M2: Multi-Mobile Computing

Naser AlDuaij, Alexander Van't Hof, and Jason Nieh

{alduaij, alexvh, nieh}@cs.columbia.edu

Department of Computer Science
Columbia University
Technical Report CUCS-005-15
March 2015

Abstract

With the widespread use of mobile systems, there is a growing demand for apps that can enable users to collaboratively use multiple mobile systems, including hardware device features such as cameras, displays, speakers, microphones, sensors, and input. We present M2, a system for multi-mobile computing by enabling remote sharing and combining of devices across multiple mobile systems. M2 leverages higher-level device abstractions and encoding and decoding hardware in mobile systems to define a cross-platform interface for remote device sharing to operate seamlessly across heterogeneous mobile hardware and software. M2 can be used to build new multi-mobile apps as well as make existing unmodified apps multi-mobile aware through the use of fused devices, which transparently combine multiple devices into a more capable one. We have implemented an M2 prototype on Android that operates across heterogeneous hardware and software, including using Android and iOS remote devices, the latter allowing iOS users to also run Android apps. Our results using unmodified apps from Google Play show that M2 can enable even display-intensive 2D and 3D games to use remote devices across multiple mobile systems with modest overhead and qualitative performance indistinguishable from using local device hardware.

1. Introduction

Users increasingly rely on tablets and smartphones for their everyday computing needs. Individual users often own multiple mobile systems of various shapes and sizes [26], and groups of users often have many mobile systems at their disposal. The multitude of mobile systems are useful in many ways. For example, users carry around a smaller form factor smartphone for every day use, but also bring a tablet on longer trips for a better document reading or movie watching experience with its larger screen, or just to have a system with another battery to be able to operate for a longer period of time unplugged. Similarly, a family may carry multiple mobile systems on a road trip so the parents can get driving directions on the smartphone while the kids watch movies or play games on a set of tablets.

As mobile systems become ever more ubiquitous, there is an increasing demand to provide users with a seamless experience across multiple mobile systems, not just use them as separate, individual systems. For example, the Netflix app and its supporting cloud infrastructure allows a user to start a movie on a smartphone, then switch to a tablet to continue watching the same movie with a bigger and better display instead of starting over from scratch and manually skipping around the movie to find where he left off. While this limited example only allows using one mobile system at a time, it points to an emerging trend of even more powerful ways of using mobile systems in which multiple mobile systems can combine their functions into a more capable one, enabling new applications. We call this *multi-mobile* computing.

For example, desktop computers often use multiple display monitors combined together to provide a unified, larger screen real estate on which to work. In a similar way, as shown in Figure 1, multi-mobile computing would enable users to put their tablets together side-by-side to provide a unified display and input surface across all the tablets, for a better viewing experience for all the users. As another example, group photos such as family photos are a common occurrence, but remain a vexing problem along a number of dimensions. One approach is to leave someone out of the photo to take the picture, another is to take a selfie with the usual limitation of how far away the camera can be. Given the plethora of smartphones, multi-mobile computing can enable users' smartphones to be placed wherever needed to take the picture while one smartphone is used by a user in the picture to remotely control the others in the group, view their respective camera previews, and use all the remotely controlled smartphones to take pictures, resulting in multiple photos from different angles, the best of which can be selected for use. These are just some of the many ways in which powerful new mobile apps will be developed that can take advantage of hardware devices across multiple mobile systems, including multiple cameras, displays, sensors, speakers, microphones, and input. Unlike simple one-to-one mirroring approaches such as AirPlay [3] which can display content from a smartphone to an AppleTV, multi-mobile computing goes a step further with the ability to combine multiple devices from multiple systems together.

Although multi-mobile computing has the potential to provide a wide range of powerful new apps with new functionality, key challenges must be met to turn this potential into reality. The fundamental technical challenge is how to enable apps to remotely share devices across multiple mobile systems with good performance. The lack of system support for combining multiple devices across mobile systems together forces each app developer who would like to provide such functionality to start from scratch, making such devel-

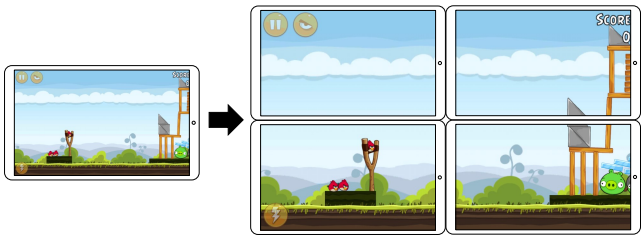

Figure 1: Multi-Mobile Computing using fused displays/input

opment difficult and error prone at best and forcing each and every developer to incur the same recurring development costs. Each app developer who attempts to develop such multi-mobile apps may come up with different approaches and user-facing user interfaces, resulting in ad hoc and unexpected interactions between multiple multi-mobile apps and an inferior user experience. Furthermore, smartphones and tablets are tightly integrated hardware platforms that come in many different sizes and incorporate a plethora of different hardware devices using non-standard interfaces; roughly 20,000 different Android systems are available [22]. This level of tremendous device heterogeneity only exacerbates the problem of how to combine multiple devices together across mobile systems.

