FLUID: Flexible User Interface Distribution for Ubiquitous Multi-device Interaction

Sangeun Oh*, Ahyeon Kim*, Sunjae Lee*, Kilho Lee*
Dae R. Jeong*, Steven Y. Ko†, Insik Shin*
*KAIST, South Korea †University at Buffalo, The State University of New York, USA
{ohsang1213,nonnos,sunjae1294,khlee.cs,dae.r.jeong,insik.shin}@kaist.ac.kr
stevko@buffalo.edu

ABSTRACT

The growing trend of multi-device ownerships creates a need and an opportunity to use applications across multiple devices. However, in general, the current app development and usage still remain within the single-device paradigm, falling far short of user expectations. For example, it is currently not possible for a user to dynamically partition an existing live streaming app with chatting capabilities across different devices, such that she watches her favorite broadcast on her smart TV while real-time chatting on her smartphone.

In this paper, we present FLUID, a new Android-based multi-device platform that enables innovative ways of using multiple devices. FLUID aims to i) allow users to migrate or replicate individual user interfaces (UIs) of a single app on multiple devices (high flexibility), ii) require no additional development effort to support unmodified, legacy applications (ease of development), and iii) support a wide range of apps that follow the trend of using custom-made UIs (wide applicability). Previous approaches, such as screen mirroring, app migration, and customized apps utilizing multiple devices, do not satisfy those goals altogether. FLUID, on the other hand, meets the goals by carefully analyzing which UI states are necessary to correctly render UI objects, deploying only those states on different devices, supporting cross-device function calls transparently, and synchronizing the UI states of replicated UI objects across multiple devices. Our evaluation with 20 unmodified, real-world Android apps shows that FLUID can transparently support a wide range of apps and is fast enough for interactive use.

MobiCom '19, October 21–25, 2019, Los Cabos, Mexico

ACM ISBN 978-1-4503-6169-9/19/10…$15.00
https://doi.org/10.1145/3300061.3345443

CCS CONCEPTS

• **Human-centered computing** → **Graphical user interfaces**; **User interface management systems**; **Ubiquitous computing**; **Mobile computing**.

KEYWORDS

Multi-device Mobile Platform; Multi-surface Computing; User Interface Distribution;

ACM Reference Format:
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 (MobiCom '19), Oct. 21–25, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3300061.3345443

1 INTRODUCTION

Recent years have seen rapid development of mobile computing and IoT technologies, such as smartphones, tablets, smart home appliances, and smart cars. These smart devices are now a natural part of our everyday lives, and many users own and interact with multiple devices. According to Cisco's Trend Report [9], there will be 8.9 to 13.4 connected devices per user in Western Europe and North America by year 2020. It will be unsurprising to see the trend continue beyond that.

With the ownership of multiple devices, one can envision many interesting and useful use cases that allow users to interact with an application using the multiple screens of different devices (so-called *multi-surface computing*). More specifically, there are three driving factors that make multi-surfaces a more attractive interaction environment. *i*) *Multi-function:* a user can distribute different functionalities offered by a single application across multiple surfaces. For example, a user can watch a live-streaming video on a surface while chatting with other viewers in real-time using an in-app messenger on another surface, as illustrated in Figure 1. *ii*) *Multi-device:* different tasks can have different device preferences due to their distinct form factors. For example, a user can watch a movie on the larger screen of a smart TV while a

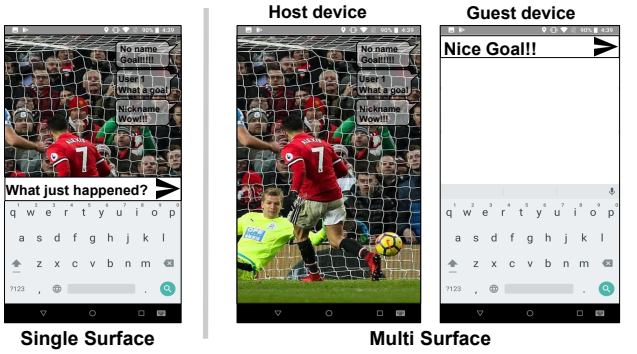

Figure 1: Live streaming app on single- and multi-surfaces; Multi-surfaces allow users to enjoy live streaming in full screen without interruption from keyboard.

video progress bar on her smartphone, which is easier to interact than a smart TV remote. *iii*) *Multi-user:* some tasks are associated with multiple users. For example, when booking flight tickets for a group of colleagues, a user can distribute personal information input boxes to other smartphones in order that each colleague can provide their personal information (i.e., passport number) on his/her own smartphone.

Previous solutions for supporting multi-surface use cases typically fall into one of the four categories with various limitations. First, some custom apps are developed with specific multi-surface usages in mind, such as cross-device continuous video display (e.g., Netflix [35]) and multi-user collaborative document editing (e.g., Google Docs [19]). Yet, this approach has limited applicability since it is only supported by custom applications. Second, several studies [53, 56] have introduced new programming models and/or development tools to extend existing applications for multi-surface operations, reducing a significant amount of development effort. However, their applicability can be limited to a small set of applications in practice, since they only maintain the same look-and-feel for stock Android UIs (user interfaces) but not for app-specific custom UIs. Using custom UIs in an application is a common practice for mobile application development. Our analysis of the top 100 apps of Google Play Store, conducted in May 2018, has revealed that all of the top 100 have their own custom UI components. Third, screen sharing (or mirroring) and app migration support a wider range of unmodified applications. The former copies a screen from one device to another usually with a larger screen or a better resolution (e.g., Vysor [10] and Chromecast [18]), while the latter moves an app process to an external device in the middle of execution and allows the app to use the surface of the external device [52]. However, they come with limited flexibility such that an app cannot fully exploit the advantages of multi-surface environments. Specifically, the screen sharing approach allows the app to display only the same screen content on multiple surfaces. It does not allow any other level of granularity, e.g., displaying different UI elements on different surfaces. On the other hand, the app migration approach allows the app to use only one surface at a time and not multiple surfaces simultaneously. For instance, it does not allow an app to display a video clip on a smart TV and a video progress bar on a smartphone at the same time.

In this paper, we propose a new multi-surface platform, FLUID (**FL**exible **UI** **D**istribution)[1], a systems solution that overcomes the limitations of previous approaches and aims to achieve the following design goals. *i*) *Flexibility:* the unit of deployment between different devices should be as fine-grained as possible to allow users to utilize multi-surfaces to their maximum flexibility. *ii*) *Ease of development (transparency):* it should not require any additional complexity for developing multi-surface applications, as compared to the development of single-device applications. It should also be able to support existing unmodified applications. *iii*) *Applicability:* it should be able to support a wide range of applications, including the applications using custom UI components. *iv*) *Responsiveness:* it should provide prompt response to user interaction across devices. For instance, users expect feedback on their inputs within 50 to 200 ms [5, 37, 46].

In order to satisfy the above design goals, FLUID allows users to dynamically select individual UI elements from the surface of one device (the host), migrate or replicate them on the surfaces of different devices (guests), and interact with them on all or some of the surfaces. In order to enable this, FLUID addresses the following three technical challenges. i) *UI partitioning and distribution*: FLUID carefully analyzes application and platform UIs to find a minimal and complete set of UI objects and their associated graphical states required to correctly render selected UI elements, and distributes only those objects and states across a host and guest devices. Our decision of performing local rendering on each device is beneficial in mobile wireless environments, since it not only saves the network bandwidth but also significantly reduces the number of network round-trips. ii) *Transparent distributed UI execution*: With distributed UIs, correct execution of an app requires cooperation between distributed states (UI objects and graphical states) and the rest of the states. FLUID transforms local method calls to RPCs (Remote Procedure Calls) to enable seamless cross-device execution for unmodified applications. iii) *UI state synchronization:* When a user replicates a single UI element on multiple surfaces and interacts with it simultaneously, FLUID synchronizes all the UI states involved to ensure the overall correctness of execution. This enables FLUID to provide greater flexibility with an option to replicate UI elements.

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

We prove the FLUID concept with a working prototype on Android using Google Pixel XL smartphones and Pixel C tablets. Our evaluation of the prototype shows that FLUID achieves high flexibility and complete transparency on 20 legacy apps. Our network and power consumption evaluation shows that our design optimizes network usage by an order of magnitude compared to other approaches (e.g., screen mirroring, app migration) for high responsiveness, and such optimized wireless link usage leads to low power consumption overhead as well.

2 USE CASES

This section discusses how users can benefit from using FLUID by presenting three categories of use cases. The current FLUID prototype supports these use cases already with existing applications.

