradigms**.

KEYWORDS

Multi-device Mobile Platform; Multi-surface Computing; User Interface Distribution; Heterogeneous-Platform; Remote Display

ACM Reference Format:
Sunjae Lee[1]*, Hayeon Lee[1], Hoyoung Kim[1], Sangmin Lee[1], Jeong Woon Choi[1], Yuseung Lee[1] and Seono Lee[1], Ahyeon Kim[1], Jean Young Song[2], Sangeun Oh[3], Steven Y. Ko[4], Insik Shin[1]*. 2022. FLUID-XP: Flexible User Interface Distribution for Cross-Platform Experience. In *The 27th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '21), January 31-February 4, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3447993.3483245

ACM MobiCom '21, January 31-February 4, 2022, New Orleans, LA, USA

ACM ISBN 978-1-4503-8342-4/22/01…$15.00
https://doi.org/10.1145/3447993.3483245

1 INTRODUCTION

One of the major trends in mobile computing is the proliferation and diversification of smart devices. Today, US households own an average of 11 connected devices, including seven with screens to view media content (e.g., smartphones or TVs) [46]. Accordingly, the media consumption habit of ordinary people is transforming. For instance, 45% of US adults often or always use smart devices while watching TV [40].

With such a trend, a new way of interacting with multiple devices has been introduced: *Single-App, Multi-Device interaction paradigm*. Such a paradigm can harness the creation of novel interactions and practical use cases across different domains ranging from entertainment to productivity. For example, a user can watch a YouTube video on a large TV screen while live chatting with other viewers using a smartphone, or a photo editor may use a desktop for its canvas and a smart pad for its editing tools in a way that the toolboxes do not occlude the canvas while editing. Moreover, the single-app multi-device paradigm can foster the creation and consumption of new types of content. For instance, TV series creators may provide information or even multiple views for a single scene such that the viewers can watch the main scene on a TV while reading information of the characters (similar to Amazon Prime X-Ray) or looking around other views to find Easter eggs on their smartphones.

Several techniques have been proposed to support such multi-device interaction (see Section 9), including ScreenCasting [5, 16, 21, 49, 54], remote displays [17, 18, 43], custom multi-display apps [20, 37, 53], and FLUID [39]. Among these, FLUID stands out since it provides a unique capability that was previously not possible—it can distribute an app's *individual* UI elements, such as video playback buttons to different Android devices. This fine-grained UI sharing, combined with FLUID's support for *unmodified* existing apps, makes FLUID well-suited for mobile environments. However, FLUID has one limitation—*it is unable to support heterogeneous platforms.*

Addressing this limitation is crucial since different device types typically have different major platforms. For instance, smartphones and tablets come with Android and iOS, laptops with Windows, and smart TVs with Tizen, WebOS, and Android. Supporting these heterogeneous platforms is critical to comprehensively support multi-device interaction. However, FLUID's design that primarily uses UI object serialization and Remote Procedure Call (RPC) as a medium for cross-device interaction is not feasible in such an environment.

This paper proposes a new cross-platform UI distribution system, FLUID-XP (*Flexible User Interface Distribution for Cross-Platform Experience*)[1], that extends FLUID to cross-platform environments. To this end, FLUID-XP uses a new system design that is fundamentally different from that of FLUID. Specifically, FLUID-XP distributes the UI elements of an app to heterogeneous devices by solving the following three technical challenges:

C1. How to transparently enable a single-display app to use multiple displays.
C2. How to distribute UIs across heterogeneous devices with minimal network traffic.
C3. How to optimize the UI distribution process when multiple UIs have different distribution requirements.

C1: Transparent Multi-Display Support. In order to support unmodified existing apps, we need to transparently enable a single-display-oriented apps to use multiple displays. More specifically, we should be able to distribute an existing app's UIs onto multiple displays without modifying the app itself. This is challenging because an app's UI elements are tightly coupled with app logic; arbitrarily distributing the UI elements can cause unexpected side effects on the app's behavior. Therefore, FLUID-XP provides a single-display illusion to existing apps so that they can transparently use multiple displays (see Section 3).

C2: Cross-Platform Multi-Device Rendering. In order to support heterogeneous platforms, we must distribute UIs in a platform-independent fashion. The simplest and the most widely used method is to distribute UIs in the form of pixels. However, transmitting the pixels of multiple UI elements simultaneously at a high frequency (e.g., 30 fps) generates a large amount of data. This could cause a number of problems, including high latency and visual degradation, especially in mobile wireless networks where connections are subject to low bandwidth, high latency, and unstable connectivity. Therefore, FLUID-XP introduces a novel split-pipeline cross-device graphics architecture that encodes and transmits only the minimal set of pixels required to compose remote displays (see Section 4).

C3: Per-UI Optimization. When distributing UIs across heterogeneous devices and platforms, it is crucial to provide high-quality user experience. However, it is challenging to do so since different UIs can have various characteristics that are often conflicting. For instance, a video UI can be loss-tolerant but delay-sensitive, whereas a text UI can be delay-tolerant but loss-sensitive. To support such UI-dependent characteristics, FLUID-XP distributes each UI element individually with its own optimized cross-device graphics pipeline, which allows FLUID-XP to customize how it renders and transmits a UI element. For instance, FLUID-XP can use a different transport protocol suitable for each UI element based on their delay or loss sensitivity. FLUID-XP can also decide where to perform the rendering of a UI element (either locally or remotely) based on platform and performance requirements (see Section 5).

We have implemented a prototype of FLUID-XP and show that FLUID-XP can transparently provide single-app multi-device functionality for unmodified legacy Android apps across heterogeneous platforms (Android, iOS, and Linux). Our coverage evaluation with the prototype shows that FLUID-XP achieves complete transparency and high flexibility in supporting various single-app multi-device use cases using 19 legacy apps from Google Play [23] and one custom app. Our performance evaluation shows that FLUID-XP operates seamlessly across heterogeneous mobile platforms with low latency and high visual quality using only minimum network usage and power consumption compared to other approaches (e.g., screen mirroring). Lastly, we conduct a usability study to demonstrate the effectiveness of FLUID-XP with real users.

[1]See http://cps.kaist.ac.kr/fluid for our demo video.

2 SYSTEM OVERVIEW

2.1 Background

Virtual Display. A virtual display is a separate logical display whose contents are rendered onto an off-screen buffer instead of a physical display. Each virtual display has its own independent UI tree and graphics pipeline. A virtual display does not have a corresponding physical display itself, but it can be explicitly visualized via an external display or other software that handles virtual displays.

Graphics Pipeline. Android graphics pipeline is composed of five sequential stages: *traversal - clipping - rendering - compositing - displaying*. 1) The *traversal* stage traverses the UI tree to measure and calculate the exact size and layout position of each UI element. The root node of the UI tree has size information of the display and this is used to make the UI properly fit into the display's form factor. 2) The *clipping* stage selects a set of partial regions of the display that needs to be redrawn, called a *damaged area*. Then it creates a list of GPU commands for redrawing the damaged area. 3) The *rendering* stage re-orders and batches the generated GPU commands for optimization and then executes them to generate pixel data of the damaged area. 4) The *compositing* stage overlays the newly drawn pixel data of the damaged area to the previous frame. 5) The *displaying* stage visually displays the composited frame to the hardware screen.

2.2 System Workflow

FLUID-XP adopts a four-phase workflow in supporting the single-app multi-device interaction. It consists of a host device, where a user installs and runs regular apps, and one or more guest devices, where the user can distribute the UI elements of any app running in the foreground. FLUID-XP requires platform modification on the host device, but for guest devices, a user needs to install only one FLUID-XP wrapper app on each guest device without modifying the platform.

Pairing. Like any other wireless display system, FLUID-XP requires pairing between the host and guest devices. The host searches for nearby guest devices using Multicast DNS (mDNS). Upon discovery, a connection is established between the devices through password-based authentication.

UI Selection. FLUID-XP provides two methods for specifying which UI elements to distribute across devices. *i)* Prior to the execution of an app, users or developers can write down a list of UI elements and a UI layout for cross-device distribution on a pre-defined metadata file (e.g., `layout.xml`). *ii)* During the app execution, a user can enter a special *UI distribution mode* and dynamically select the UI elements to distribute by simply touching them on the screen.