To address these problems, we introduce M2, a system for multi-mobile computing by enabling remote sharing of heterogeneous devices across multiple mobile systems. We observe that mobile systems use devices in a manner quite different from traditional desktop and server systems. Vertically integrated mobile systems offer a tall interface from apps to hardware devices through several layers of software stack. Mobile apps do not access hardware devices through a thin layer between apps and the operating system, but rather through user-level system services that manage the hardware devices. There are some exceptions, mainly networking and storage, for which widely-used cross-platform abstractions already exist for sharing; our focus in this paper is on user-facing devices common on mobile systems such as sensors, cameras, audio, and display. Furthermore, system services manage hardware devices using native frameworks that provide interfaces similar to public application programming interfaces (APIs) used by apps. Based on these observations, M2 takes a unique approach to partitioning device functionality between a *device server*, a system that serves its devices to other systems making them remotely accessible, and a *device client*, a system that runs an app using the remote device. On the server, M2 provides device access by simply running an app that uses existing public APIs to access devices. On the client, M2 modifies user-level system services to support remote devices and expose them to apps, allowing apps to make use of multiple local and remote devices at the same time. Apps see the same device abstraction for remote devices as they do for existing local devices, enabling app developers to use the same familiar, existing public APIs for accessing remote devices. M2 leverages higher-level user-level APIs and services to operate across heterogeneous hardware and software stacks with local performance similar to the existing user-level device software stack in mobile systems.

While M2 enables app developers to use multiple devices directly, M2 introduces the notion of *fused devices*, which provide a single device abstraction based on fusing information from multiple devices. For example, a fused display device could be defined based on the local display and three other remote displays such that all four displays are to be treated as a 2x2 matrix unified together as one larger display. Instead of requiring each app developer to incur the recurring cost of creating his own mechanism or algorithm for deciding how to use multiple devices of the same type, fused devices allow developers to leverage predefined ways of combining multiple devices that may be created by other developers, thereby simplifying multi-mobile app development. Fused devices also provide a way for unmodified apps designed to interact with only one device of a given type to transparently take advantage of M2 to enable multi-mobile functionality.

M2 leverages higher-level device abstractions and widely deployed mobile system hardware features to optimize the transfer of device data across mobile systems. For low-bandwidth devices such as input and sensors, M2 simply transfers raw data formats used by device abstractions across mobile systems. For higher-bandwidth devices such as cameras, display, and audio, M2 takes advantage of encoding and decoding hardware widely deployed on mobile systems to efficiently compress device data and transfer it in well-known compressed video and audio formats. This simple approach overcomes the performance problems of previous remote display mechanisms and yields a high quality visual and audio experience across a wide range of content, including 3D graphics.

We have implemented an M2 prototype on Android and demonstrate its effectiveness at providing multi-mobile computing functionality transparently with existing unmodified Android apps using both Android and iOS remote devices. M2 allows any stock Android or iOS system to become a device server by running an app which can be made available in Google Play or the Apple App Store, and only requires modest user-level framework modifications to allow an Android system to become a device client. We show that M2 operates seamlessly across heterogeneous mobile hardware and software systems, including iOS and the latest three major Android versions, Lollipop, KitKat, and JellyBean, and multiple tablet and smartphone hardware. We demonstrate that M2 provides multi-mobile functionality with low latency and only modest performance overhead across even high-bandwidth devices such as cameras, display, and audio, even for 3D graphics-intensive apps. Using both standard WiFi networks and WiFi Direct, our experiences show that the display performance using multiple remote devices with a wide range of popular apps from Google Play is visually indistinguishable from using local devices.

2. Android Device Overview

We first provide a brief overview of the way devices are used in mobile systems, using Android as an exemplary system. As shown in Figure 2, Android can be thought of as having a tall interface to devices through multiple layers of software. Apps are written in Java and are composed of various activities implemented using Java classes. Apps call Java frameworks, which function as libraries that provide the core public APIs used by developers for

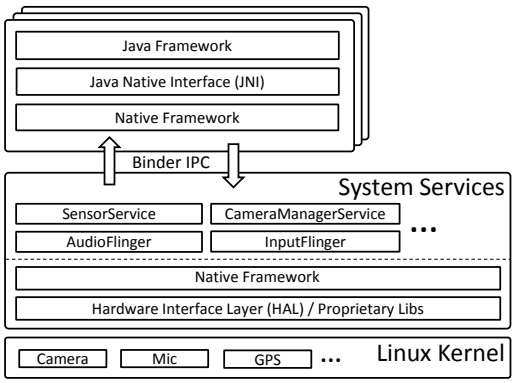

Figure 2: Android architecture

device	abstraction	interface	number of streams
sensor	event type	callback	one sensor event stream
input	source type	callback	one input event stream
location	provider name	callback	one location data stream
mic	audiosource type	method	per source
camera	camera id	method	one camera at a time
audio	implicit	method	mixes audio streams
display	surface name	method	per surface

Table 1: Android device abstractions

Android functionality including accessing devices. Frameworks use Java Native Interface (JNI) to package up calls and pass them through Android's Binder IPC mechanism to communicate with Android system services, which are shared, long-running system processes that run in the background and are used to manage devices. Almost without exception, apps do not interact with devices directly, but instead via system services that manage access to their respective devices across multiple apps.