Better usability. FLUID can be useful in a variety of scenarios, since it allows users to assign UI elements dynamically as they see fit according to device characteristics such as screen sizes and available modes of input. For instance, most live streaming apps (e.g., Twitch [28], LiveMe [29]) allow users to chat using a chat UI while watching videos using a video UI. When a user is chatting, the chat UI and keyboard is typically overlaid on top of the video UI, covering more than a half of the video UI on a smartphone. This can seriously degrade usability; the user may miss important moments such as soccer goal scenes in live sports broadcasts. With FLUID, the user can distribute the video and chat UIs across separate devices to watch live broadcasts on one device while chatting with others on another device. As another example, it is typically troublesome to precisely control the progress bar of a video player with a smart TV remote or enter destinations on a car navigation touch screen from a rear seat. With FLUID, a user can simply duplicate the video progress bar or the navigation input UI to a smartphone to enjoy the convenience of using a smartphone.

Collaborative use. In many cases, collaborating on a single task with multiple users is limited to output sharing (e.g., screen sharing [10, 15, 18]). Input collaboration, where multiple users give parallel input for a single task, is scarcely used except for a few cloud-based web applications (e.g., Google Docs [19]). With FLUID, users can transform any Android application into a collaborative app that supports both input collaboration and output sharing. For example, when filling in information for a group ticket reservation, it could be much troublesome if each person has to fill in their information sequentially on a single device (e.g., as the Expedia app expects [13]). With FLUID, users can now distribute personal information UIs to each user's device, and fill in the information in parallel.

Privacy protection. When sharing some information across different devices through app-level sharing or screen mirroring, there are risks of exposing privacy-sensitive information. For example, when mirroring the screen of a smartphone to a public device (i.e., smart TV) in a meeting room, a user may have to login or open a pattern lock at the risk of exposing the user's password or lock pattern to other colleagues because each letter she types in or the patterns being drawn are live-streamed to the public device. In addition, when sharing a specific email content or a specific photo to the public device, a user might have to expose the list of emails or photos as well. FLUID has an advantage of selectively deploying specific UI elements (such as an email content UI or a photo view UI) on a public device while leaving the other UIs (including an email list UI and a photo list UI) private on her smartphone.

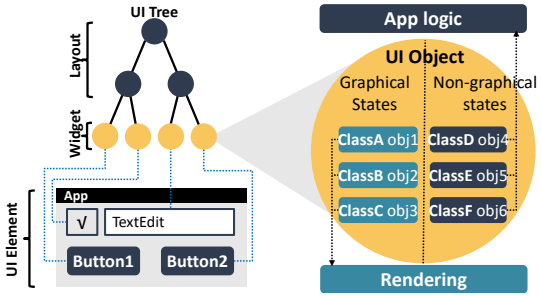

Figure 2: Android UI architecture

3 BACKGROUND

To concretely explore the design space for UI distribution, we target Android apps with graphical user interfaces (GUIs) running on the Android virtual machine (ART). Thus, it is important to understand how Android's UI sub-system works, especially regarding how Android organizes and renders UI elements. This section provides a brief overview for that, and defines our terminology used throughout the paper.

UI architecture. As shown in Figure 2, an app consists of a collection of *UI elements*, e.g., text input windows and buttons; from a user's perspective, a UI element is the smallest unit which users can interact with an application. Android has two types of UI objects, *layouts* and *widgets*, and maintains them in a tree structure (called a *UI tree*) per application. A widget is a graphical component that has a one-to-one mapping to a UI element (e.g., a button), and a layout is a container that manages how its child UI objects are positioned on their screen. Each UI object maintains a set of states that fall into two categories; i) *graphical* states are the data required and accessed during rendering (e.g., color, text, picture, animation), and ii) *non-graphical* states are all other data unrelated to rendering, typically used for app logic (e.g., event listeners). Note that the graphical states are typically implemented as objects, and we refer to those objects as *UI resource objects* (in short, *UI resources*). Each UI element is then associated with a set of UI objects and UI resources.

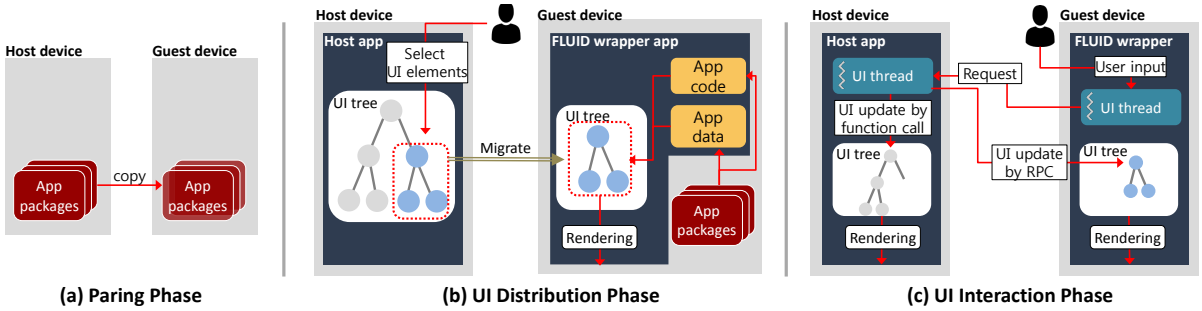

Figure 3: FLUID architecture and workflow overview

UI thread. Each app has a *UI thread* that manages the UI tree and updates the states of all UI objects. For example, when a user clicks a button, the UI thread triggers the execution of the event handler associated with the button clicked, and the event handler can change the graphical state of the button (e.g., color). Note that such UI object and resource updates are done by method calls, following the OOP encapsulation principle.

Renderer thread. Each Android app has a separate *renderer thread* to render UI objects. When the UI thread requests to draw UIs, the renderer thread traverses through the UI tree, from the root to leaf nodes, and executes rendering-related methods. We will elaborate on this in Section 5.1.

4 FLUID: SYSTEM OVERVIEW

Motivated by the use cases described in Section 2, we present a systems solution, FLUID, that allows a user to interact with a single unmodified app across multiple surfaces concurrently. We use a UI element as the unit of distribution since it is the smallest unit of functionality that users can use selectively. This section presents an overview of FLUID.

4.1 Workflow

Figure 3 shows FLUID's three-phase workflow.

Pairing. FLUID first arranges pairing between a group of trusted devices (a host and guest devices). The host device searches nearby and shows a list of potential guest devices, and a user simply chooses the guests from the list along with a set of apps for multi-surface interaction. Upon a user selection, the host device sends the app package files (i.e., APKs) of the selected apps to each of the selected guest devices. After that, this process is no longer required unless APK files are updated on the host device.

UI distribution. FLUID provides an intuitive interface for selective UI distribution based on multi-finger tapping gestures[2]. On the host device, upon user request, FLUID first partitions the host app's UI tree into two parts—the subtree that corresponds to the user-selected UI elements and the rest of the UI tree. The subtree consists of UI layout and widget objects as well as their UI resources. Once partitioned, FLUID sends the subtree to each guest device. On a receiving guest device, FLUID executes a generic *wrapper* app to reconstruct the received UI subtree and associated UI resources, and displays the corresponding UI elements via local rendering (which we call *guest* UI elements).

UI interaction. Upon displaying the guest UIs on a guest surface, FLUID allows the user to interact with the host app through host and guest UIs simultaneously across devices in the same manner as if all the UIs were on the same device. For instance, the user can swipe a video progress bar on a guest surface to find which part of a video clip to display on the host. To this end, the host app logic and guest UI objects should interact with each other across devices. FLUID supports such interaction, transforming local function calls to cross-device function calls whenever necessary.

4.2 System Design

In order to enable our multi-surface execution model, FLUID addresses the following challenges:

C1. How to partition and distribute UI objects while fully supporting their functionality with minimal overhead?
C2. How to manage cooperation between UI objects and app logic transparently when they are distributed across different devices?
C3. How to preserve app consistency when UI objects are replicated across multiple devices?

C1. A key principle of FLUID is to partition and distribute UI objects while minimizing cross-device communication as illustrated in Figure 4. Based on our observation that rendering occurs frequently (e.g., 20-30 FPS) and its result has a large amount of data, FLUID aims to achieve local rendering of guest UI elements while eliminating any communication in the rendering process. This is particularly beneficial in mobile environments, since mobile wireless networks are often subject to low bandwidth, high latency, and unstable connectivity. FLUID not only saves the network bandwidth but also significantly reduces the number of network round-trips, leading to better responsiveness compared to screen

[2]Our demo video demonstrates how a user initiates UI distribution.