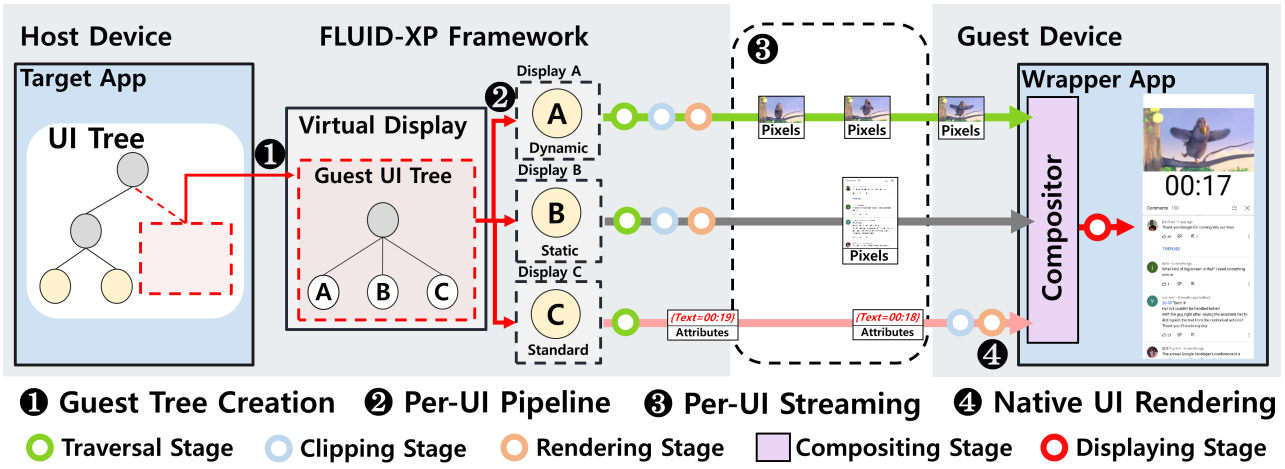

Figure 1: FLUID-XP Design Overview

UI Partitioning & Distribution. Once the device pairing and UI specification are done, the user accomplishes UI distribution by a single tapping of a button on the FLUID-XP wrapper app on a guest device. To this end, FLUID-XP mainly performs the following three tasks transparently: *i) Multi-display*: FLUID-XP internally prepares the UI elements to be distributed on multiple displays, *ii) Multi-device rendering*: FLUID-XP host device then renders the UI elements for multiple displays and devices, and *iii) UI casting and displaying*: FLUID-XP casts the result of the rendering to each guest device. The FLUID-XP wrapper app then displays received UI elements according to the layout.

UI Interaction. Once UI elements are distributed and displayed on one or more guest devices, FLUID-XP and its wrapper app enable users to interact with the UI elements on all devices without requiring any app code modification.

2.3 System Design Overview

In order to enable the workflow described in Section 2.2 and address the three challenges (**C1**, **C2**, and **C3**) listed in Section 1, FLUID-XP adopts three design choices—single-display illusion, split-pipeline cross-device graphics architecture, and per-UI optimized distribution (see Figure 1).

Single Display/UI-Tree Illusion. In order to provide transparency in expanding single-display app to multi-displays (**C1**), FLUID-XP gives an illusion to existing apps that they are using a single display when in fact they are using multiple displays. Apps do not need to be modified in any way to leverage multiple displays, and all UI elements behave as if they were placed on a single display. In order to accomplish this, FLUID-XP takes a new approach to how a UI tree is managed. In essence, FLUID-XP internally creates a *guest UI tree* for each display (❶ in Figure 1), determines which display a UI element is to be displayed on, and maps each UI element on the corresponding guest UI tree. In doing so, FLUID-XP still presents a single UI tree abstraction to the app and hides the existence of multiple guest UI trees. At the same time, the rendering system recognizes the guest UI trees and renders each UI tree independently. The advantages of this approach are that *i)* a developer can use the familiar single-display programming model when developing a new app that leverages multiple displays, *ii)* existing apps can be supported as-is, and *iii)* the rendering system can render each guest UI tree separately for a particular display as needed without worrying about how to deal with other UI elements that should not be displayed on that display.

Split-Pipeline Cross-Device Graphics Architecture. FLUID-XP aims to distribute different guest UI trees not only across multiple devices with the same platform (e.g., two Android devices) but also across devices with different platforms (e.g., Android and iOS) in the most network efficient manner (**C2**). In order to accomplish this, FLUID-XP examines the existing graphics pipeline and considers the entire design space as detailed in Section 4. Based on the analysis, ideal stages are identified in the graphics pipeline where the division of labor should occur between a host device and guest devices. Specifically, FLUID-XP splits the graphics pipeline so that *traversal-clipping-rendering* stages are executed on the host device, and the *compositing-displaying* stages are executed on the guest devices. FLUID-XP's design supports a wide range of heterogeneous platforms, while minimizing the amount of UI data transfer among devices without incurring extra computational overhead.

Per-UI Optimized Distribution. It is highly challenging to support a high-quality user experience in cross-device UI distribution, particularly when distributing multiple UIs with conflicting distribution requirements. User experience can be seriously degraded when the system fails to satisfy the distribution requirement, thereby hurting visual fidelity and interactivity compared to the case of a single device. To address this challenge (**C3**), FLUID-XP employs a flexible architecture that allows per-UI graphics pipeline processing (❷ in Figure 1). FLUID-XP leverages this flexible architecture to handle individual UI elements differently according to their characteristics, such as delay sensitivity, loss tolerance, and platform dependency. For instance, FLUID-XP uses a different network transmission protocol for each UI element suitable for its characteristics (❸ in Figure 1) and implements a FLUID-like distribution protocol that performs remote-side rendering for UI elements with less platform dependency (❹ in Figure 1). Our evaluation in Section 7 shows that the one-size-fits-all strategy does not meet the different requirements of individual UI elements and is therefore unable to provide high-quality UX.

3 TRANSPARENT MULTI-DISPLAY EXPANSION

Many mobile operating systems provide APIs for developing multi-display apps (e.g., Android: `Presentation`, iOS: `rootViewController`, Tizen: `Window`). However, utilizing such an API puts a significant burden on the developer; specifically, it requires them to manage multiple UI trees and write an app that can adapt its UI according to the presence of an external display. In fact, according to our analysis, only 1% of smartphone apps are utilizing such APIs[2].

This section describes how FLUID-XP transparently expands existing single-display apps to multi-display apps without enforcing extra engineering burdens. In particular, FLUID-XP partitions the UI elements of an app, designed for a single display, into multiple displays while maintaining the single-display programming abstraction for transparency.

3.1 Multi-Display Layout Interfaces

FLUID-XP provides two types of interface, allowing a user or developer to specify minimal information needed for UI distribution: which UI elements to distribute in which layout.

Runtime UI selection: This interface allows users to dynamically select UI elements to distribute during the execution of an app. A user can tap with three fingers to enter a special mode, then simply tap individual UI elements to select, then tap again with three fingers to exit the mode.

Metadata XML File: For more detailed specification, a developer can write a metadata XML file specifying which UI elements to distribute and how to lay them out. It is as simple as writing a layout XML file for Android apps.

3.2 Single-Display Illusion

FLUID-XP maintains a single-display illusion to an app so that all UI elements behave as if they were displayed on the same display. In order to accomplish this, FLUID-XP internally manages two types of UI trees. One type is a single *logical* UI tree that is exactly the same as the original UI tree. This UI tree is used as a single-display abstraction and this is the only UI tree that an app can interact with. The other type is *guest* UI trees and FLUID-XP creates one guest UI tree for each additional display. FLUID-XP creates these guest UI trees only when a user triggers UI distribution and uses them for UI rendering and distribution. If the user directly selects target UI elements at run time, FLUID-XP identifies them, creates corresponding guest UI trees, and maps the UI elements to their guest UI trees according to which display each UI element needs to be displayed. If there is a pre-defined layout file, FLUID-XP constructs guest UI trees and maps UI elements based on the file.

Conceptually, managing logical and guest UI trees means creating two nodes that represent the same UI element, one for the logical tree and another for a guest UI tree. This could lead to consistency issues across the two nodes and increased memory consumption. Thus, our implementation only creates a single node for each UI element and uses two types of edges to correctly manage logical and guest UI trees. The first type is *logical* edges and FLUID-XP uses them to represent a logical tree. The second type is *guest* edges and FLUID-XP uses them to represent guest UI trees. FLUID-XP then exposes only the logical edges to the app as the single-display abstraction, and internally uses the guest edges for UI rendering and distribution. We note that for a guest UI tree, FLUID-XP still needs to create additional nodes to form a proper tree, i.e., intermediate nodes and a root node, since only leaf nodes can represent visible UI elements.

[2]We investigated Google PlayStore's top 5,866 apps, and only 66 of them uses multi-display APIs at the time of writing

Stage	Solutions	Input Unit	OS Dependency
Traversal	[39]	UI Objects	O
Clipping	[10, 56]	UI Attributes	O
Rendering	[8, 36, 41]	Graphics Primitives	O
Compositing	FLUID-XP	Pixels (delta)	X
Displaying	[33, 34, 48]	Pixels (screen)	X
	[3, 5, 21]	Pixels (delta)	

Table 1: Possible offloading points for multi-device graphics pipeline

4 CROSS-PLATFORM MULTI-DEVICE GRAPHICS PIPELINE

FLUID-XP aims to adopt a multi-device graphics pipeline that supports a wide range of heterogeneous platforms with minimal network usage. Once UI elements are mapped to the guest UI tree for distribution, FLUID-XP renders the guest UI tree and displays them on guest devices. This section examines various options available to design a multi-device graphics pipeline and introduces techniques to support it efficiently, namely per-UI Rendering and guest-side composition.

4.1 Split-Pipeline Architecture

FLUID-XP employs a split-pipeline model for transparent cross-device rendering, and it chooses to split at the *compositing* stage in the graphics pipeline. This means that FLUID-XP runs traverse, clipping, and rendering on a host device and compositing and displaying on a guest device. The reason is that this design is most suitable in terms of both heterogeneous platform support and performance. As shown in Table 1, there are several candidates of the splitting point, and they are subject to different characteristics in various aspects, such as performance, platform-dependency, the range of UIs supported, and development cost.