Each type of device has an associated system service. For example, the SensorService manages sensor devices, the AudioService manages audio devices, and the CameraManagerService manages the camera device. System services implement vendor-independent software-related device functionality using a plethora of native frameworks provided by Android. Services interface with the Hardware Abstraction Layer (HAL), a standardized Android interface for accessing hardware, to call vendor-specific libraries, some of which are proprietary, which implement vendor-specific device functionality. These libraries interface with the Linux operating system kernel to access the device hardware via device drivers. Other mobile ecosystems such as iOS have similar software stacks for accessing devices in which higher-level frameworks communicate with underlying system services via an IPC mechanism, and those system services then manage lower-level device functionality.

Mobile apps do not see the traditional file-based device abstraction provided by the kernel, but instead interact with whatever abstraction is provided by system services. Instead of a uniform device abstraction for all devices, each system service provides its own specialized abstraction for its own type of device. Table 1 lists the major types of user-facing I/O devices supported in Android and the respective device abstractions used.

The device abstraction provided by frameworks to apps and supported by system services is generally an object with an associated identifier or name. For sensor, input, microphone, and camera, the device abstraction object is an event, source, audiosource, and camera, respectively, each of which is identified by a number. For example, an accelerometer sensor is a different numerical type than an orientation sensor. For location and display, the device abstraction object is a provider and surface, respectively, each of which is identified by name. For audio, the device abstraction is implicit, as the system automatically determines the device used for audio output. Device data is provided or in one of two ways. For asynchronous events such as sensor, input, and location data, an app registers a callback listener method that is called when the respective data is available. Otherwise, an app directly calls a method for using the device, as in the case of microphone, camera, audio, and display.

Devices have different models in terms of whether data is provided in a single stream or multiple streams, and whether multiple devices of a given type are accessible at the same time. For sensors, each app activity receives all data that it registers for as one sensor event stream returning to the same callback function, so the app is responsible for examining the event type of each object to determine the sensor providing the data. This model applies to input and location devices as well. For microphones, each app may access multiple microphones at the same time based on audiosource type. For display, each app may draw to multiple display surfaces, which are composed together by `SurfaceFlinger`. For camera, although multiple cameras can be accessed, an app can typically only access one camera at a time. Finally, the audio device is implicit and not directly identified by apps, though apps may process multiple streams of audio which `AudioFlinger` will mix together.

3. Usage Model

M2 is designed to be simple to use by both users who want to use multiple mobile systems, and developers, who want to develop apps and functionality using the M2 application programming interface (API). We first discuss the usage model for end users, then discuss the API for developers as well as various mechanisms provided by M2 to transparently support unmodified apps.

A mobile system is a server if it has a device that is being shared with other systems, and is a client if it is accessing a device being shared by another server. Apps that access remote devices are run on the client, whereas servers simply make their devices accessible. A client may use multiple servers, a server may be in use by multiple clients, and a mobile system can be both a server and a client at the same time.

Users can turn their mobile systems into servers by simply downloading the M2 app from the respective app store. For example, an M2 Android app would be used for Android systems, and an M2 iOS app would be used for iOS systems. No other software is needed to allow a mobile system to share its devices with other systems. By default, no devices are shared. Using the app, the user can share one or more devices by creating a *device profile*. A device profile consists of a profile name, a list of devices being shared, a password, and optional access control options that can restrict the systems that can access the device based on IP.

For each device listed in the profile, the user can specify a limit on how many clients can access a device at the same time; for example, a limit of one means that only one client can access the device at a time. Devices not listed in a profile are by default not shared. Device profiles can also be disabled at any time so that they cannot be used by other mobile systems.

Device data on the server is processed by the M2 app. Whenever the app is running, device data can be captured and sent to the client. User-related input and output is processed when the app is visible to the user. For example, when the input device is shared, input data is captured by running the M2 app by processing touchscreen and button input just like any other app and forwarding it to the client. Similarly, when the display device is shared, display output data from the client is made visible by drawing the data to the server's screen when the M2 app is running in the foreground and visible to the user. From M2's perspective, the M2 app simply makes the devices on the server system accessible remotely, and otherwise treats the server like a dumb, stateless peripheral system. At the same time, if using an Android system, the M2 app runs like any other Android app and users can switch between the M2 app, treating the system like a stateless peripheral, and any other Android app, treating the system as a full-fledged Android computer.

To run apps that access remote devices on other server mobile systems, the M2 native frameworks must be installed on the client mobile system. Once installed, a user can make remote servers accessible by downloading and running the M2 app on the client. Using the app, the user can specify a device profile on a server, input the required password, and the respective remote devices will then be accessible on the client. Apps running on the client can then access those remote devices. A server device profile remains active on the client until it times out from inactivity or is explicitly disabled by either the server or client. The M2 app shows both currently active device profiles as well as previously accessed device profiles that are not currently active, the later to make them easy to access again in the future. Accessing device profiles can also be done by other apps in a programmatic fashion.