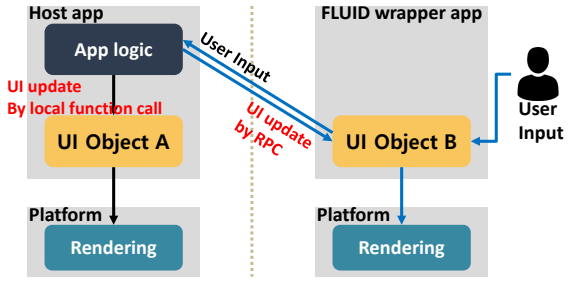

Figure 4: UI partition & distribution

sharing that performs rendering only on the host device (as discussed in Section 9.3).

To this end, FLUID aims to determine a minimal-yet-complete set of UI objects and UI resources required to render the selected UI elements on the guest device and only migrate those to the guest. In the case of stock Android UIs, such a set can be easily predefined because their related UI resources are known in advance. However, the same approach cannot be applied to the case of app-specific custom UIs since it is not publicly known which UI resources they use. Thus, FLUID leverages static code analysis and runtime object tracking to determine such a set precisely without any false negatives (see Section 5). This way, FLUID supports custom UIs while the state-of-the-art studies on multi-device UI distribution [53, 56] may fail to re-create the same look-and-feel of custom UIs.

C2. FLUID aims to support the programming abstraction of a single device for multi-surface operations. When a user interacts with an application across multiple surfaces, guest UI objects and the host app logic need to cooperate with each other. Specifically, upon user input, the guest UI objects should update UI states according to the host app logic and/or trigger the execution of some host app functionality. For example, when a video is paused and a user taps on a video display window, the corresponding UI object triggers a touch event listener that displays a video clip and causes a progress bar to proceed in synchronization. FLUID supports such cooperation, which is originally defined in terms of local method calls within the same address space. FLUID transforms them into cross-device RPCs transparently, addressing the following issues: 1) function call interception and forwarding, and 2) seamless cross-device RPC execution (see Section 6). This way, FLUID supports existing legacy applications without requiring any modification to their code, maximizing applicability.

C3. In order to provide greater flexibility, FLUID gives users an option to replicate a UI element on multiple surfaces such that they can choose to use a single or multiple devices when interacting with the UI element. For example, when watching a video clip on a smart TV, a user can choose to replicate the video progress bar on her smartphone as well as on her smart TV in order to control it from both or either of the devices. FLUID ensures that it *deterministically* applies all updates to replicated UI states across all devices so that the UI states are in sync. FLUID does this by making sure that on all devices, it first *triggers* all updates in the exact same way, and then *executes* them in the exact same way as well. Detailed are described in Section 7.

5 SELECTIVE UI DISTRIBUTION

This section describes what FLUID does in migrating UI elements. For responsiveness, FLUID seeks to deploy only a minimal and complete set of UI objects and UI resources required for rendering. To this end, it employs static code analysis and runtime object tracking to find it, and performs cross-device UI re-creation.

5.1 Static Code Analysis

Before UI distribution, FLUID performs static code analysis for target apps to identify a set of candidate UI objects and UI resources that the renderer thread may use. For each UI object, the renderer accesses the UI object's UI resources when executing the following rendering functions: *i*) `measure()` to calculate the size of each UI element to display, *ii*) `layout()` to calculate the position of each UI, and *iii*) `draw()` to draw the UI elements.

Since UI resources are the objects accessed during rendering, we identify them by leveraging a static analysis technique called the Class Hierarchy Analysis (CHA) using *Soot* [24], an open-source static analysis tool. CHA produces an exhaustive call graph, where each call site points to every possible class method that can be invoked. CHA does this by analyzing all possible class types that each call site object can have (i.e., each call site object's declared type and all children types). However, Soot's implementation of CHA is not directly applicable to finding UI resources, since *i*) it is designed for regular programs with `main()` (which Android apps do not have), and *ii*) it does not analyze a specific part of a program (e.g., UIs), but rather a whole program.

Thus, FLUID adopts CHA in the following three ways. First, FLUID synthesizes a dummy program that has a `main()` function in it. Second, while synthesizing the dummy program, FLUID copies all classes from a target app (i.e., an APK file) as well as the platform library (since the platform library defines default UI classes). Third, FLUID puts an invocation for each of the rendering functions of the `View` class (the root UI class in Android) within the dummy program's `main()`. This allows Soot to perform correct CHA starting from `main()` of the dummy program since every UI class is derived from `View`. This way, we generate an exhaustive call graph that contains all potentially-reachable UI resources. From the call graph, we extract a list that includes all the

UI classes that the target app has and all classes (i.e., UI resources) which are potentially accessible when each UI class is rendered. This final list has no false negative, although it may have false positives (which do not affect correctness).

The analysis result is deterministic on the same app unless it is updated. Therefore, we envision a model where a server (e.g., an app market server or a FLUID service server) performs the analysis and attaches the result into an APK file of each app, whenever a new version is released. We observe that the sizes of our analysis result range from 163 to 236 Kbytes with 20 existing apps (described in Section 9.1), i.e., 0.3% to 13.2% increases from their original APK file size. Thus, we conclude that it is negligible considering the amount of storage equipped in recent devices. In addition, we observe that the execution of our analysis tool takes from 1m 59s to 3m 30s for the existing apps using a desktop machine with 8 cores and 64 Gbytes RAM.

5.2 Runtime Object Tracking

FLUID seeks to reduce the false positives included in the static analysis results through runtime object tracking. To this end, FLUID uses a serialization library called *Kryo* [49]. However, we modify it to suit our purposes. The reason is because, upon serializing an object (a target), Kryo explores all reachable objects from the target dynamically and serializes all of them together. As a result, it may serialize unnecessary objects (i.e., non-graphical states) when used in FLUID. Thus, we modify Kryo to serialize a minimal set of objects, i.e., the UI objects and resources that FLUID needs to deploy on a guest device. Our modified Kryo receives the UI objects that a user chooses as input and tracks the UI resources that exist as member fields of each UI object by using the analysis result mentioned in Section 5.1. It effectively serializes only the intersection between the set of reachable objects that our static analysis has produced and the set of reachable objects that Kryo explores at run time.

5.3 Cross-device UI Re-creation

Upon receiving serialized UI resources from a host device, the FLUID *wrapper* app running on a guest device reconstructs the received UI subtree and associated UI resources through its own UI thread, and displays the guest UI elements via local rendering. To do so, it uses Android's standard APIs for programmatically creating UI elements. We note that re-creating UI objects needs app code and other resource files (e.g., images, fonts, etc.) for custom UIs. For this, the wrapper app dynamically loads them from the APK file delivered at pairing time.

6 TRANSPARENT RPC SUPPORT

Once we deploy guest UIs on guest devices, the host app logic and guest UIs need to interact with each other. FLUID aims to support the programming abstraction of a single device for multi-surface operations, maximizing applicability. Thus, FLUID transparently extends inter-object function calls within the same address space to cross-device RPCs as shown in Figure 5(a). This section describes how FLUID supports transparent function call interception and seamless RPC execution.

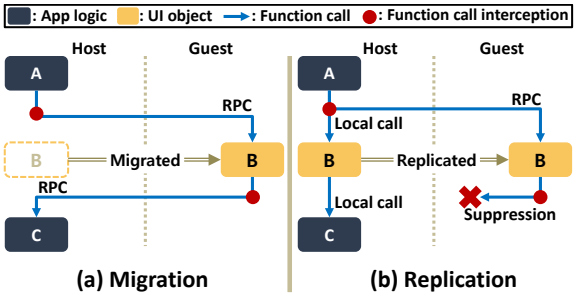

Figure 5: Transparent RPC support with migrated and replicated UI objects

6.1 Transparent Function Call Interception

Normally, a function call is made by storing its arguments and a return address in registers or the stack, and jumping to the address of the function entry point. On virtual machine (VM)-based systems such as Android, the VM stores and manages the entry point of each function. Thus, FLUID modifies the Android VM (ART) and intercepts function calls (see Figure 5(a)). FLUID replaces the entry point address of a target function with the address of FLUID's *code gadget* which transparently converts local function calls into RPCs (more details in Section 8). Upon intercepting a function call, FLUID determines if it should be handled as an RPC call by checking which device the corresponding UI resides in. If the UI is on the guest side, FLUID creates an RPC message along with the arguments of the target function from the registers and the stack, and transmits it to a guest (callee) device. The FLUID code gadget on the host (caller) device then jumps to the return address of the target function upon receiving a return value or an error code from the callee device. However, if the UI only exists on the host side, FLUID just jumps to the original code of the function instead of generating an RPC message. Such interception can cause unnecessary performance overhead, but we show that it is negligible in Section 9.2.