The first three stages, traversal, clipping, and rendering, take platform-dependent data as input. Specifically, UI objects, UI attributes, and graphics primitives are all part of the input but cannot be shared directly between different platforms. Although one could develop a translation layer for each platform to support interoperability, it is extremely challenging to fully translate the unique features of individual platforms (e.g., developer-defined custom UIs) to maintain the same look-and-feel across heterogeneous platforms. Due to this reason, it is not ideal to split the graphics pipeline at any of these stages.

On the other hand, the last two stages, compositing and displaying, are suitable for cross-platform support since their input is *pixels*, a universal, platform-independent data format used by all display devices. Thus, if we split at any of these two stages, we can send raw pixels to a guest device and display the pixels on the guest device's screen to maintain the same look-and-feel. Note that the displaying stage takes all pixels of the entire screen as input, while

the compositing stage takes only the pixels that need to be updated, i.e., the delta between what is currently displayed and what needs to be displayed at the next screen refresh cycle.

Most solutions for cross-platform and multi-device interaction, such as mirroring tools [16, 21, 43], mainly split at the displaying stage, meaning that they only run the displaying stage on a guest device. This approach comes with less implementation complexity, but is subject to a large amount of pixel data to transfer between devices since the input to the displaying stage is all pixels of the entire screen. A few attempts [33, 34, 48] have been made to reduce the amount of data transfer by performing *diff analysis* to find the delta between consecutive display frames. However, the computational overhead for a diff analysis can be substantial due to the high resolution of modern mobile devices. For example, Pixel 4 XL has a resolution of 1440x3040, and it takes 80 ms to analyze the delta between two frames by a pixel-by-pixel comparison. If the resolution is scaled down to reduce the amount of computation, its accuracy could go down [28].

On the other hand, the input to the compositing stage is only the pixels that need to be updated. Thus, splitting at the compositing stage does not require any diff analysis and it is an ideal strategy to run the compositing and displaying stages on a guest device. However, a significant challenge is how to do this in a device-independent fashion, especially when dealing with a device that has a GPU. If there is a GPU, then it is the proprietary GPU software stack that runs both rendering and compositing as a black box, which in turn makes it impossible to split the graphics pipeline at the compositing stage.

To address this challenge, FLUID-XP leverages the *virtual display abstraction* that exists on mobile platforms under different names (e.g., *VirtualDisplay* on Android, *UIWindow* on iOS, etc). Each virtual display has an individual UI tree and runs a separate graphics pipeline with unique display settings (e.g., resolution, orientation). In addition, each graphics pipeline triggers its rendering process only when pixels on its corresponding display need to be updated. Thus, for every UI element to be distributed, FLUID-XP creates a new virtual display and attaches it to the virtual display's UI tree. Then, whenever a UI element is updated, only the graphics pipeline corresponding to the UI triggers the rendering. This generates only the pixels for the updated UI element, i.e., there is no need to perform a separate diff analysis to locate the updated pixels

One caveat is that the update granularity here is not at the level of pixels but at the level of virtual displays (i.e., UI element). This means that if FLUID-XP assigns a single UI element to a virtual display and a small portion of a UI element changes, FLUID-XP needs to send all the pixels that belong to the UI element. In addition, mobile platforms often restrict the number of per-app virtual displays to a certain number (e.g., six for Android). Thus, FLUID-XP sometimes has to assign multiple UI elements to a single virtual display. If that happens, FLUID-XP needs to send all pixels that belong to the virtual display even when a single UI element is updated. Yet, based on our observations, it is very rare for a UI element to change only small portion of itself. Instead, in most cases, a group of UIs working together to serve a common purpose change all at once. For instance, the comments UI in YouTube is in fact a group of dozens of text views, and when a user scrolls the comments UI, all text views move together as a single unit.

Depending on how UI elements are assigned on virtual displays, FLUID-XP sends pixels from multiple virtual displays to a single guest device so that they are processed together through the compositing and the display stages. Currently, FLUID-XP's default policy is to assign a group of UI elements with similar characteristics together on the same virtual display. For example, UI elements with low-update frequencies (e.g., buttons and images) and UI elements with high-update frequencies (e.g., videos). As we detail in Section 5, this enables per-UI optimization. Other, more sophisticated policies could be possible and we leave it as a future work.

Another benefit of using virtual displays is that we can use a guest device's resolution for each corresponding virtual display. Thus, once a host generates pixels, they are already suitable to display on each guest device without requiring re-translation. Once virtual displays finish generating new pixels for updated UI elements, FLUID-XP encodes the pixels and sends them to their corresponding guest devices along with their layouts as described in the next section.

4.2 Guest-side Composition & Interaction

Upon receiving encoded pixels and layout information of each virtual display, the FLUID-XP wrapper app on a guest device takes the following two steps to properly composite and display multiple streams of pixels on its display.

First, it creates a raw pixel container named a *surface* (following Android's terminology). A surface is essentially an image buffer and the FLUID-XP wrapper app uses the size and position from the layout received from the host when creating a surface. If a layout needs to change, the host can send a new layout along with new pixels, and the FLUID-XP wrapper app adjusts the layout of the corresponding surface accordingly each time. We note that a FLUID-XP host device renders each virtual display using the corresponding guest device's screen resolution. Therefore, there is no need to adjust or translate pixels for different resolutions. As we detail in Section 6, FLUID-XP performs some minimal pixel adaptation on guest devices since heterogeneous platforms use different coordinate systems and scaling methods to interpret a pixel.

Second, if the FLUID-XP wrapper app needs to display multiple (host-side) virtual displays as directed by the host, the FLUID-XP wrapper creates multiple surfaces (one for each virtual display) and composites them as a single frame to display. Each surface decodes its own pixels individually and can use a different streaming protocol as described in Section 5.

Furthermore, FLUID-XP enables users to seamlessly interact with UI elements distributed to all guest devices. To do so, the FLUID-XP wrapper app translates and forwards each input event to the host device. Then, the host device identifies the target UI using the layout information, and forwards the event to the corresponding virtual display.

5 PER-UI OPTIMIZATION

FLUID-XP adopts a flexible system architecture to handle individual UI elements differently based on their characteristics and requirements. This allows more fine-grained optimization of multi-device rendering by using a mixture of state-of-the-art multi-device rendering techniques (e.g., FLUID, Chromecast, streaming protocols).

This section describes how the FLUID-XP prototype implements different multi-device rendering techniques (i.e., per-UI streaming and native UI rendering) and utilizes them appropriately for UIs with different characteristics.

5.1 Per-UI Streaming

When distributing UIs across devices, we aim to support a high-quality UX. However, It is challenging to achieve both high visual quality and low latency, in particular, in mobile wireless networks due to non-negligible packet loss and delays. To resolve this issue, FLUID-XP takes a flexible approach that employs different transmission protocols for different sets of UI elements to satisfy various requirements.

Video streaming apps (e.g., YouTube) put together different types of UI elements with different performance requirements on the same screen. Dynamic UIs such as video windows are frequently updated and allow some distortion due to data loss. However, they are sensitive to delays such as video freezes. On the other hand, static UIs that are updated relatively slowly, such as video lists or chat windows, have opposite characteristics: delay-tolerant but loss-sensitive. In general, it is difficult to meet such various requirements while using only one transmission protocol. For instance, TCP, which provides a loss-free data transmission service, can support static UIs without any distortion but cannot satisfy high update frequencies of dynamic UIs due to its slow transmission speed. On the other hand, UDP can stream dynamic UIs at high transmission rates, but it is not suitable for streaming static UIs because image distortion may occur due to frequent traffic loss.

In this regard, we allow UIs with different characteristics to have different transmission protocols, thereby optimizing the transmission efficiency across all UIs on the screen. As explained in the previous section, FLUID-XP adopts per-UI graphics pipelines by assigning different subsets of UI elements to different virtual displays. We leverage such design to put a subset of UI elements with similar characteristics, such as dynamic or static UIs, onto the same virtual display. This way, when transferring the rendering results (i.e., pixels) of each different virtual display, FLUID-XP can employ a different transmission protocol suitable for the characteristics of UI elements that are assigned to the virtual display.

5.2 Native UI Rendering

FLUID-XP is designed primarily from a platform-independent perspective. However, some platform-specific features can easily lead to significant performance improvement. FLUID-XP is designed to easily extent in this direction. As an example, we introduce a native UI rendering technique.

There is a certain set of UI elements that are common to all GUI systems, namely *standard* UIs. Since these UIs have very well-defined looks and purposes, the UI attributes necessary for reproducing them are obvious. For instance, a button UI needs text and a color, and a text UI needs text, a text size, and a text color.

FLUID-XP can assign the standard UIs onto a single virtual display and offload the handling of the virtual display to a guest device from the clipping stage on. That is, the rendering of the standard UIs takes place on the guest device. To support this, FLUID-XP carries out a few tasks for standard UIs. The host device marks their virtual display as invisible to avoid host-side rendering and sends their graphical attributes to the guest device whenever any of them is updated. FLUID-XP wrapper app then renders the graphical attributes using the existing, native UI rendering method of the guest device. Such guest-side native UI rendering can significantly improve the user experience by avoiding the computation and networking costs of transferring pixel data. However, this approach cannot be applied to non-standard UI elements.