Users may be concerned about unauthorized access to devices on their mobile system. Device profiles provide security to prevent outsiders not running on the user's system from accessing the user's devices. Our goal with M2 is to ensure that it does not result in additional security risks with remote device access than the security mechanisms currently provided by mobile systems. In the case of standard Android apps, once a user has granted the app permission to access various devices such as location services, cameras, and the network, an app is free to capture that data and send it elsewhere. As a result, M2 works to prevent unauthorized access from outside the local system, but does not guard against unauthorized access to local devices by apps already given permission by the user to run on the local system.

Given that a number of devices may be available on a client, M2 allows users to define *usage profiles* to indicate which collection of devices are to be used by an app. A usage profile specifies which devices from which server profiles are to be used. Usage profiles are ordered, so that M2 will select the first usage profile for which all its devices are available. For example, if a usage profile for using a particular server tablet's display is ordered before a usage profile for using the local system's display, then M2 will use the remote display whenever it is available and only use the local system's display if the server tablet is not available. Note that availability is determined dynamically as servers may disconnect at any time given the nature of mobile systems and networks. Usage profiles can be defined to be system-wide, or can be used on a per application basis so that different applications may use different usage profiles at a given time.

A usage profile not only indicates which server devices are to be used, but may also indicate further information about how they are to be used. For example, how a set of devices is used may depend on the relative positioning of the mobile systems. We expect that in the future, M2 will provide mechanisms to automatically detect the relative position of mobile devices as positioning changes dynamically [35], but for simplicity, relative position is currently determined based on user input, either statically as part of the usage profile or dynamically when the usage profile is selected for use. For example, given a set of mobile systems, the M2 app currently assumes that the systems are positioned in an NxM matrix and asks users to swipe across left to right each row of systems to determine their relative positioning. This information can then be used by other apps, for example in determining how to display visual content across multiple screens, or how to output different channels of audio content to different speakers.

From an app developer's perspective, remote devices available through M2 and normal local devices are all just devices. M2 simply provides a model of making multiple devices available to apps running on the client. For example, if a system has a local display and can access three remote server displays, an app then can pick and choose among the four different displays, or use them all together. Based on this approach, M2 is mostly compatible with standard APIs provided by mobile ecosystems such as Android, so that all standard APIs used for interacting with local devices can also be used for interacting with remote devices provided by M2. This includes APIs for queries devices and their capabilities. How an app uses the combination of local and remote devices available to it is app-dependent and up to the developer. Section 4 describes the M2 API in further detail.

In addition to making individual remote devices available to apps, M2 introduces *fused devices*, which provide a single device abstraction based on fusing information from multiple devices. The idea is similar to fused location providers in Android [15], which combine GPS, WiFi, and cellular services to improve location accuracy. For example, a fused display device could be defined based on the local display and three other server displays such that all four are to be treated as a 2x2 matrix unified together as one larger display. Input can also be handled in the 2x2 matrix by using a fused input device to control the fused display device. Similarly, a fused camera device could be defined to combine the previews of multiple cameras together by tiling them in a single preview display. Instead of requiring each app developer to incur the recurring cost of creating his own mechanism or algorithm

for deciding how to use multiple devices of the same type, fused devices allow developers to leverage predefined ways of combining multiple devices that may be created by other developers, thereby simplifying multi-mobile app development. Fused devices also provide a way for legacy apps designed to interact with only one device of a given type to transparently be able to take advantage of M2's multi-mobile functionality without requiring them to be modified or rewritten. For example, instead of using the local display, M2 can enable an app to use a fused display device, appearing to the app to be the same as the local display but instead allow the app to display content across multiple displays transparently. Fused devices appear to apps and users as any other device, and can therefore be selected by usage profiles and apps in the same manner as other individual devices.

4. M2 API and Fused Devices

M2 provides an API to support new multi-mobile apps that desire explicit control of multiple remote devices, in addition to enabling existing unmodified applications to become multi-mobile aware via fused devices. Using Android, we describe the M2 API, which leverages Android's existing public API to utilize M2's multi-mobile computing paradigm. We also discuss examples of fused devices for different types of devices. While M2 also provides APIs for device discovery to programmatically provide functionality described in Section 3, we omit this due to space constraints and focus here on the API for app developers writing mobile apps.

Sensors M2 uses the same Android APIs for sensors based on callbacks and event types indicating the type of sensor providing the respective event. These types are defined in Android as numerical identifiers, currently ranging from 0 to 21. To support the same interface for sensor devices across multiple mobile systems, we assign a system number to each mobile system. We then modify the framework to cap the number of local sensor types to 255, a number much greater than the current number of sensor types available, and change the type of the sensor events from remote devices to its type plus the remote system number times 256. For example, accelerometer data received from remote system 1 would have type 256 instead of type 1 for the local system. Unmodified apps are unaffected by M2 sensor data and new apps developed using this API can select sensor data from multiple mobile systems. Note that different systems may have different sensors. For example, an app running on a Nexus 7 tablet can access a barometric pressure sensor on a Nexus 4 smartphone that is not available on the Nexus 7.