6.2 Seamless RPC Execution

Even after intercepting local calls and transforming them into RPCs, care must be taken to correctly execute RPCs. Broadly, there are two categories of problems that we address for correctness. The discussion here assumes that a host is making an RPC call to a guest, but it is equally applicable the other way round. First, a problem occurs when a

target function has reference type arguments, such as uniform resource identifiers (URIs), since such references are only valid on the host device. To resolve this, FLUID on the host checks whether each argument is of a reference type or a value type. If it is the former, FLUID copies the referenced resource object to the guest device and allocates it properly with a new reference. Then, FLUID on the guest replaces the reference-type argument with the new reference such that the target function can access the allocated resource correctly on the guest device.

Second, when executing a target function, it may access objects that do not reside on the guest device (i.e., non-graphical objects). In order to enable such accesses, FLUID employs *virtual* objects, which are proxy objects for the real objects that exist on the host device. When the target function accesses a virtual object, FLUID forwards it to the host device through an RPC (see Figure 5(a)).

7 CROSS-DEVICE UI REPLICATION

In order to maximize flexibility, FLUID gives users an option to replicate a UI element on multiple surfaces and use it on all or some of the surfaces. For example, multiple users can share the same UI element on their smartphones to carry out collaborative tasks, such as playing a hidden picture puzzle together or filling out forms jointly (e.g., putting passport numbers for a group flight reservation). In this section, we explain how FLUID enables this UI replication.

7.1 UI Replication Overview

The basic mechanism for UI replication is the same as the UI distribution and RPC mechanisms described in Sections 5 and 6. This means that i) FLUID still deploys the UI objects and UI resources necessary for rendering on guest devices, and ii) FLUID still uses RPCs to transform local method calls into remote method calls whenever graphical states and non-graphical states need to interact across devices. The difference is that UI replication now displays and manages *replicated* UI elements as illustrated in Figure 5(b). In other words, all devices (host and guest devices) now have a copy of each (replicated) UI element and its graphical states, and synchronize them across all devices. While doing so, FLUID also allows a user to interact with the replicated UI elements on all or any of the surfaces.

To enable this, FLUID implements *dual execution*, where FLUID ensures that updates to replicated UI states occur *deterministically* on every device (see Figure 5(b)). A state update to a UI object occurs through the execution of a method of the UI object; with dual execution, FLUID invokes a UI object method not only on the local device that a user interacts with, but also on all other devices that have replicas of the UI object. Our dual execution requires us to address two additional challenges, UI state synchronization and duplicate execution suppression, which we discuss in the rest of this section.

7.2 UI Synchronization and Its Guarantee

In order to synchronize the states of replicated UI objects and UI resource objects, FLUID makes deterministic state updates for replicated objects. In other words, FLUID enforces that the replicated executions at different devices are identical so that they can produce the same UI states. In general, enforcing deterministic execution requires identifying the sources of non-determinism and enforcing determinism on them. The sources of non-determinism in a mobile system include thread scheduling, user input, sensor input, network input, file reads, inter-process communication, hardware specification, random numbers, clock readings, etc. A replication system that enforces determinism must implement a mechanism to make those non-deterministic events deterministic across multiple devices. For example, a previous replication system for mobile devices (Tango [22]) logs all non-deterministic events from a "leader" device and forwards them to a "follower" device. Unlike Tango that provides full replication, FLUID focuses on UI replication and employs a customized design suitable for UI replication.

FLUID's replication design leverages the following two observations regarding Android's UI execution model. First, Android's UI system has a single UI thread, i.e., there is only one thread that updates the states of all UI objects. Second, there is a single input event queue that drives the execution of a UI thread. In other words, the execution of a UI thread is the one that dequeues an input event from the input queue and executes an event handler associated with the input event. This execution model is not specific to Android; many UI systems, for example, Swing [42], Qt [11], SWT [14], and Cocoa [3], follow this execution model since it avoids race conditions when updating UI states [55].

Based on these observations, our UI replication enforces deterministic updates of replicated UI states. More specifically, we use two techniques to deterministically apply all updates to replicated objects (UI objects and UI resources) in the exact same way. The techniques we use are *deterministic triggering* of UI state updates and *deterministic execution* of UI state updates. The combination of these two techniques guarantees the overall UI state synchronization since every aspect of a UI state update (triggering the update and executing it) becomes deterministic.

Deterministic triggering of UI state updates. In order to enforce deterministic triggering of UI state updates, we ensure that the host device imposes a total ordering of all UI updates, and the guest devices simply follow the ordering from the host. This would be easy if all UI updates were triggered by the events from the host (e.g., user input on

the host device), since we would just need to make RPCs to guest devices whenever there is a UI update to trigger the UI update on the guest devices. However, our goal for UI replication is to allow users to interact with replicated UI elements across multiple devices, and user input on a guest device can potentially change the state of a UI object. Thus, we forward all user input events from guest devices to the host device, and the host device enqueues the forwarded events to its input event queue to process them. When there is a UI update, the host device not only executes it locally but also makes an RPC to each of the guest devices to trigger the execution of the UI update on the guests. These local and RPC calls occur asynchronously from each other in order to preserve user interactivity. In addition, we enforce that the UI thread running on a guest device (as part of the container app) does not update replicated objects directly in any way. This means that the only way to update the state of a replicated UI object on a guest device is through RPCs from the host.

In summary, FLUID's UI replication only processes two types of input events—(i) all events in the host's input event queue (e.g., user input on the host), and (ii) user input events from guests. All other events from guest devices are excluded from processing, since our goal is enabling UI interactions on multiple surfaces. Since the host receives all user input events from guest devices, its input event queue naturally imposes a total ordering of all the events that FLUID processes. This mechanism ensures that we trigger UI state updates deterministically.

Deterministic execution of UI state updates. In order to execute UI state updates deterministically, FLUID always relies on the host to supply non-deterministic values while executing a UI state update. For example, consider a method call on a button UI object that updates the state. The execution of the method call might use a hardware-dependant value, such as IMEI. Since different phones will have different IMEI values, we cannot guarantee determinism if we use a local IMEI value while executing the method. Thus, instead of moving objects with non-deterministic values (e.g., hardware information, clock readings, random numbers, file reads, etc.), FLUID creates virtual objects corresponding to them, and always receives such values from the host via RPCs. For instance, since most hardware information is accessed by binder objects connected to external system services, our basic mechanism can naturally handle it by simply replacing binder objects with virtual objects.

7.3 Duplicate Execution Suppression

As discussed, our UI replication makes RPCs to execute non-replicated parts of an app. At the same time, its original UI residing in the host device also executes the non-replicated parts as well. Unfortunately, this has an undesirable consequence of *duplicate execution*, where a non-replicated method is invoked multiple times due to replicated executions on different devices. For example, suppose a user replicates a video playback control on a guest device from the host. When she clicks on a play button on either the host or the guest surface, FLUID arranges each device to update the graphical state of the play button through dual execution. Each device then makes an individual RPC to the host app logic (which is a non-replicated part) to start the video. However, this will result in two calls and the host app logic will end up toggling the video status twice, which will pause the video instead of playing it. To address this problem, FLUID caches the result of a method execution and provides the cached result to an RPC for the method (see Figure 5(b)).

8 IMPLEMENTATION

To build a FLUID prototype, we have mainly modified Kryo to serialize UI objects and UI resources (1,111 LoC), Android Java framework to manage distributed UI subtrees and event queues of UI threads across devices (1,766 LoC), and Android VM, ART (Android Runtime), with a code gadget that consists of ARM assembly code and C++ native code for control flow hijacking (1,874 LoC). We have also added the FLUID system service to support device pairing (1,566 LoC). Due to the space limit, this section only describes implementation details for control flow hijacking, which is one of the key techniques to support transparent cross-device RPC and seamless dual execution.

Function call interception. We have modified the class loader of Android ART to implement the transparent function call interception described in Section 6. When loading Java classes into memory, the ART class loader creates an `ArtMethod` object, which is the metadata of the functions of each class, and sets the entry point of each function. At this point, we set the entry point of the function to the address of FLUID code gadget.

Function return interception. To record the return value of a function as described in Section 7.3, we use the LR (Link Register) register to access and store return addresses. Specifically, when intercepting a call to the target function, FLUID changes the LR register value to the address of FLUID code gadget and jumps to the original entry point of the target function. This way, upon completing the execution of the target function, the control flow returns to the FLUID code gadget again, and FLUID records its return value.