6 IMPLEMENTATION

User Input Handling. In order to support seamless user interaction, FLUID-XP implements input translation between devices. Any input given to a guest device is translated to appropriate input events and transmitted to the host device. We will not discuss the translation mechanism in detail for it is a well-known and widely-used technique (e.g., Android Emulator [24], SCRCPY [16], Vysor [54]).

H.264 RTP Streaming. When streaming each virtual display as a form of pixels, FLUID-XP uses H.264 video encoding [12], and Real-time Transport Protocol [42]. Specifically, for each virtual display, FLUID-XP creates a new set of H.264 hardware encoder and RTP packetizer. Then for each encoder and packetizer, upon receiving newly rendered raw pixels, FLUID-XP automatically encodes, packetizes, and transmits encoded pixels to the guest device. Then the guest device's wrapper app depacketizes, decodes, and displays each streamed H.264 packets at their designated positions. In doing so, it uses the native hardware decoder of the guest device.

Screen Coordinate & Frame Buffer Scaling. FLUID-XP redesigns a multi-device graphics pipeline toward cross-platform. Yet, there are some issues to be addressed to maintain the same shape across heterogeneous platforms. Different platforms have their own coordinate systems and scaling methods. For instance, Android uses density-independent pixels (dp) as the base coordinate system and scales items according to pixel density. iOS employs a different coordinate unit, *point*, and adopts a device-dependent scale factor to convert points to pixels. FLUID-XP implements a coordinate-conversion system that translates different coordinate systems between Android and iOS. FLUID-XP also implements frame buffer scaling to resolve differences in DPI (density-per-inch) between devices.

7 EVALUATION

We have implemented a FLUID-XP prototype to demonstrate and evaluate its full functionality across heterogeneous devices for unmodified existing apps. We have implemented the FLUID-XP host prototype on Android Open Source Project (AOSP) and the FLUID-XP guest wrapper app on three different platforms: Android, iOS, and Ubuntu.

For our experiments, we use Google Pixel 4 XL (AOSP v10) smartphone as a host device, and Google Pixel 4 XL (Android), Samsung Galaxy Tab S7 (Android), Apple iPhone 11 (iOS), and Lenovo ThinkPad X1 Carbon (Ubuntu) for guest devices. We connect all devices on the same Wi-Fi network with a single access point. The access point provides a throughput of 140 Mbps, and a round-trip time (RTT) with the median, average, and standard deviation of 4.27, 10.55, and 13.99 ms, respectively.

Type	Use Case Scenario	Corresponding UIs	App Name (Downloads)	App #	Network Usage		
					Whole Screen (MB)	Corresponding UI (MB)	Saving (%)
Live Chat	While watching live broadcast, participate in live chat using remote device	Chat Box	Twitch (100M)	1	34.24	4.93	85.61
			LiveMe Pro (1M)	2	70.05	0.77	98.90
Editing	Utilize painting & video editing tools from remote device	Tool Box	Infinite Painter (10M)	3	9.27	2.74	70.41
			VITA (10M)	4	31.36	8.36	73.36
Multi-View	Watch videos in multi-view environment by using remote devices	Multi-View Video	U+ Idol Live (1M)	5	44.30	0.47	98.95
			Sports Live (Custom Application)	6	31.93	26.11	18.21
Video Conferencing	While viewing one participant's screen in full screen, view another participant's screen on remote device	Participant's Screen	Google Meet (100M)	7	54.11	30.65	43.36
			Microsoft Teams (100M)	8	42.54	36.17	14.97
Video Controls	While watching video in full screen, control the video with remote device	Seek Bar & Play/Pause Button	Amazon Prime Video (100M)	9	59.60	21.27	64.31
			MX Player (500M)	10	57.51	2.22	96.13
Recommendation List	While watching video in full screen, scroll through the list of recommended videos using remote device	Recommendation List	Youtube (5B)	11	37.33	3.59	90.38
			Kakao TV (1M)	12	24.92	14.78	40.67
Maps	While using map in full screen, view detailed information about each location using remote device	Place Details	Google Maps (5B)	13	10.46	0.61	94.18
			Naver Map (50M)	14	12.35	4.81	61.02
Camera	When taking a group photo, place camera at a distance and push capture button using remote device	Camera Capture Button	SODA (10M)	15	60.87	0.49	99.19
			HD Camera (10M)	16	60.29	1.50	97.52
Collaboratory Usage	Let multiple users fill in information collaboratively through remote devices	Text Input Field	Booking.com (100M)	17	6.60	0.62	90.63
			Agoda (10M)	18	2.68	1.10	59.08
Login	Fill in login information using remote device	Login Field	Instagram (1B)	19	1.84	1.37	25.58
Interaction with Smart Watch	While viewing real-time running data through remote smart watch, store data and perform data computation on a mobile device.	Text Information Field	Nike Run Club (10M)	20	1.43	0.89	38.07

Table 2: A list of use case scenarios and apps for coverage test.

7.1 Coverage

In order to see how well FLUID-XP supports existing unmodified apps for transparent UI distribution, we evaluate 19 apps from Google Play and 1 proof-of-concept (POC) app ('Sports Live') in 11 use case scenarios as shown in Table 2. The POC app is included as an example of emerging multi-view video apps. We have confirmed with this experiment that FLUID-XP successfully supports all 20 apps in various multi-device use case scenarios in a platform-independent manner. To further explore the usability of FLUID-XP, we provide user study results with a few representative apps and use case scenarios in Section 8.

The 'Network Usage' column in Table 2 shows the total amount of data transferred for one minute when FLUID-XP streams the whole screen and when it does only a subset of UIs described in the 'Corresponding UIs' column. The 'Network Usage Saving' column shows how much saving FLUID-XP can achieve with a fined-grained streaming approach, reducing the unit of streaming from the whole screen to selective UI elements. Our experiment results show that FLUID-XP can significantly reduce the amount of network data transferred depending on use case scenarios.

7.2 Performance

We evaluate the performance of FLUID-XP for its seamless UI distribution. Unless stated otherwise, we repeat each experiment ten times, and use an H.264 encoder with a 1440x3040 resolution, a 8MB bit-rate, 30 fps, and one-second key-frame interval. Furthermore, to accurately measure the latency between the host and the guest, we synchronize the clocks of the two devices using a Desktop RTP Server through a wired connection, with an average clock error of 0.62 ms.

UI Streaming Latency. Figure 2 compares the streaming latency of FLUID-XP's fine-grained UI distribution and whole-screen distribution for each of the 20 apps listed in Table 2. We define latency as the additional time required to display UI elements on a guest device compared to the single-device case. We measure it as the delay from when FLUID-XP finishes host-side rendering to when FLUID-XP starts guest-side displaying. The UI distribution

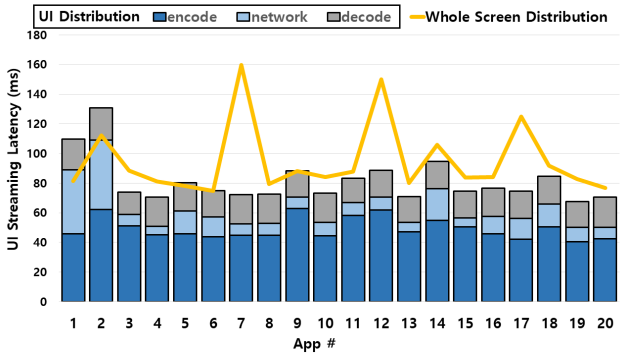

Figure 2: UI streaming latency

latency is represented as a bar graph, with the latency further broken down into encoding, decoding, and network transfer time. The whole-screen distribution latency is represented as a yellow line graph. We measure each latency component through system-level logging.

In most cases, UI distribution shows slightly lower streaming latency than the whole-screen distribution. Still, in some cases, one of the two shows noticeably higher latency than the other. This is because the streaming latency depends mainly on the size of each frame, and FLUID-XP does not take part in determining the frame size. Specifically, when displaying a UI element on a guest device, FLUID-XP re-scales the UI according to the resolution of the guest device and the preference of its user. Thus, the frame size of a UI element is entirely dependent on this re-scaling. For instance, FLUID-XP's use case scenario of Twitch and LiveMe Pro is to watch each app's chatbox UI in full screen on the guest device. This scales each app's chat UI to the full-screen resolution, which results in an increased frame size (40 KB and 48 KB, respectively) compared to the SCRCPY that does not re-scale (15 KB and 14 KB, respectively). Nevertheless, considering that the safe boundary for a satisfactory streaming experience is 160 ms [9], FLUID-XP distributes UI elements on different devices fast enough for interactive use.