To support existing unmodified apps, M2 provides a fused device in which the type identifiers are not remapped to different ranges based on remote system number, but instead all retain the original numerical range. Based on usage profiles discussed in Section 3, different sensors from different remote systems can be selected and used, with the restriction that only one type of each sensor can be used at a time. This enables existing apps to take advantage of remote sensors without any modifications.

Input M2 leverages existing Android APIs that support the addition and use of multiple input devices such as game controllers. Each such device is identified by a source type. M2 assigns unique input source identifiers for remote input devices in the same manner. Apps using multiple input devices can check for the source type of the input event object to determine the remote device that sent the input and process it accordingly.

To support existing unmodified apps, M2 provides a fused device which simply combines input events from multiple remote input devices. Existing unmodified apps can be controlled with multiple remote input devices even if they were designed with only one input device in mind because they simply do not check for the source type when using the standard callbacks for processing touchscreen input events, so no further assignment of source types is required. This can be useful when multiple touchscreen input devices are used in turn by users, not all at once.

Microphone M2 leverages existing Android APIs that already facilitate the use of multiple microphone devices by simply adding remote microphone devices as additional audio sources. No API changes are needed. The same APIs used for local audio sources are used for remote audio sources, such as Android's `AudioRecord` and `MediaRecorder` APIs.

To support existing unmodified apps, M2 provides a fused device which simply combines multiple remote audio sources by treating them as audio tracks to the same audio mixing mechanism used by Android's audio device. Alternatively, individual remote microphones may also be selected using usage profiles.

Location M2 leverages existing Android APIs, test providers, to support use of multiple location devices. Test providers are added to location services to represent remote location services and are simply named `remoteN`, where `N` is the remote system number. Apps can request location data by simply specifying the provider's name through Android's `requestLocationUpdates` method. No API changes are needed.

To support existing unmodified apps, M2 provides a fused device by using existing Android fused location devices, which are already used for combining data from multiple sources such as GPS, WiFi, and Cellular. In this manner, multiple remote location sources can be combined to provide more accurate location information. For example, indoor localization may benefit from fusing local and remote location data, where the latter comes from a system that is outside or next to a window. Alternatively, individual remote location devices may also be selected using usage profiles.

Camera M2 leverages existing Android APIs for using multiple camera devices by simply adding remote cameras as additional camera devices available on the local system with unique camera identifiers. Apps can open remote cameras by using the respective matching identifiers. No syntactic API changes are needed, but the underlying semantics are changed slightly as existing APIs do not support simultaneous use of multiple camera devices.

To support existing unmodified apps, M2 provides a fused device that combines preview data from multiple camera sources and unifies it into one display. Taking a picture results in multiple photo snapshots being taken. Alternatively, individual remote camera devices may also be selected using usage profiles.

Audio Unfortunately, existing Android APIs provide no support for enabling apps to explicitly specify either which audio device to use, or being able to use multiple audio devices. However, earlier Android APIs that are currently deprecated do provide this capability, such as the `setRouting` method which sets the audio device to use. In the context of supporting only local audio devices such as headphones and speakers, the lack of an API for choosing the audio device makes sense because the choice really belongs to the user and, for example, whether the headphones are plugged in or not. However, given that M2 provides support for multiple remote devices, this pre-existing Android API makes sense. M2 leverages this Android API to facilitate explicit specification of audio devices, including the use of multiple remote audio devices. Other audio APIs remain the same and can be used with both local and remote audio devices.

To support existing unmodified apps, M2 provides a fused device that sends audio data to multiple remote audio devices for playback, along with separate control channel information to each device. This control channel information can be used with a mask to represent audio channel information, such as 0 for left channel, 1 for right channel, and 2 for both channels. Remote audio devices can then mask out the channel data that is not needed and play the respective audio channel to provide stereo output across multiple remote audio devices. Current Android frameworks are limited to supporting two channel audio, so 5.1 surround sound is not possible with a fused device.

Display M2 leverages existing Android APIs that already facilitate the use of multiple display surfaces, which are used by M2 to represent remote displays in the same manner as the local display. Apps may utilize display surfaces in conjunction with Android's public API to encode and decode display data to reduce bandwidth costs during transmission over the network. No API changes are required. Section 5 provides further detail on how encoding and decoding are used by M2.

To support existing unmodified apps, M2 can provides various fused display devices, depending on the desired functionality. For example, to enable multiple remote displays to be used together as one larger display, M2 sends the same display data to multiple remote display devices along with separate control channel information to each device. This control channel information indicates the relative position of each display and what portion of the content should be shown on the respective display. Remote display devices can then scale and clip the display data based on the control information to display the right portion of the data to create a unified combined display effect across multiple display devices. M2 is able to support recording the fullscreen and displaying a portion of the screen on the same local device by assigning different layer stack numbers to `Surfaces` at the `SurfaceFlinger` system service level.