Conflict with garbage collector. One of the problems that we have faced with FLUID code gadget is a conflict with the GC. When intercepting a function call, the assembly code of FLUID first saves its caller's context (e.g., registers) on the stack and then performs computation to make several decisions, such as deciding where to forward the call and

Type	User case scenario	Custom UI type	App name (Downloads)	FLUID				App migration size (Kbytes)
				Network round-trip	Arg. (Bytes)	Ret. (Bytes)	UI distribution size (KBytes)	
Usability	Edit text on different device	Edit Text	Color note (100M)	1	0	0	8.7	16,500
			Text editor (1M)	1	587	0	35.8	46,100
	Control media with different device	SeekBar, Button	VLC Player (100M)	1	14	0	998.5	38,000
			Music Player (0.5M)	1	128	0	358.4	18,700
	Control painting tool with different device	Scroll, Button, Image	PaperDraw (10M)	1	666	0	2,026.1	21,300
			Paint (1M)	1	1,602	0	272.5	63,400
	Chatting with different device while broadcasting	Edit Text, Button	LiveMe (50M)	2	72	1	45.3	85,600
			Afreeca TV (10M)	5	8	1	234.3	43,000
	Search destination with different device	Edit Text, Button	Naver map (10M)	1	67	1	37.9	199,000
			Maps.me (50M)	1	0	1	126.1	94,500
	Read document with different device	Text, Scroll	File Viewer (1M)	0	0	0	8.9	11,700
			Bible KJV (10M)	0	0	0	20.7	97,200
Privacy	Login with personal device	Edit Text	Instagram (1B)	1	5	1	12.5	45,900
			PayPal (50M)	1	94	0	19.6	54,000
	Unlock pattern with personal device	Pattern	Smart AppLock (10M)	1	8	0	3.6	29,600
			AppLock (10M)	1	8	0	106.3	67,500
	Sharing photo to public device	Image	Gallery (10M)	0	0	0	182.9	51,200
			A+ Gallery (10M)	0	0	0	362.8	66,300
Collabo. Use	Fill in information collaboratively	Text, Edit Text	eBay (100M)	1	55	1	62.9	73,500
			Booking.com (100M)	1	67	1	97.7	93,000

Table 1: Use case list for coverage test. 'Custom UI type' is a super class of distributed UI. 'Arg.' and 'Ret.' indicate the maximum amounts of data transferred for the arguments and return values of RPCs, respectively.

whether the call is being made to a replicated portion. Unlike Dalvik VM that maintains two separate stacks for each thread, one for Java code and the other for native code, ART uses a single, unified stack for both Java and native code. In ART, when the GC walks through the stack in order to reach Java objects in the heap, problems occur since it is not able to parse the stack frame that FLUID directly manages to store the context information. Thus, we have modified the ART garbage collector such that it skips the stack frames associated with FLUID native code and walks through Java stack frames properly without crashing.

9 EVALUATION

We have implemented a FLUID prototype to demonstrate its complete operation of selective UI distribution across multiple surfaces for unmodified applications, maintaining the same look-and-feel successfully for not only stock Android UIs but also custom UIs.

The FLUID implementation used for our evaluation is based on Android Open Source Project (AOSP) v.8.1.0 (Oreo). We have used a Google Pixel XL (smartphone) and a Pixel C (tablet) across the evaluation. During the evaluation, we have enabled all cores to run at the maximum CPU frequency (2x2.15GHz & 2x1.6GHz), and connected all devices to the same Wi-Fi access point with a throughput of 45 Mbps, and a round-trip time (RTT) with the median, average, and standard deviation of 32.1, 42.9, 40.31 ms, respectively.

9.1 Coverage Test

In order to explore how well FLUID supports possible use cases and existing applications, we have evaluated 10 use case scenarios described in Section 2. For each use case scenario, we have used two legacy applications from Google Play. Table 1 shows the list of use cases and apps we have evaluated. We have confirmed that all 20 legacy apps use custom UIs, and FLUID successfully enables them to operate over multiple surfaces of heterogeneous devices: phone-to-phone, phone-to-tablet, and tablet-to-phone.

The 'network round-trip' column in Table 1 shows the maximum number of communication between the host and guest devices during the execution of a single method on the guest device in each scenario. It is noteworthy that the number ranges only from 0 to 5, mostly 1; no communication occurred for the case where no input was given (e.g., File Viewer, Gallery). This shows that our selective UI distribution model (described in Section 5) have successfully partitioned a complete set of UI objects and UI resources, thereby minimizing the number of network round-trips.

The 'UI distribution size' column represents the amount of data that FLUID has transferred in the process of UI migration. The 'app migration size' column represents the amount

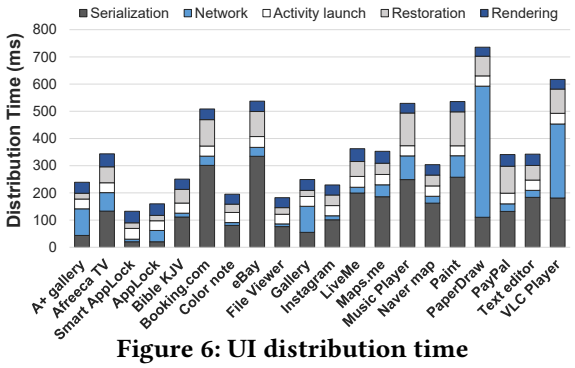

Figure 6: UI distribution time

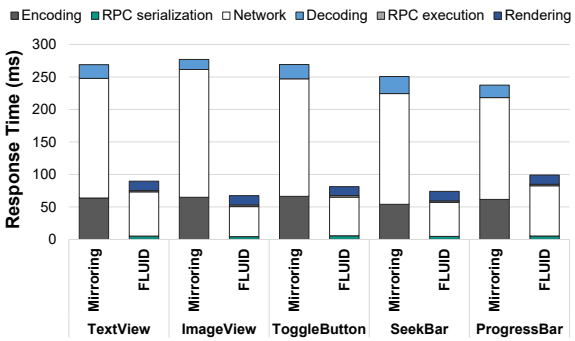

Figure 7: UI response time

of data in memory (i.e., code, stack and heap) that each app allocated right after we launched it (i.e., when its memory usage is at the minimum), for which we use Android Profiler tool [20] to measure. Since an app migration approach (e.g., Flux [52]) transfers its entire memory usage, this measurement can serve as a lower bound on the amount of data to transfer if we were to migrate an entire app, which would be incomparably larger than the UI distribution size. This indicates that FLUID has transferred a very small portion of the app's entire data space for UI distribution, demonstrating that FLUID has successfully identified not only complete, but also a minimal set of UI resources.

9.2 Performance Test

We quantitatively evaluate the performance of FLUID for its multi-surface operations. For each experiment, we have repeated ten times with three different device setups: phone-to-phone, phone-to-tablet, and tablet-to-phone. However, we only show phone-to-phone results for brevity because the results of the other device setups show a similar tendency[3].

UI distribuion time. In order to evaluate the performance of FLUID in deploying UI elements on a guest device, we define *UI distribution time* as the time difference between the user input that triggers the UI distribution and the final screen update on the guest device, which is inclusive of the time required for network transfer and rendering. Figure 6 breaks down the UI distribution time measured for 20 legacy apps listed in Table 1. It ranges from 132 to 735 ms depending on which UI type to deploy, since different UI elements are subject to different sets of UI objects and UI resources for rendering. It shows that FLUID distributes UI elements on a different device fast enough for interactive use. It shows that FLUID distributes UI elements on a different device fast enough for interactive use.

It is worth noting that in many of the cases, serialization has large overhead. As mentioned earlier, we have modified a serialization library called Kryo [49] in our current prototype. Even though our modification reduces the amount of serialized objects, the overhead of serialization remains high. However, we have noticed that there are performance optimization opportunities in Kryo, e.g., avoiding deep and frequent recursions that the current Kryo implementation has. We expect that we can bring down the performance overhead by leveraging those opportunities. We also expect that using a different serialization library that is more optimized for performance can reduce the overhead.

UI response time. For UIs deployed on a guest surface, we measure what we call *UI response time*. It is the delay between when a user gives a touch input on the guest surface and when the guest surface completes displaying the result of the touch input. Figure 7 shows the average UI response times in updating five most popular UI widgets with FLUID, compared to an open-source screen mirroring program, SCRCPY [15]. We note that a UI response time in FLUID depends on the size of the arguments passed to an RPC, and we set it to the maximum argument size of 2,000 bytes. 2,000 bytes is larger than the largest RPC argument size we have observed (1,602 bytes) while profiling 20 legacy apps, as shown in Table 1. The figure shows FLUID outperforms the screen mirroring approach by 2x to 4x.

RPC performance overhead. We have measured the overhead that FLUID imposes when supporting RPC (as discussed in Section 6.1) by comparing the function execution times for three cases: *i)* without any interception (i.e., on stock Android), *ii)* with interceptions but no RPC, and *iii)* with both interceptions and RPCs. We have used a custom app that repeatedly calls a simple function `TextView.setText()` 100 times. Our results show that the three cases respectively took 215 us, 221 us, and 2,811 us on average (with the standard deviations of 63 us, 61 us, and 876 us, respectively) to run the simple function. It means that the interception overhead of FLUID is 6 us on average, which is negligible. On the other hand, the third case has relatively large latency due to the RPC overhead. It is an inevitable cost for cross-device communication, but we expect it can be mitigated by using advanced wireless technologies such as 5G [38] or 802.11ad [39].