Furthermore, we note that Android's H.264 encoder incurs extra VSync [19] wait latency that is unrelated to actual encoding, and

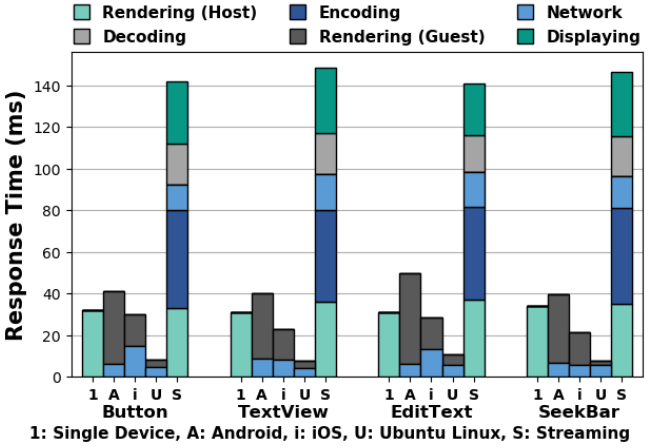

Figure 3: UI response time

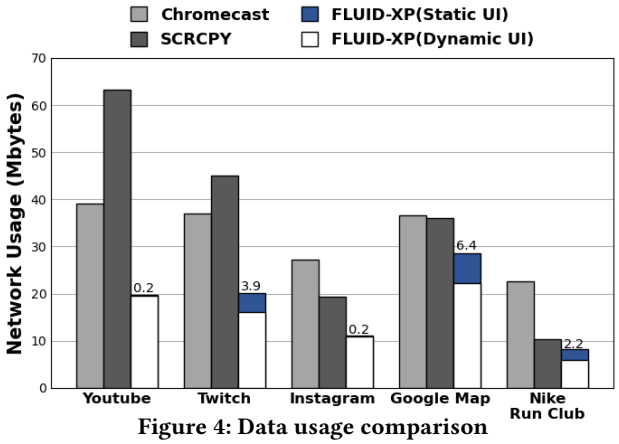

Figure 4: Data usage comparison

this is reflected on our encoding latency results. Android's H.264 incurs this extra latency because it starts the encoding of newly-rendered pixels only when it receives a new VSync signal, which is refreshed at a regular rate. Pixel 4 XL used in our experiments has a 60 Hz VSync refresh rate, which means that Android's H.264 encoder has an 8 ms of the average waiting time for a new VSync signal before it starts encoding. VSync rates are higher on other devices (e.g., 120 Hz for Samsung Galaxy 21 [45]) and FLUID-XP will have shorter overall encoding latency on those devices.

UI Response Time & Native UI Rendering. To evaluate the impact of native UI rendering, Figure 3 compares the average response times of different UI elements in five configurations: the single-device case (denoted by "1"), native UI rendering with Android, iOS, and Ubuntu guest devices ("A", "i", and "U"), and UI streaming with an Android guest device ("S"). We measure the response time as the delay from when the user makes a touch input to the guest device to when the guest device finishes displaying the result of the touch input, which is inclusive of the time required for host-side input event handling. The figure shows that native UI rendering reduces the response time significantly compared to the pixel streaming case, since it can avoid encoding and decoding. It is also interesting to see that the native UI rendering cases are comparable to the single-device case.

Figure 3 shows the difference in rendering latency across different platforms. The results show that the rendering latency is heavily affected by how each platform's graphics pipeline works. In the Android graphics pipeline, input event handling, UI tree traversal, and rendering are tightly synchronized with the periodic (60 Hz) VSync signals. Therefore, Android Graphics pipeline's average VSync wait time is 24 ms. However, iOS (120Hz input VSynz, 60Hz Render VSync) waits only 4 ms for input handling and 8 ms for the rendering, and therefore has average VSync wait time of 12 ms. Furthermore, the VSync latency for the Ubuntu is not included in the measurement because the VSync latency is unmeasurable in the user space of Ubuntu. Thus, we measure the rendering latency in our Ubuntu wrapper app as best as we can and approximate the precise latency. In order to evaluate the streaming performance of FLUID-XP across heterogeneous platforms, we also conduct UI streaming experiments with iOS and Ubuntu guest devices. For iOS, the average response times of Button, TextView, EditText, and SeekBar are 139.5, 135.3, 114.7, 124.4 ms, respectively, and those of Ubuntu are 99.93, 96.65, 96.21, and 91.19 ms each. As explained previously, the measurements for Ubuntu do not include rendering computation time and platform-specific delays. The experiment results are comparable to the Android case, and it shows that streaming pixels is a viable way to distribute UI across different platforms.

7.3 Network Usage

Figure 4 compares the amount of network usage across FLUID-XP and two popular screen streaming apps, Chromecast [21] and SCRCPY [16] (46.1k GitHub stars), for five scenarios from Table 2. The numbers shown in the graph indicate the amount of network data that FLUID-XP uses only for the static UIs. For fair comparison, we take the following two steps. First, since Chromecast and SCRCPY only support full screen mirroring, we distribute all UI elements with FLUID-XP as well. Second, we modify the source code of SCRCPY such that FLUID-XP and SCRCPY use the same encoder with the same configurations. On the other hand, our setup with Chromecast is not favorable to FLUID-XP or SCRCPY due to the limitation of Chromecast—it only supports the maximum resolution of 510x1080, which is 1/9 times smaller than the resolution for FLUID-XP and SCRCPY (1440x3040). With these configurations, we run our experiments for 1 minute for the three network-heavy apps (YouTube, Twitch, and Instagram) and 10 minutes for the other two apps (Google Maps and Nike Run Club).

Our results show that FLUID-XP consumes 20% to 69% lower network usage compared to other streaming solutions for the five scenarios. Though FLUID-XP and SCRCPY use the exact same encoder configuration, the main reason for such a difference is that SCRCPY streams the entire screen even when only one UI element is updated, while FLUID-XP only transmits the updated UI element. FLUID-XP reduces network traffic significantly when dynamic and static UIs have a large difference in update frequency as in the case of video apps (e.g., YouTube, Twitch, and Instagram). On the other hand, the benefit of FLUID-XP gets smaller for Google Maps and Nike Run Club, where the update frequencies of UIs are similar.

When compared to Chromecast, it is noteworthy that FLUID-XP has lower data usage in all five cases even though Chromecast supports only 1/9 resolution compared to FLUID-XP. Furthermore, considering that FLUID-XP is capable of selectively distributing

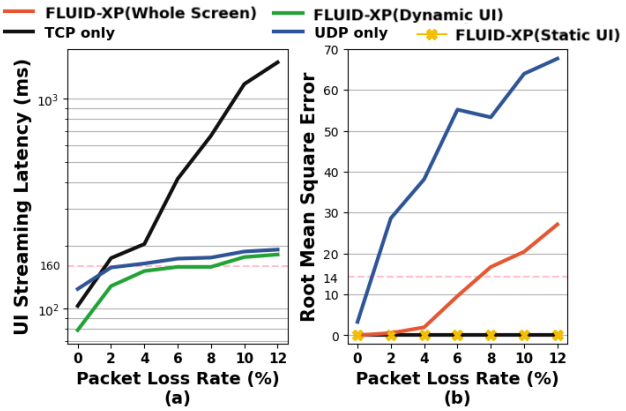

Figure 5: Streaming quality: (a) latency and (b) visual fidelity

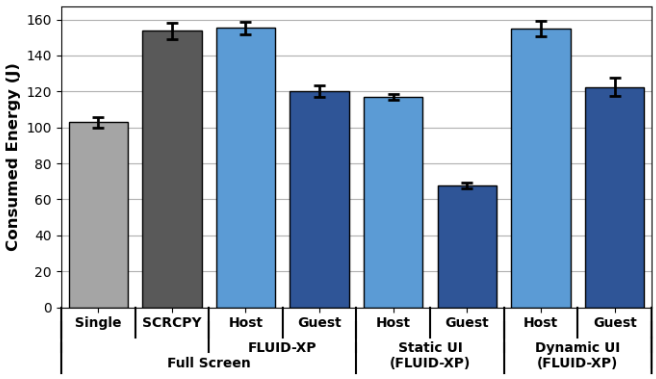

Figure 6: Energy Consumption

partial UI elements, FLUID-XP's real world data usage will be smaller depending on which UIs to distribute.

7.4 Streaming Quality Assessment

Figure 5 shows how well FLUID-XP can meet the different requirements of individual UI elements with per-UI streaming discussed in Section 5. We run our experiments with an in-house test app that consists of two UIs: one dynamic delay-sensitive UI (video) with an update frequency of 30 fps and one static loss-sensitive UI (video description text) that does not change after its first render. Our experiments are conducted under three different transmission schemes: the whole screen is encoded and transmitted using "TCP" only or "UDP" only, while FLUID-XP distributes UIs separately by transmitting the dynamic UI through UDP and the static UI through TCP. In order to evaluate the impact of the different schemes in different network states, we emulate packet loss using `iptables` [15] that drops packets according to the given packet loss rate. We use two metrics: the UI streaming latency and Mean Squared Error (MSE) [27, 52].