5. M2 Architecture

The M2 architecture addresses two key issues in enabling remote device sharing to support multi-mobile apps. The first issue is how should device functionality be partitioned between a server which is making its device remotely accessible, and a client where the mobile app will execute and make use of the devices. The second issue is how should the system transmit device data over the network in an effective low-latency manner for both low-bandwidth and high-bandwidth devices.

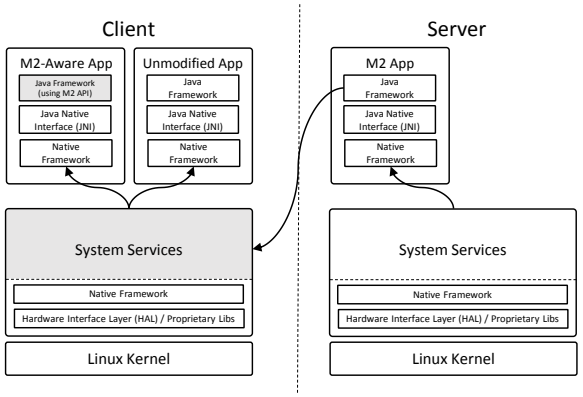

Figure 3: Overview of M2 architecture; modified Android components highlighted in grey

5.1 Client-server device stack

Given the background regarding the Android device infrastructure discussed in Section 2, there are a number of ways in which device functionality can be partitioned between server and client. One approach would be to partition the device stack at the kernel interface using traditional device files as the abstraction between server and client. The server where the device resides has the real device file, and the client has a virtual device file which essentially forwards device interactions to the server, as done by Rio [1]. However, this approach has a number of problems. First, since this is done below the HAL at the level of vendor-specific devices, it only works for systems which use the same vendor-specific hardware devices and any attempt to operate across different hardware is problematic. Second, operations at the device file level may involve pointer references to memory in the address space of the calling process and providing a shared memory abstraction across systems to support these device-level operations would be complicated and likely to suffer from performance degradations at best or even worse, service failures, as a result of device disconnections due to network drops. Third, common device-level operations on mobile systems involve vendor-specific `ioctls` which may depend on more than just the input arguments or may involve references to memory which would be extremely difficult to identify and reproduce correctly across systems for any moderately complicated device. Finally, device vendors may choose to implement substantial device functionality in proprietary vendor libraries with very little functionality in the driver itself and these vendor libraries may manipulate device state via shared memory in any opaque manner such that recreating the same effect with virtual device files simply forwarding calls to the real device files is problematic at best.

Another approach would be to partition the device stack at the HAL layer given that it is a device abstraction layer, but this approach also has various problems. First, at the server, system services expect to manage underlying devices on behalf of An-

droid apps running on the server, so avoiding device conflicts with unmodified system services becomes problematic as they are unaware of the devices being used by a remoting mechanism at the HAL layer. Second, it may be desirable to process device data in some fashion to optimize network performance. The HAL layer provides no support for implementing such additional functionality, as HAL support is often implemented in proprietary vendor-specific libraries, making it more difficult to implement any functionality required beyond basic forwarding between server and client. Furthermore, since the HAL layer is low-level, higher-level abstractions provided by Android are unavailable, making it more difficult to leverage higher-level semantics for processing data. Finally, because the HAL layer is low-level and Android dependent, providing device servers at the HAL layer would require modifications that preclude using this approach with stock Android systems, and would make it difficult at best to use this approach with non-Android systems even if they were modified to support device remoting.

We make three observations regarding the device software stack in mobile systems that suggest an approach for partitioning device functionality between server and client in M2. First, for the hardware devices of interest such as input, camera, audio, display, sensors, and GPS, mobile apps do not access such hardware devices directly but go through system services. As a result, there is no need to provide apps with device abstractions corresponding to remote devices at lower layers of the software stack below system services because apps will never see them; anything below the level of system services is irrelevant as far as apps are concerned. Second, the primary interface between apps and devices is at user-level, not kernel-level. System services are entirely implemented at user-level. This suggests that a user-level approach to remote device access is likely to be sufficient given that a user-level approach is used for local device access. In this context, traditional arguments in favor of kernel-level approaches for performance reasons are less likely to apply to mobile systems given their device software stacks. Finally, the native frameworks used to implement system services provide many of the same interfaces and functions as the Java frameworks used by apps. Although the former represents an internal API, its similarities in the context of device access to the public API used by apps suggests that, at least for those those similar APIs, they are likely to remain relatively stable across different versions of Android.

Based on these observations, M2 takes a unique approach to partitioning device functionality between server and client that leverages the characteristics of mobile systems. On the server, M2 provides device access by implementing it using public APIs provided by Java frameworks to apps; this access via Java frameworks is forwarded on to the client. On the client, M2 modifies system services to support remote devices and expose them to apps. Instead of only calling native frameworks to access local devices, system services interface with native frameworks modified to also redirect to remote devices. Figure 3 shows an overview of the M2 architecture in Android.