[3]Remaining results are provided in our supplement file [40].

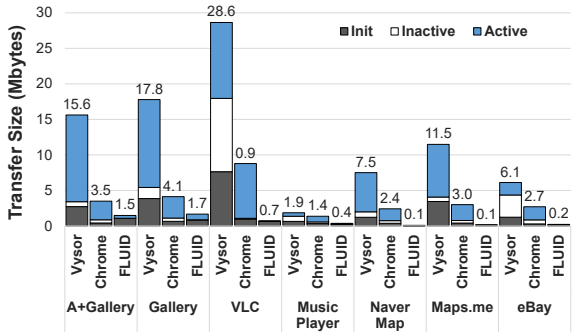

Figure 8: Data usage comparison

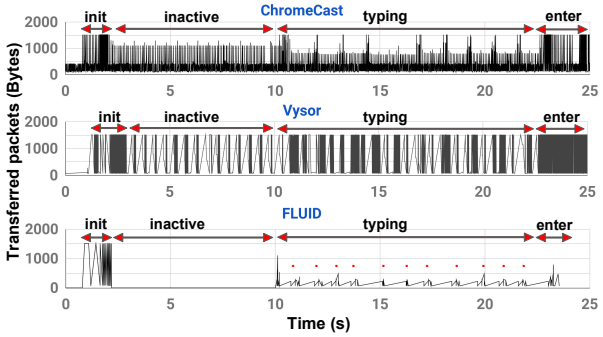

Figure 9: Byte transfer over time

9.3 Network Optimization

Figure 7 indicates that network transfer time dominates the UI response time for both FLUID and SCRCPY. In order to further investigate their network usage, we have designed an experiment where we measure the amount of network traffic in the following three stages: i) init: initialization for user interaction, i.e., launching an app's activity for screen sharing and UI re-creation on a guest device for FLUID, ii) inactive: an idle period until the 10-second mark where we do not perform any interaction, and iii) active: a period of touch events starting from the 10-second mark. By this experiment, we compare the network usage of FLUID and screen mirroring, which can be used for several multi-surface scenarios. While screen mirroring has to copy the entire screen between devices, FLUID allows users to selectively deploy only the desired UIs to other devices to support the same use cases. The approaches result in different resource usage, in particular, as to network traffic amounts which affect response time and energy consumption.

We have performed this experiment for 7 legacy applications under their respective UI scenarios shown in Table 1 using FLUID as well as two of the most popular screen sharing applications, Chromecast and Vysor. Figure 8 shows the total amount of data transferred across 7 legacy applications. We observe that FLUID incurs an order of magnitude smaller amount of transferred data than the other two approaches in all cases. We also note that Vysor supports mirroring in the full resolution of the host device at the cost of transferring a large amount of data, while Chromecast provides a lower resolution to reduce the amount of data transferred. It is interesting to see that FLUID provides the same resolution as that of Vysor with a much smaller network cost, even smaller than Chromecast providing a lower resolution.

We further investigate how different approaches behave by looking into the case of Naver Maps. The scenario that we have evaluated is deploying a text input box on a guest device, tapping the text input box to set the focus, and typing 11 characters and the enter key in the text input box as the location to search. Figure 9 plots the amount of transferred data over time, with the three stages of *init*, *inactive*, and *active*. In the figure, high density areas represent very frequent network transmissions, and Vysor shows different densities over time, representing that it employs an adaptive screen update frequency. In the active stage, Vysor shows denser lines than in the inactive stage, indicating that it sends screen updates more frequently when user input is given. In any case, it has transferred a large amount of data. Chromecast has similar densities periodically, indicating that it has transferred frames at a fixed frequency regardless of user input. It has transferred a smaller amount of data than Vysor, most likely due to its lower resolution. On the other hand, FLUID shows a completely different network usage, and has transferred close to zero amount of data in the inactive stage and a much smaller amount of data (roughly 209 bytes on average) at a remarkably lower frequency in the active stage. This is because FLUID has transferred only the data (e.g., RPC arguments and return values) necessary to support cross-device inter-object cooperation through RPCs. It is interesting to see that there are exactly 11 peaks (marked with red dots) in between setting focus on the input box and pressing the enter key, which correspond to the 11 characters typed. We observe that FLUID optimizes network performance by minimizing both the amount of data transferred and the number of network round-trips.

9.4 Power Consumption

To evaluate the power consumption of FLUID, we have conducted an experiment on 3 legacy apps under the same scenario used for Figure 8 using Monsoon Power Monitor [50]. For each application, we have measured the average power consumption of the host and guest devices of FLUID, the host devices of Chromecast and Vysor, and a single device system. Figure 10 shows that for both host and guest devices, FLUID uses power comparable to the single device system. Vysor and Chromecast, on the other hand, shows noticeable overhead compared to FLUID.

To examine the source of power consumption, we have plotted the power consumption of Naver Maps over time.

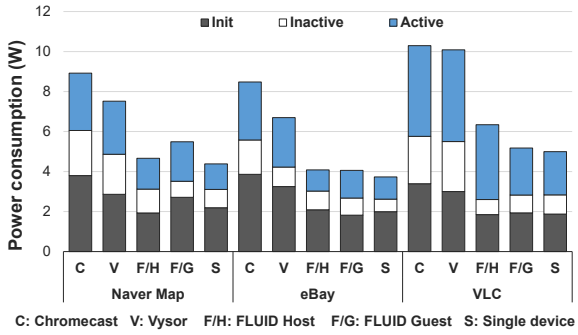

Figure 10: Power consumption comparison

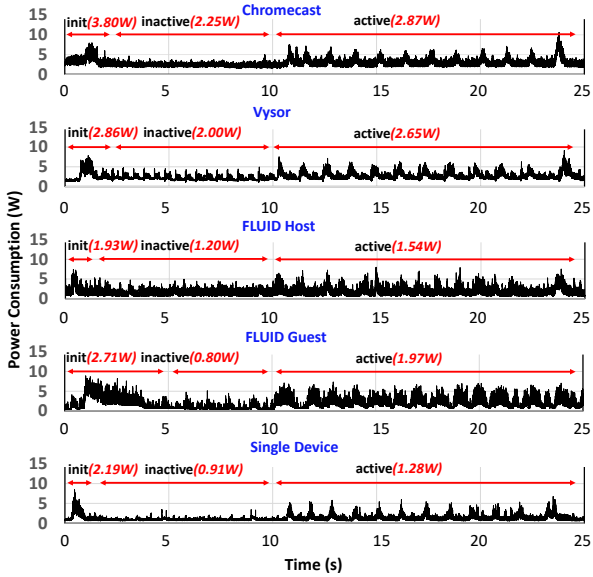

Figure 11: Power consumption over time

As Figure 11 demonstrates, power consumption follows the same trend as byte transfer over time (Figure 9), which indicates that the source of FLUID, Vysor, and Chromecast's power consumption overhead is mostly from network usage.

We note that we have not explicitly addressed power optimization in this paper. Instead, we mostly focused on optimizing the performance by reducing network latency, as evaluated in Section 9.3. It is interesting to see that, since network usage dominates the power consumption overhead, efficient use of wireless network leads to reduced energy consumption as well.

9.5 Compatibility

The core FLUID design principles, such as object serialization and cross-device RPC, are dependent on the Android's class definitions. However, since such definitions are different according to the Android versions, FLUID may not work properly between the devices equipped with different Android versions. To explore the variance of class definition,

	v.6.0 to v.8.1	v.8.1 to v.6.0	v.7.0 to v.8.1	v.8.1 to v.7.0
Same Field	96%	87%	98%	94%
Same Method	90%	82%	96%	92%

Table 2: Compatibility across Android versions

we have measured the differences of field and method definitions between *Oreo* (v.8.1), *Nougat* (v.7.0) and *Marshmallow* (v.6.0) for common classes. As Table 2 shows, the majority of field and method definitions are identical across the versions. From our observation that most Android apps have the same or similar look-and-feel regardless of Android versions, we can deduce that most Android UI objects share common methods and fields as a base.

To test our hypothesis, we have implemented a simple translation layer prototype inside the serialization and RPC management to translate four UI objects: `TextView`, `EditText`, `Button`, and a custom UI (`SeekBar`) from Oreo to Marshmallow. The prototype serializes the common fields, along with dummy data for other fields, and translates common methods of the four UI objects (e.g., *setText()*), according to the definition of Marshmallow. We have confirmed that all four UIs and their common methods work seamlessly and maintain the same look-and-feel across Oreo and Marshmallow. We leave the automation of this translation layer to be a topic for future work.