Figure 5 (a) shows that, as the packet loss rate increases, the latency for TCP increases exponentially, by far exceeding the average latency of 160 ms, a maximum packet delay latency required for satisfactory video streaming [9]. whereas UDP and FLUID-XP (Dynamic UI) maintain a delay around 160 ms even at the 12% packet loss rate. Figure 5 (b) shows that TCP and FLUID-XP (Static UI) maintain 0 MSE regardless of the packet loss rate, due to its lossless transmission mechanism. On the contrary, UDP's MSE increases rapidly due to the high distortion of each frame. 14 MSE, a value corresponding to 25 Peak Signal-to-Noise Ratio (PSNR), is an acceptable image quality threshold [50] and UDP exceeds the threshold even at the 1% loss rate. In the figure, FLUID-XP(Whole Screen) represents the MSE of the entire screen at each frame update of the dynamic UI. As we can see, its rate of increase is much smaller than that of pure UDP, and maintains excellent quality up to 4% packet loss rate. This is because FLUID-XP significantly reduces the network usage of the test app (up to 50%), similar to what FLUID-XP does for other apps discussed in Section 7.3. Furthermore, frame distortion due to packet loss only affects the dynamic UI.

Overall, we observe that TCP guarantees high visual fidelity for all network environments, but due to its exponentially-increasing delay, the user experience drops to the level where the total time the video freezes is sometimes longer than the playing time (depending on the loss rate, we see 14–91 s cumulative freeze durations for the frames that are longer than 160 ms). UDP can minimize such video freezing (4.4–26 s), but due to its high distortion, reading the video description (i.e., the static UI) is nearly impossible. FLUID-XP, on the other hand, benefits from the advantages of both protocols by transmitting the video UI through UDP, minimizing the video freeze (4.4–24 s), and at the same time, providing lossless quality to the video description text by transmitting it through TCP. In this particular scenario, after the 6% packet loss rate, both TCP and UDP cannot provide a proper user experience due to either delay or distortion, whereas FLUID-XP can still provide a moderate user experience.

7.5 Energy Consumption

We also measure the energy consumption of FLUID-XP. We calculate the energy consumption by reading the current and voltage information from the /sys/class/power_supply/battery/[3] [29], which is commonly used to measure the power consumption of smartphones with a non-removable battery[4, 11, 13, 25, 26, 31, 32] and has error rate less than 10%[4]. We use the display brightness of 100% on all devices and always start our experiment with a fully charged battery.

Figure 6 compares the average energy consumption of FLUID-XP host and guest devices when executing Twitch for one minute. Twitch has two main UIs (a video UI and a text chatbox UI), and we test three scenarios for cross-device distribution: using both UIs ("Fullscreen"), the video UI only ("Dynamic"), and the chat UI only ("Static"). Note that in the "Fullscreen" scenario, FLUID-XP distributes the UIs using the *Per-UI Streaming* technique (in Section 5). For comparison, we compute the average energy consumption of the host device when using only a single device ("Single") and when using SCRCPY to distribute the whole screen content. For SCRCPY, we use the same encoder configurations as in Section 7.3.

[3]Power Management Integrated Circuit (PMIC) periodically writes power related measurements (e.g., voltage, current, battery capacity) with error rate of ±1%.
[4]The PMIC used in our experiment has 3hz frequency. According to Nguyen et al. (2016), the error rate of energy consumption calculation using PMIC of frequency 2hz is less than 10% [38].

Application	Performance [0, 6]	Usefulness [-3, 3]	Satisfaction [0, 6]
YouTube	3.72	1.88	4.32
Twitch	5.28	2.28	4.96
Painter	4.96	1.88	4.56
Sports Live	5.41	2.18	5.05
Google Meet	4.88	2.20	5.12

Table 3: User Study Results Comparison between Apps

The results show that SCRCPY uses **49%** more energy than the single device use case. On the other hand, FLUID-XP's energy consumption differs depending on the scenario (i.e., which UI to distribute). In the case of distributing only the chatbox UI ("Static UI"), the FLUID-XP host device consumes only **13%** more energy, and the guest device consumes **34% less** energy than the single device use case. This is because FLUID-XP significantly reduces the frequency of encoding and network transmission when distributing a UI with a low update frequency (e.g., the chatbox UI with 1 fps).

When distributing the video UI ("Dynamic UI") or the entire screen, similar to SCRCPY, both FLUID-XP host and guest devices use more energy than the single-device use cases. However, considering that SCRCPY is a state-of-the-art screen distribution app, and its distribution mechanism is widely-used in other systems [5, 43, 44], we assume that this level of energy consumption in return for multi-device interaction is acceptable to the users. One thing to note is that when streaming the entire screen, FLUID-XP's per-UI streaming technique (in Section7.4) only consumes no extra energy compare to SCRCPY despite its relatively complex structure and high streaming quality.

8 USABILITY STUDY

We further evaluated the effectiveness of using FLUID-XP through a usability study. We selected five different apps to use on our Android host device. As guest devices, we used different OSes (Android, iOS, and Ubuntu) to create different user experiences. The goal of our evaluation is to assess the perceived performance, usefulness, and satisfaction of using FLUID-XP across different types of apps.

8.1 Participants and Study Procedure

We recruited 25 participants (10 female, 15 male, mean age 25.56, stdev=4.50, max=38, min=20) through an online community posting. Each study session was 40 minutes long, and the participants were paid approximately 9 USD. Each participant was given a task scenario for each app. They were asked to use all five apps in sequence, first using a single device and next using two devices powered by FLUID-XP. Users were given Likert scales to mark their scores for questions asking the perceived performance, usefulness, and satisfaction of using FLUID-XP. The study ended with a post-survey asking their thoughts on using FLUID-XP. To ensure safety during the COVID-19 pandemic, we asked all participants and our staff to wear masks during the study. We measured all participants' body temperatures, which were within the normal range. We note that the recruitment and the experiments were in accordance with our institution's IRB policies.

Application	**Most** useful (%)	**Least** useful (%)
YouTube	10.7	23.1
Twitch	21.4	23.1
Painter	21.4	34.6
Sports Live	14.3	3.8
Google Meet	32.1	15.4

Table 4: Participants Response on the Most and Least Useful Apps (Multiple choices were possible)

8.2 Study Scenarios

To evaluate FLUID-XP on diverse user scenarios, we carefully selected five mobile apps and gave a short task scenario for each of them. For YouTube, participants were asked to (1) search the "related video" list and read the comments while a video is playing in the full screen mode, (2) select one video from the list to play, and (3) turn back to the full screen mode. For Twitch, they were asked to (1) open the chat window while watching a streaming in the full screen mode and (2) live chat with other viewers. For Painter, they were asked to draw an animal using at least three different pens and three different colors. For Sports Live, they were asked to watch a soccer game using two features of the app: (1) a feature to watch highlights (e.g., scoring a goal) while watching the live game in the full screen mode and (2) a feature to watch a highlight in different view angles. For Google Meet, they were asked to attend an online class with their camera turned on to show their faces to the instructor.

8.3 Results and Findings

While there were variations in the scores across apps, participants thought FLUID-XP was high performance, useful, and satisfying. As shown in Table 3, all apps scored over 3.5 for the perceived performance and over 4 for satisfaction (0: *very bad*, 6: *very good*). Also, the usefulness scores were positive across all apps (-3: *not useful at all*, 3: *very useful*—compared to using a single device).

An interesting observation about the scores with YouTube is that participants still gave a positive usefulness score for using FLUID-XP while they were least satisfied with its performance. Satisfaction score was also high, scoring 4.32 out of 6. This indicates that even with low performance, users can still be satisfied and find it useful to use FLUID-XP.

In the post survey, we asked participants to choose the most and least useful apps from the study. The result is summarized in Table 4. All three scores (perceived performance, usefulness, and satisfaction) were high for the Sports Live app. Among 25 participants, only one participant thought it was the least useful scenario. 32.1% of the people responded that Google Meet was the most useful app to use in a multi-device environment. A notable finding is that although the average usefulness of using FLUID-XP for the Painter was the lowest, still 21.4% of the participants thought it was the most useful app. Overall, participants had diverse opinion on which app was the most or the least useful. This indicates that users may have varying preferences in how they use multiple devices for a single app. Therefore, providing flexibility through platforms like FLUID-XP would benefit from giving users the option to choose their display across devices.

Below, we share some of the quotes from the participants to help understand the use experience for each app. We note that

participant number is up to 27 because two participants did not finish the study due to technical issues.

YouTube: High usefulness and satisfaction scores despite low perceived performance. P9 responded that "Having to switch from the full screen to the small screen to browse the list of related videos was always bothersome on a single device. When using a second device (to display the list of related videos), it was convenient because I didn't have to." Similarly, P16 responded that "Reading comments is one of the most entertaining aspects of watching YouTube, and having a second device to read comments was satisfying enough to accept performance deterioration to some degree." These responses indicate that the participants valued the usefulness and satisfaction of using FLUID-XP to watch YouTube than the technical limitations with the performance.

Twitch: It was a great experience, but I don't think the interaction was practical. P1 responded that "It was convenient to write chat messages in the multi-device mode because the chat message and keyboard were separated from the video, making it easy to watch the video in full screen while typing." However, a few pointed out that it is not practical to use multi-device in watching entertaining videos because they usually watch them in their bed or free time while relaxing and it feels cumbersome to use two devices.