This split architecture provides multiple benefits. On the server, because device access at the server is provided at user-level entirely using public APIs, server device access can be provided entirely by running an app without any changes to the software stack on the server. This provides maximum flexibility and ease of development as providing device access is no more difficult to developing other mobile apps. By building on public APIs that are supported for all Android apps across all Android versions, it is easy and straightforward to support any Android hardware and software platforms, enabling device remoting across heterogeneous devices. Since the server is just an app, its interactions with devices are managed by system services along with any other app, allowing remoting of devices and use of those devices by apps running on the system to co-exist seamlessly using existing mechanisms. Furthermore, given that servers are built on public APIs typical of most mobile ecosystems, we expect that building servers for non-Android systems such as iOS and enabling iOS devices to be remotely shared would be relatively straightforward. Although server device access is provided at user-level, device access in general for standard Android apps using local devices is also provided at user-level, so we expect that a user-level implementation will not adversely affect performance.

On the client, because device access is implemented within system services, remote devices are available to apps running on the client with the same interface as other local devices also provided by system services, making it easy for apps to use remote devices in the same manner as they use local devices. By implementing remote devices at the highest layer compatible with how other local devices are accessed by apps, M2 can leverage higher-level semantics available at that layer to simplify its implementation and avoid low-level implementation complexities and device-specific dependencies. By avoiding device-specific dependencies, M2 easily supports heterogeneous Android hardware devices. Since system services are implemented using native frameworks, M2 can leverage those same frameworks to reduce implementation complexity rather than having to reimplement low-level interfaces such as the HAL, which while providing a device independent interface, is implemented in practice by vendor-specific libraries that are often proprietary especially for more complex devices, and those cannot be reused to aid the development of supporting comparable remote devices in M2. The fact that the native frameworks often provide similar functionality to the Java frameworks that provide the public API for apps often makes it easy to make from one to the other in providing straightforward implementation support within system services for various remote devices. Although native frameworks are considered internal APIs that may change, M2's use of a subset of native frameworks that are mostly similar to those used to support public APIs reduces the likelihood of changes to the specific internal APIs used by M2, making it easier to port and support across different Android versions. As evidence for this, our M2 prototype implementation has been tested to work, reusing the exact same code, across the latest three major Android versions, JellyBean, KitKat, and Lollipop.

device	data	fidelity	reliability
sensor	raw	lossless	retry
input	raw	lossless	retry
location	raw	lossless	retry
mic	raw/encode	lossless/lossy	retry/discard
camera	raw/encode	lossless/lossy	retry/discard
audio	raw/encode	lossless/lossy	discard
display	encode	lossy	discard

Table 2: M2 network communication mechanisms

By relying on the decoupling of apps from devices provided by system services, M2 treats remote devices as dumb peripherals that are stateless from the point of view of apps. This provides useful properties in the presence of intermittent network disconnections between server and client due to system mobility or other environment conditions affecting wireless networks. App state is entirely on the client and network disconnections do not cause app failures. For example, app-related graphics and display state is entirely on the client encapsulated in state in the app as well as display surfaces managed by `SurfaceFlinger`, the Android display-related system service. If a disconnection happens, the app continues to function properly and can continue to draw to its respective display surface, oblivious as to whether the system service is still able to send the data to the remote display device.

5.2 Data and Network Communication

M2 clients and servers communicate over standard network sockets and are designed to interact over WiFi and WiFi Direct networks. To optimize network performance in the presence of both low-bandwidth and high-bandwidth devices, there are three key issues that need to be addressed: the choice of primitives to use in communicating device data between clients and servers, whether the data is communicated using lossless or lossy mechanisms, and how data loss is handled. Table 2 provides a summary of how M2 addresses these issues for different devices.

A primary issue in remoting devices is the choice of wire protocol primitives used for transmitting data. For low-bandwidth devices, the choice is not as crucial and device data can simply be transmitted in raw form in a manner that makes it simple to process at the client and server without concern for network performance. For high-bandwidth devices such as display, the choice of primitives for transmitting data can be crucial for ensuring adequate remote device performance. For example, simply sending raw display frames at native resolution for a tablet at internal 60 frames per second (fps) update rates would require gigabit network speeds just for mirroring display content to only one device. This is clearly not feasible on current WiFi networks.

M2 leverages common hardware features on smartphones and tablets to address this problem. These systems now include video and audio encoding and decoding hardware, the former of which would be much too expensive to do in software on modern CPUs in the absence of hardware support. Since high-bandwidth devices send visual and audio data, M2 simply uses hardware video encoding to compress display data and hardware audio encoding to compress audio data before transmitting it across the network. At the server, the data is decoded and outputed. A benefit of this approach is that the bandwidth required to display high fidelity content is limited by the display resolution and frame rate, so even complex 3D graphics scenery does not require more bandwidth than 2D imagery.

M2 uses H.264 video encoding and AAC audio encoding for display and audio devices, respectively, which are commonly available on smartphones and tablets, though other encoding formats can also be used. These encoders can be configured to use different resolutions, bit rates, and frame rates, which M2 can adjust based on what devices are being used and available bandwidth; M2 by default uses 30 fps frame rates since they are visually indistinguishable from higher frame rates for end users [28]. Both camera and microphone also send video and audio data, which can also be encoded. In the case of camera, M2 encodes the camera preview data, which can be bandwidth intensive if sending raw frames, but does not encode the actual pictures taken, which are transmitted much less frequently.