10 RELATED WORK

App-level multi-surface support. Many custom applications are available for multi-surface operations, including Google Docs [19], Netflix [35], and YouTube [21]. They allow multiple individual copies installed on different devices to share the same app contents through cloud servers, enabling a user to continue tasks across multiple devices and/or to perform collaborative tasks with others simultaneously. However, this approach requires a significant amount of engineering effort to develop custom apps, and its applicability is very limited in that only custom apps can support certain multi-surface operations. On the other hand, as a system solution, FLUID is able to support a wider range of unmodified applications while incurring no additional development cost.

New programming models/tools. There are several studies that propose new programming models or development tools for multi-surface app development. UIWear [53] automatically generates an app for wearable devices by extracting UIs from a smartphone app. CollaDroid [56] introduces a code instrumentation tool that transforms an app such that it can distribute UIs to other devices. They can reduce development costs by automatically creating multi-surface apps without requiring app source code (i.e., Java code). However, they can only maintain the same look-and-feel for stock Android UIs, and they cannot do so for app-specific custom UIs. This is because they do not consider how

to extract the app-specific graphical states related to custom UIs. On the other hand, FLUID performs code analysis on Android's platform code as well as app code to figure out the graphical states of custom UIs without false negatives and provide greater applicability. In addition, there are many tools [6, 7, 16, 26, 36] that allow app developers to co-design UIs for multiple devices. Although these tools help easing the process of developing multi-device apps, it has the initial learning curve for each tool and still requires efforts at development time. There are also other studies [25, 33, 34, 43, 54] on development frameworks that distribute the UIs of web applications, but they cannot be applied to mobile apps.

Screen sharing. There are a number of screen sharing (or screen mirroring) applications that transfer screens from one device to another, such as Vysor [10], Scrcpy [15], Teamviewer [51], and Chromecast [18]. However, since this approach basically duplicates screens at a coarse-grained level (i.e., at the level of the whole device screen or the whole app screen), its use cases are limited to take full advantage of multiple surfaces from the user's viewpoint. For instance, it does not support one of FLUID's use cases where a single app deploys different UIs on different surfaces. Exceptionally, Chromecast allows an app to transfer only partial UIs through customization; that is, it is applicable only to customized apps (developed to use Chromecast), but not to other unmodified apps.

App/thread migration. Flux [52] supports app migration such that an app migrates to another device in the middle of execution and use the surface of the guest device. However, since Flux allows the app to only use one surface among multiple at a time, the app cannot distribute different UIs to different surfaces and use them concurrently, which is possible with FLUID. A great deal of work has been done to support thread migration, such as MAUI [12], CloneCloud [8], and COMET [23]. While aiming at achieving performance gains by offloading compute-intensive workloads to servers with higher computing power, they typically do not consider the optimization issues involved in offloading UI workloads for interactive use, which is one of the focuses of this paper.

Cross-device RPC. RPC has been used in many distributed systems for decades with the support of API libraries such as Java RMI [44], Microsoft .NET Remoting [30], and JSON-RPC [32]. Mobile Plus [41] may be the closest work to ours in terms of extending within-device function calls to cross-device. However, there are a significant difference that Mobile Plus extends *inter-process* method calls to cross-device RPC while FLUID extends *intra-process* method calls within the same address space and supports dual execution. Such a key difference introduces new challenges that were not considered by Mobile Plus, including how to intercept local method calls transparently and provide correctness for replicated RPC calls.

Cross-device I/O sharing. There are a few studies that propose multi-device systems for I/O sharing. Rio [2] supports sharing of I/O resources across devices via virtualization at the device file layer. M2 [1] presents a data-centric solution that utilizes high-level device data to support I/O sharing between heterogeneous devices. MobiUS [45] enables display sharing between devices to play a video with high resolution. However, these systems do not allow apps to utilize different device screens at a fine-grained level (i.e., at the level of UI elements), which is possible with FLUID.

11 DISCUSSION

Direct memory writes to UI objects. The current prototype of FLUID does not support direct memory writes to UI objects and resources, i.e., direct writes to their public fields. However, such cases rarely happen since most app developers follow the principle of encapsulating the graphical states of UI objects and updating them only through method calls, instead of direct writes, to avoid race conditions [31]. We can still extend FLUID to provide support via field-level distributed shared memory (DSM) (e.g., COMET [23]), just for public fields. Alternatively, FLUID can leverage write barriers used in a GC, where every write is checked for access. By leveraging this, FLUID can transparently intercept every write as we do for method calls.

Native object serialization. Unless a developer uses a serialization library, C++ by itself does not have any support for serialization. This is because it does not provide runtime type information as Java does through reflection. Thus, the current FLUID prototype adds serialization for only the native objects of Android C++ graphics libraries (e.g., Skia [47]) but not for custom graphics libraries. We have confirmed that the current prototype still supports the 20 different legacy applications listed in Table 1, since they all use custom Java objects but not any custom native objects. Furthermore, we have studied 43 open-source apps which received 4,505 stars scores on average on GitHub [17] and checked only one of them employs custom C++ graphics libraries. It means most apps tend to use the Android C++ graphics libraries. If the C++ compiler provides complete runtime type information in the future, FLUID can support custom native objects without changing its design.

Multi-surface layouts. During the coverage tests described in Section 9.1, we have noticed a case which could improve usability with slight help from app developers. Specifically, the current prototype of FLUID allows a user to select only visible UI elements for migration or replication, and this may yield a situation different from the user's expectation. As an example, in *Maps.me*, a related query window switches from invisible to visible only after the user starts typing in a text input box. This makes it impossible to migrate both UIs

before the user types in. Yet, such a limitation can be easily overcome if app developers can combine the text input box and its related query window into a single container UI. In addition, FLUID allows the renderer thread on a guest device to render guest UI objects by changing the scale according to the guest surface resolution for better usability. This does not violate our deterministic execution guarantee (in Section 7.2), since it does not modify the graphical states of replicated UI objects. However, if app developers provide layout guidelines, they can precisely control how FLUID displays UIs on different surfaces. For example, if a UI element on the guest should be placed at a relative position of other UIs left on the host, FLUID handles it in a naive manner by just ignoring the configuration and placing the element at an arbitrary position (e.g., the middle of screen). The layout guidelines can better address such issues that may harm app usability due to UI layouts changed by FLUID unintentionally.

Failure handling. Failures may occur due to various causes (e.g., battery depletion, network failure, etc.), and FLUID can handle failures differently for migration and replication. For UI replication, FLUID simply discards failed guests as all updates are replicated. For UI migration, there can be two options. One option is to employ (i) record-and-replay techniques [27, 52] to re-create failed guest UIs on the host surface if a guest fails, and (ii) clean up the remaining UI objects on each guest device if the host fails. The second option is to use UI replication by default even for UI migration, and simply make migrated UIs invisible on the host. This simplifies failure handling. We leave the evaluation of these different options as part of our future work.

Runtime layout change. Upon runtime configurations (e.g., orientation) change, a host app may apply a new UI layout; if this occurs, the app removes all UI objects of the old layout, and creates new UI objects according to the new layout. What this means for guest UIs is that they need to be removed and re-created as well according to the new layout. Although we do not handle such cases in the current FLUID prototype, we can extend it to automatically deploy new UIs upon any configuration change. Since UIs in different layouts are essentially the same ones, their looks are similar; we can leverage this insight and potentially construct mappings between UI objects of different layouts by measuring their similarity. Using this information, if the orientation of the host device is changed from portrait to landscape, we can automatically replace the old UI with the new UI of the landscape layout. We leave it as our future work.

Dynamically loaded UI objects. Even if a UI object is dynamically created, our static analysis can identify its graphical states as long as its definition (bytecode) is statically included in its app or the platform library. However, it is difficult to support UI objects if their code is dynamically loaded, since it cannot be analyzed beforehand. For example, an app may create some UIs using bytecode received from remote servers at runtime. To support such UIs, FLUID can be extended to run the static analysis for the dynamically loaded code blocks on the host device.

Multiple devices with similar I/O spec. FLUID allows users to select a device on which UIs are placed if there are multiple devices with similar I/O peripherals. Also, FLUID can use UI assignment policies as done in previous systems [25, 43] to distribute UIs across devices automatically.