Painter: Contradicting results on the usefulness. P5 commented that "It is a great advantage that the tool box doesn't hide the canvas. Also, one of the strengths is that a user can use both hands to draw and change tools." Similarly, P8 picked Painter as the most useful app because "Among the five scenarios, Painter app best resolved the limitation of using a single device". On the other hand, P18 said that "I think it would be more useful if the screen of a single device is just larger, instead of moving the tool box to a different device."

Sports Live: No one hates to replay highlights while not missing the live game. P12 commented that "It occasionally happens in live sports that, while replaying a highlight, another highlight moment is happening live. (Using two screens) is good because I can watch replays without missing the live game."

Google Meet: The most useful scenario. P7 responded that "During online lectures due to COVID-19 pandemic, students access the class with more than one devices because some features don't work on their main device. This causes delay and connection failures. It seems useful because these limitations are resolved (when using two devices but still in a single session)."

8.4 Points of Discussion

We summarize two interesting discussion points from the user study. First, developers could use FLUID-XP to conduct rapid testing of single-app multi-device functionality for their apps in the early stages of their development.This is because FLUID-XP allows developers to immediately test the multi-device features of an unmodified app originally developed for a single device, enables low-cost development maintaining the single-device programming abstraction, and provides support for deploying it quickly to a wider range of real users regardless of their device platforms for early-stage feedback. While early versions may not be perfect in terms of performance and stabilization, users may still get a sense if the interaction is useful and satisfying for them as our participants did in our YouTube task scenario. Second, users may easily create personalized multi-device interactions using FLUID-XP thanks to its flexibility—allowing them to choose the interaction options that best fit for them. From the user study, we found that people have different perspectives on which multi-device interaction best fits for them. This implies that a one-size-fits-all pattern for a single-app multi-device interaction that is designed and provided by the developers may not satisfy all users. FLUID-XP can help overcome this limitation by letting users choose their own display.

Overall, we believe that our user study demonstrated the impact of the technology powering FLUID-XP to be useful in various other apps, such as stock trading apps, education apps, and gaming apps.

9 RELATED WORK

App-Level Multi-Display Support. Some apps [20, 22, 37, 47] are designed with multi-device support in mind. They are installed individually on different devices and use cloud servers [22, 47] or network pairing technology [20, 37] to synchronize among multiple instances. However, it incurs considerable development costs for multi-device support, and their multi-device use cases are limited to the scenarios determined during the development phase. On the other hand, FLUID-XP allows a user to select and distribute any UI elements flexibly without requiring code modification.

Existing UI Distribution Methods. There are a few approaches that support the cross-device distribution of selective UI elements. For instance, some *multi-window* desktop s (e.g., Adobe Photoshop [1], Microsoft Office [35]) allow to open multiple windows (UIs), and some of such windows can be deployed to other devices with wireless secondary display techniques (e.g., Apple Sidecar [7], Duet Display [14]). However, they are applicable to only a small set of multi-window apps. A recent study, FLUID [39], supports a wide range of apps, but its applicability is restricted to Android platform only. On the other hand, FLUID-XP can be applied to a wide range of apps across heterogeneous platforms.

Cross-device Graphics pipelines. FLUID [39] is a system-level solution for the Android platform that remotely renders UIs through UI object serialization and transmission. Since all of the computation is executed locally on guest devices, there is little or no interaction delay and no visual degradation. However, FLUID does not support heterogeneous platforms.

UIWear [56] and Sinter [10] perform guest-side rendering by only transporting the graphical attributes of a UI to a guest device. The UI is then reproduced on the guest-side to synchronize with the host device's display. Although they minimize network usage and support heterogeneous platforms, they fail to achieve visual fidelity since reproduced UIs differ from platform to platform. Furthermore, they do not support multi-media UIs such as video UIs, which are prevalent in today's mobile apps.

VNC [41], THINC [8], and RDP [36] are remote display systems that use custom display commands to transport display content over the network. Such mechanisms allow a host to simply forward the commands to a guest device and continue the rendering by executing the commands on the guest-side. However, although this method is efficient in representing the graphically simple UIs such as text and buttons, it suffers from performance degradation, especially when representing display-intensive multimedia apps such

as video playback [30]. This is because the above commands cannot exploit the temporal correlation inside the multimedia content.

Miao et al. (2016) [34], Tan et al. (2010) [48], and Virtualized Screen [33] propose pixel-based streaming approaches that only encode and transport updated pixels. This reduces bandwidth usage while maintaining visual fidelity across heterogeneous client devices. However, identifying updated pixels requires pixel-by-pixel analysis on each frame. Although they manage to reduce the computation overhead by classifying the pixels into pre-defined categories (e.g., text block, graphics block) and perform block-wise analysis, the overhead is too high to be practical in mobile environments with limited computational power.

Screen cast solutions such as Miracast [3], Airplay [5], Chromecast [21], and many others [16, 18, 43, 49, 54] capture the screen of one device and copies it onto a remote device's screen. This supports heterogeneity with simple implementation but suffers from high bandwidth usage and low visual fidelity due to the overhead of compressing and transmitting each and every full-screen frame as well as updating the entire frame buffer for each new frame.

10 DISCUSSION

Additional Per-UI Optimizations. The current prototype of FLUID-XP handles individual UI elements differently based on their own characteristics by optimizing only transmission protocols and rendering points. However, if we replace the existing proprietary encoder stack with a customized one, the flexible system architecture of FLUID-XP makes it possible to orthogonally apply additional per-UI optimization techniques related to encoding schemes. For example, FLUID-XP can optimize some encoding parameters such as bit rates, key-frame intervals, and resolutions, considering the features of each UI. We leave the exploration of such extra optimizations as part of our future work.

Unsupported Legacy Apps. FLUID-XP cannot support legacy apps that use separate UI rendering engines (e.g., Unity [51], WebKit [55]) such as 3D games or web apps because they do not create intermediate UI trees which are commonly used in mobile platforms. UI elements of such apps are managed through internal data structures provided from the app-level UI engines. This feature makes it impossible for FLUID-XP to manipulate UI trees and apply per-UI rendering and optimization techniques. However, the app-level engines also have similar UI tree structures and graphics pipelines internally, so if we can modify them, the proposed design of FLUID-XP can be comprehensively applied to them.

Scroll Interaction Optimization. In our usability study, we received feedback from participants that they experienced a slight delay in response while scrolling through the YouTube's list of related videos on a guest device. We then examined this phenomenon more closely and observed that such scrolling generates and transmits a bulk of frames. There may be two approaches to address this problem. One approach is pre-caching that speculatively renders the next frames of scrollable UIs, pushes them to the guest device, and caches them there. Typically, we can predict next frames because the user's scrolling direction determines them. Therefore, FLUID-XP can leverage pre-caching to hide some rendering, encoding/decoding and network delays. The second option is to reduce the number of frames generated by scrolling while skipping some frames. This raises an open question of how many frames can be skipped without compromising user experience. We leave it as future work to explore these approaches in depth.

Applying FLUID-XP to Other Systems. Although the current prototype of FLUID-XP is specialized for Android, its general design is applicable to other mobile platforms. FLUID-XP is designed under the key assumption that UI elements are managed by a tree structure and can be assigned to a virtual display with a multi-stage graphics pipeline. This assumption is a common design paradigm of GUI-based systems, so we can apply our system design to different systems while addressing the following issues. *i) Providing a single-display illusion.* Most mobile systems provide tree structures (e.g., UIView hierarchy [6] on iOS) to manage UI elements at the platform level, as in Android. Thus, we can provide a single-display illusion by duplicating the original tree as the two types of UI trees (i.e., logical & guest trees). *ii) Applying a split-pipeline architecture.* Most mobile systems provide special APIs (e.g., UIWindow [6] on iOS) for creating a virtual display. These APIs allow us to easily apply a split-pipeline model by assigning a target UI to a virtual display. Furthermore, it is possible to utilize the proposed per-UI optimizations for each UI.

Cross-device I/O Sharing. M2 [2] introduces a system design for sharing I/O between devices with heterogeneous platforms. The key idea is to create a virtual device driver that exchanges raw input data between devices. Inspired by this design, FLUID-XP implements a similar approach for handling user's key events, allowing full utilization of the native input methods of a guest device. For instance, when the EditText UI gets migrated to a guest device, the key input method of the guest device (e.g., voice input for a smartwatch, handwriting recognition for a tablet) is triggered. All other I/O devices (e.g., sensors, speakers) can be similarly shared between devices.

11 CONCLUSION

We have designed and implemented FLUID-XP, a novel multi-device system that supports innovative cross-device interaction across heterogeneous platforms (i.e., Android, iOS). FLUID-XP selectively partitions individual UI elements of unmodified apps and distributes them across multiple devices in a platform-independent way, enabling per-UI optimization according to the unique characteristics of individual UI elements. Our prototype implementation has proven that FLUID-XP successfully supports highly flexible and transparent UI distribution for a wide range of real-world apps, demonstrating high responsiveness and platform independence. In addition, our user study results show the effectiveness of FLUID-XP with real users based on their quantitative and qualitative feedback. We expect FLUID-XP to foster the development of novel apps that advance multi-device user experience to the next level.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers and shepherd for their insightful and constructive comments that helped us improve this paper. This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No.2020R1A2C2005479, No.2021R1F1A1063785, No.2020R1I1A1A0107238512, and ERC (NRF-2018R1A5A1059921)).

REFERENCES

[1] Adobe. 2021. Adobe Photoshop. https://www.adobe.com/products/photoshop.html.

[2] Naser AlDuaij, Alexander Van't Hof, and Jason Nieh. 2019. Heterogeneous Multi-Mobile Computing. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (Seoul, Republic of Korea) *(MobiSys '19)*. Association for Computing Machinery, New York, NY, USA, 494–507. https://doi.org/10.1145/3307334.3326096

[3] WIFI Alliance. 2021. Miracast. https://www.wi-fi.org/discover-wi-fi/miracast.

[4] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V Krishnamurthy, and Amit K Roy-Chowdhury. 2019. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SenSys '19)*. 96–109.

[5] Apple. 2021. Apple AirPlay. https://www.apple.com/airplay/.

[6] Apple. 2021. UIKit - Apple Developer Documentation. https://developer.apple.com/documentation/uikit/.

[7] Apple. 2021. Use your iPad as a second display for your Mac with Sidecar. https://support.apple.com/en-us/HT210380.

[8] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. 2005. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. Association for Computing Machinery, New York, NY, USA, 277–290. https://doi.org/10.1145/1095810.1095837

[9] Arkadiusz Biernacki and Kurt Tutschku. 2013. Performance of HTTP video streaming under different network conditions. *Multimedia Tools and Applications* 72, 2 (2013), 1143–1166. https://doi.org/10.1007/s11042-013-1424-x

[10] Syed Masum Billah, Donald E. Porter, and I. V. Ramakrishnan. 2016. Sinter: Low-Bandwidth Remote Access for the Visually-Impaired. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 27, 16 pages. https://doi.org/10.1145/2901318.2901335

[11] Yimin Chen, Xiaocong Jin, Jingchao Sun, Rui Zhang, and Yanchao Zhang. 2017. POWERFUL: Mobile app fingerprinting via power analysis. In *IEEE INFOCOM 2017*. IEEE, 1–9.

[12] Cisco. 2021. H.264. https://www.openh264.org/.

[13] Jürgen Cito, Julia Rubin, Phillip Stanley-Marbell, and Martin Rinard. 2016. Battery-aware transformations in mobile applications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 702–707.

[14] DuetDisplay. 2021. Duet Display. https://duetdisplay.com.

[15] Herve Eychenne. [n.d.]. iptables(8) - Linux man page. https://webkit.org/.

[16] Genymobile. 2021. Genymobile/scrcpy GitHub. https://github.com/Genymobile/scrcpy.

[17] Richard Goodwin. [n.d.]. How To Enable Android 10's Secret "Desktop Mode". https://www.knowyourmobile.com/user-guides/how-to-enable-android-10-desktop-mode/.

[18] Google. 2021. Android Auto. https://www.android.com/auto/.

[19] Google. 2021. Android VSync Document. https://source.android.com/devices/graphics/implement-vsync.

[20] Google. 2021. Cast from the YouTube app and YouTube.com. https://support.google.com/chromecast/answer/2995235?hl=en.

[21] Google. 2021. Google Chromecast. https://store.google.com/product/chromecast.

[22] Google. 2021. Google Docs. https://docs.google.com/.

[23] Google. 2021. Google Play. https://play.google.com/store.

[24] Google. 2021. Run apps on the Android Emulator. https://developer.android.com/studio/run/emulator.

[25] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys '14)*. 68–81.

[26] Liang He, Yu-Chih Tung, and Kang G Shin. 2017. iCharge: User-interactive charging of mobile devices. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*. 413–426.

[27] S. Hu, L. Jin, and C. C. J. Kuo. 2014. Compressed video quality assessment with modified MSE. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2014 Asia-Pacific*. 1–4. https://doi.org/10.1109/APSIPA.2014.7041643

[28] Chanyou Hwang, Saumay Pushp, Changyoung Koh, Jungpil Yoon, Yunxin Liu, Seungpyo Choi, and Junehwa Song. 2017. RAVEN: Perception-Aware Optimization of Power Consumption for Mobile Games. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah, USA) *(MobiCom '17)*. Association for Computing Machinery, New York, NY, USA, 422–434. https://doi.org/10.1145/3117811.3117841

[29] kernel.org. [n.d.]. sysfs-class-power. https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-class-power.

[30] Albert Lai and Jason Nieh. 2002. Limits of Wide-Area Thin-Client Computing. *Performance Evaluation Review* 30 (08 2002). https://doi.org/10.1145/511399.511363

[31] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. 2014. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 424–439.

[32] Youngmoon Lee, Liang He, and Kang G Shin. 2020. Causes and fixes of unexpected phone shutoffs. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys '20)*. 206–219.

[33] Y. Lu, S. Li, and H. Shen. 2011. Virtualized Screen: A Third Element for Cloud–Mobile Convergence. *IEEE MultiMedia* 18, 2 (2011), 4–11. https://doi.org/10.1109/MMUL.2011.33

[34] Dan Miao, Jingjing Fu, Yan Lu, Shipeng Li, and Chang Wen Chen. 2016. A High-Fidelity and Low-Interaction-Delay Screen Sharing System. *ACM Trans. Multimedia Comput. Commun. Appl.* 12, 3, Article 44 (May 2016), 23 pages. https://doi.org/10.1145/2897395

[35] Microsoft. 2021. Microsoft Office. https://www.microsoft.com/en-us/microsoft-365.

[36] Microsoft. 2021. Understanding the Remote Desktop Protocol (RDP). https://docs.microsoft.com/en-us/troubleshoot/windows-server/remote/understanding-remote-desktop-protocol.

[37] Netflix. 2021. How do I use my mobile device to watch Netflix on my TV? https://help.netflix.com/en/node/49.

[38] Quang-Huy Nguyen, Johannes Blobel, and Falko Dressler. 2016. Energy Consumption Measurements as a Basis for Computational Offloading for Android Smartphones. In *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. 24–31. https://doi.org/10.1109/CSE-EUC-DCABES.2016.157

[39] Sangeun Oh, Ahyeon Kim, Sunjae Lee, Kilho Lee, Dae R. Jeong, Steven Y. Ko, and Insik Shin. 2019. FLUID: Flexible User Interface Distribution for Ubiquitous Multi-Device Interaction. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article 42, 16 pages. https://doi.org/10.1145/3300061.3345443

[40] Sarah Perez. 2018. Nielsen: the second screen is booming as 45% often or always use devices while watching TV. https://techcrunch.com/2018/12/12/nielsen-the-second-screen-is-booming-as-45-often-or-always-use-devices-while-watching-tv/.

[41] realvnc. 2021. Real VNC. https://www.realvnc.com/en/.

[42] RFC. 2021. RFC 3350 - RTP: A Transport Protocol for Real-Time Applications. https://tools.ietf.org/html/rfc3550.

[43] Samsung. 2021. Samsung DeX. https://www.samsung.com/us/explore/dex/.

[44] Samsung. 2021. Samsung Dual Monitor Setup with Windows or Mac. https://www.samsung.com/us/support/answer/ANS00085802/.

[45] Samsung. 2021. Samsung Galaxy S21 Ultra Display Review. https://www.xda-developers.com/galaxy-s21-ultra-display-review/.

[46] Todd Spangler. 2019. U.S. Households Have an Average of 11 Connected Devices - And 5G Should Push That Even Higher. https://variety.com/2019/digital/news/u-s-households-have-an-average-of-11-connected-devices-and-5g-should-push-that-even-higher-1203431225/.

[47] Steam. 2021. Steam Remote Play. https://store.steampowered.com/remoteplay/.

[48] K. Tan, J. Gong, B. Wu, D. Chang, H. Li, Y. Hsiao, Y. Chen, S. Lo, Y. Chu, and J. Guo. 2010. A remote thin client system for real time multimedia streaming over VNC. In *2010 IEEE International Conference on Multimedia and Expo*. 992–997. https://doi.org/10.1109/ICME.2010.5582993

[49] TeamViewer. 2021. TeamViewer - The Remote Connectivity Software. https://www.teamviewer.com/en/.

[50] N. Thomos, N. V. Boulgouris, and M. G. Strintzis. 2006. Optimized transmission of JPEG2000 streams over wireless channels. *IEEE Transactions on Image Processing* 15, 1 (2006), 54–67. https://doi.org/10.1109/TIP.2005.860338

[51] Unity. 2021. Unity Real-Time Development Platform. https://unity.com/.

[52] Dr. D. Vatolin. 2021. Everything about the data compression. https://www.compression.ru/video/quality_measure/info.html.

[53] Videolan. 2021. VLC 3.0. https://www.videolan.org/vlc/releases/3.0.0.html.

[54] Vysor. 2021. Vysor.io. https://www.vysor.io/.

[55] Webkit. 2021. WebKit: A fast, open source web browser engine. https://webkit.org/.

[56] Jian Xu, Qingqing Cao, Aditya Prakash, Aruna Balasubramanian, and Donald E. Porter. 2017. UIWear: Easily Adapting User Interfaces for Wearable Devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah, USA) *(MobiCom '17)*. Association for Computing Machinery, New York, NY, USA, 369–382. https://doi.org/10.1145/3117811.3117819