Applying FLUID to other systems. Although FLUID is specialized for Android, its general design is applicable to other mobile platforms. FLUID was designed under the key assumption that UIs are managed by a hierarchical structure and their states can be updated by functions called from a UI thread. This is a common design paradigm of GUI-based systems (e.g., UIView [48] in iOS). To support such mobile platforms, we will face the following three challenges: *i)* Identifying graphical states of each UI object via static analysis. It is difficult to apply CHA to an app compiled down to a binary. However, if app developers perform CHA provided by compilers (e.g., *Swift SIL optimizer* [4]), FLUID can use its result easily. *ii)* Serializing UI objects with graphical states. It is difficult to serialize objects if a system does not provide complete runtime information. However, as mentioned earlier, we can address it by extending compilers to produce such information. *iii)* Transforming function calls into RPCs transparently. If a system is not VM-based, it is possible to use instrumentation and add extra code to generate and transmit RPC messages, instead of call interceptions.

12 CONCLUSION

We have presented FLUID, an Android-based platform that enables innovative uses of unmodified applications on multiple device. FLUID selectively migrates or replicates individual UI elements on multiple surfaces, and transparently manages cross-device cooperation between the distributed UI objects and app logic in a deterministic manner. Our prototype implementation has proved that FLUID achieves highly flexible and transparent cross-device UI executions on a wide range of unmodified, real-world apps, providing high responsiveness. We expect FLUID to accelerate the development of creative and useful applications to provide a variety of novel multi-device user experiences.

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 ERC (NRF-2018R1A5A1059921) funded by the Korea Government (MSIT). Steven Y. Ko was also supported in part by the funding from the National Science Foundation, CNS-1350883 (CAREER) and CNS-1618531.

REFERENCES

[1] 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 (MobiSys '19)*.

[2] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. 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 (MobiSys '14)*.

[3] Apple. 2019. MacOS Cocoa. http://developer.apple.com/technologies/mac/cocoa.html.

[4] Apple. 2019. Welcome to Swift.org. https://swift.org/.

[5] Stuart K. Card, Allen Newell, and Thomas P. Moran. 1983. *The Psychology of Human-Computer Interaction.* L. Erlbaum Associates Inc.

[6] Pei-Yu (Peggy) Chi and Yang Li. 2015. Weave: Scripting Cross-Device Wearable Interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*.

[7] Pei-Yu (Peggy) Chi, Yang Li, and Björn Hartmann. 2016. Enhancing Cross-Device Interaction Scripting with Interactive Illustrations. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*.

[8] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*.

[9] VNI Cisco. 2018. Cisco Visual Networking Index: Forecast and Trends, 2017–2022. *White Paper* (2018).

[10] ClockworkMod. 2019. Vysor. https://www.vysor.io/.

[11] The Qt Company. 2019. The Qt Framework. https://www.qt.io/.

[12] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*.

[13] Expedia. 2019. Expedia. https://www.expedia.com/app.

[14] Eclipse Foundation. 2019. The SWT Toolkit. http://eclipse.org/swt/.

[15] Genymobile. 2019. Scrcpy. https://github.com/Genymobile/scrcpy.

[16] Giuseppe Ghiani, Marco Manca, and Fabio Paternò. 2015. Authoring Context-dependent Cross-device User Interfaces Based on Trigger/Action Rules. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia (MUM '15)*.

[17] GitHub. 2019. The world's leading software development platform - GitHub. https://github.com/.

[18] Google. 2019. Chromecast. https://store.google.com/product/chromecast.

[19] Google. 2019. Google Docs. https://www.google.com/intl/en-GB/docs/about/.

[20] Google. 2019. Measure app performance with Android Profiler. https://developer.android.com/studio/profile/android-profiler.

[21] Google. 2019. YouTube. https://www.youtube.com.

[22] Mark S. Gordon, David Ke Hong, Peter M. Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. 2015. Accelerating Mobile Applications Through Flip-Flop Replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*.

[23] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*.

[24] Sable Research Group. 2019. Soot - A Java optimization framework. https://github.com/Sable/soot.

[25] Tom Horak, Andreas Mathisen, Clemens N. Klokmose, Raimund Dachselt, and Niklas Elmqvist. 2019. Vistribute: Distributing Interactive Visualizations in Dynamic Multi-Device Setups. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*.

[26] Steven Houben and Nicolai Marquardt. 2015. WatchConnect: A Toolkit for Prototyping Smartwatch-Centric Cross-Device Applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*.

[27] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile Yet Lightweight Record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*.

[28] Twitch Interactive. 2019. Twitch. https://www.twitch.tv/.

[29] LiveMe. 2019. LiveMe - Live Broadcasting Community. https://www.liveme.com/.

[30] Scott McLean, Kim Williams, and James Naftel. 2002. *Microsoft .Net Remoting.* Microsoft Press.

[31] Zigurd Mednieks, Laird Dornin, G. Blake Meike, and Masumi Nakamura. 2011. *Programming Android.* Oreilly & Associates Inc.

[32] Matt Morley. 2013. JSON-RPC 2.0 Specification. https://www.jsonrpc.org/specification.

[33] Michael Nebeling and Anind K. Dey. 2016. XDBrowser: User-Defined Cross-Device Web Page Designs. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*.

[34] Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira Norrie. 2014. Interactive Development of Cross-device User Interfaces. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*.

[35] Netflix. 2019. Netflix. https://www.netflix.com.

[36] T. Nguyen, J. Vanderdonckt, and A. Seffah. 2016. Generative Patterns for Designing Multiple User Interfaces. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft '16)*.

[37] Jakob Nielsen. 1994. *Usability Engineering.*

[38] Paul Nikolich, C Lin, Jouni Korhonen, Roger Marks, Blake Tye, Gang Li, Jiqing Ni, and Siming Zhang. 2017. Standards for 5G and beyond: Their use cases and applications. *IEEE 5G Tech Focus* (2017).

[39] Thomas Nitsche, Carlos Cordeiro, Adriana Flores, Edward Knightly, Eldad Perahia, and Joerg Widmer. 2014. IEEE 802.11ad: directional 60 GHz communication for multi-Gigabit-per-second Wi-Fi [Invited Paper]. *IEEE Communications Magazine* (2014).

[40] Sangeun Oh, Ahyeon Kim, Sunjae Lee, Kilho Lee, Dae R. Jeong, Steven Y. Ko, and Insik Shin. 2019. Supplement of "FLUID: Flexible User Interface Distribution for Ubiquitous Multi-device Interaction". http://cps.kaist.ac.kr/fluid/MobiCom19FLUIDSup.pdf.

[41] Sangeun Oh, Hyuck Yoo, Dae R. Jeong, Duc Hoang Bui, and Insik Shin. 2017. Mobile Plus: Multi-device Mobile Platform for Cross-device Functionality Sharing. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*.

[42] Oracle. 2018. JDK Swing Framework. http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.

[43] Seonwook Park, Christoph Gebhardt, Roman Rädle, Anna Maria Feit, Hana Vrzakova, Niraj Ramesh Dayama, Hui-Shyong Yeo, Clemens N. Klokmose, Aaron Quigley, Antti Oulasvirta, and Otmar Hilliges. 2018. AdaM: Adapting Multi-User Interfaces for Collaborative Environments in Real-Time. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*.

[44] Esmond Pitt and Kathy McNiff. 2001. *Java.Rmi: The Remote Method Invocation Guide.* Addison-Wesley Longman Publishing Co., Inc.

[45] Guobin Shen, Yanlin Li, and Yongguang Zhang. 2007. MobiUS: Enable Together-viewing Video Experience Across Two Mobile Devices. In *Proceedings of the 5th International Conference on Mobile Systems,*

Applications and Services (MobiSys '07).

[46] S. B. Shneiderman and C. Plaisant. 2005. *Designing the user interface*. Pearson Addison Wesley.

[47] Skia. 2019. Skia Graphics Library. https://skia.org/.

[48] Neil Smyth. 2012. *iOS 12 App Development Essentials*.

[49] Esoteric Software. 2019. Kryo. https://github.com/EsotericSoftware/kryo.

[50] Monsoon Solutions. 2019. Monsoon power monitor. https://www.msoon.com/.

[51] TeamViewer. 2018. TeamViewer - Remote Support, Remote Access, Service Desk, Online Collaboration and Meetings. https://www.teamviewer.com.

[52] Alexander Van't Hof, Hani Jamjoom, Jason Nieh, and Dan Williams. 2015. Flux: Multi-surface Computing in Android. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*.

[53] 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 (MobiCom '17)*.

[54] Jishuo Yang and Daniel Wigdor. 2014. Panelrama: Enabling Easy Specification of Cross-device Web Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*.

[55] Sai Zhang, Hao Lü, and Michael D Ernst. 2012. Finding errors in multithreaded GUI applications. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 243–253.

[56] Jiahuan Zheng, Xin Peng, Jiacheng Yang, Huaqian Cai, Gang Huang, Ying Zhang, and Wenyun Zhao. 2017. CollaDroid: Automatic Augmentation of Android Application with Lightweight Interactive Collaboration. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*.