An interesting issue in the context of this encoder/decoder limit is how display data is transmitted from a client using multiple remote devices to those remote devices for display. In the case of multiple displays being combined into one larger display so that each device only displays a portion of the data, it may be desirable to send each device only its respective portion of the display data to reduce network bandwidth requirements. However, this requires that display data for each remote device be separately encoded at the client. Given the limited number of encoder/decoder streams supported, M2 instead encodes the complete display data and transmits the same encoded data to all remote devices and provides control information to indicate to each device what portion of the display data should be shown. Although this uses some additional bandwidth, it saves on the number of encoders used at the client. The display data can also be scaled and resized appropriately for viewing at the remote display based on the hardware characteristics of the respective screen.

A second issue in remoting devices is the fidelity of the device data and whether or not the data must be precise and lossless, or can be approximate and lossy. For most devices, lossless data is expected, and there is really not much cost to providing this precise fidelity for low-bandwidth devices. However, for high-bandwidth devices such as display, M2 uses H.264 video encoding which provides substantial bandwidth reductions by encoding the data in a lossy format. However, the primary concern with data fidelity in this case is its visual appearance to end users when displayed, and advances in encoding algorithms and implementations have made it that despite the use of a lossy format, the visual difference versus the lossless data, assuming suitable video encoding configuration parameters, can be indistinguishable to the user. While display data is always sent encoded and therefore lossy, audio data can be sent raw and lossless or encoded and lossy, trading off bandwidth and hardware encoder availability. For camera, the camera preview is encoded and lossy, but to preserve the fidelity of the pictures taken, the picture data is not further encoded beyond what is done at the device and is therefore transmitted in a lossless manner.

A third issue in remoting devices is the reliability of the data transmission and how packet drops should be handled. In terms of

network protocols, TCP is used for all control messages, and a mix of TCP and UDP are used for transmitting data, depending on the type of data being sent. For devices which expect precise, lossless data, M2 uses TCP which retransmits data if needed to ensure reliable data delivery. For devices such as display, audio, and the camera preview, UDP and RTP is used instead because it is more important for data to be delivered on time than to ensure that all data is transmitted reliably. Data that is late as indicated by timestamps or not delivered due to packet loss is simply discarded. The microphone is an interesting case as whether lost data is acceptable depends on how it is used, and in fact, systems such as Android separately identify whether the microphone is used to record audio to a file or simply for audio input as for teleconferencing. In the former case, it may be desirable to ensure reliable transmission with some delay in the case of retransmissions as opposed to real-time delivery with some loss. M2 makes this a configurable option with the default case using UDP and RTP as teleconferencing is a much more common use case for mobile systems. Note that although it may appear that UDP Multicast may be a better protocol for devices such as display which may transmit the same data to multiple remote devices, UDP Multicast implementations have poor performance and frequently are not even supported with current WiFi routers. As multicast implementations improve in the future, this may become a viable option to using point-to-point network protocols.

Finally, M2 uses a push model for all devices such that the sending side, the client in the case of output devices and the server in the case of input devices, pushes data to the other side, as opposed to waiting for it to be pulled on request. Even in the case of input devices processed with callback functions, the callback is executed on the device server and the device data is immediately pushed to the client. This reduces latency, which is important for device interactions occurring over the network.

6. Evaluation

Using the Android Open Source Project (AOSP), we implemented an M2 prototype and measured its performance across a range of Android hardware and software, including both tablets and smartphones. We also implemented an M2 iOS device server, demonstrating the ability to access remote devices across heterogeneous hardware and software, including iOS on an iPad mini. We present results using Nexus 4 (768x1280 display, Qualcomm Snapdragon S4 Pro 1.5GHz quad-core CPU) smartphones with Android 4.3 JellyBean, Nexus 7 (1200x1920 display, Qualcomm Snapdragon S4 Pro 1.5GHz quad-core CPU) tablets with Android 4.4 KitKat, Nexus 9 (1536x2048 display, Nvidia Tegra K1 2.3GHz dual-core CPU) tablets with Android 5.0 Lollipop, and a 1st generation iPad mini (768x1024 display, Apple A5 1GHz dual-core CPU) tablet with iOS 8.2. We conducted experiments with both WiFi Direct and regular WiFi, the latter by connecting systems to an ASUS RT-AC66U WiFi router; the router was used by default unless otherwise indicated. Only the Nexus 9 supports and uses IEEE 802.11ac, while the other systems use IEEE 802.11n. We run both benchmarks and unmodified Android apps from Google Play [14] to demonstrate the effectiveness of M2 in delivering multi-mobile computing transparently to existing real-world apps with good performance across heterogeneous Android hardware and software.

We first focus on measuring display performance since it is crucial for mobile systems and a key challenge for remoting performance. This is done by configuring one or more systems as display and input device servers for a client running an Android app. To measure real app performance, we used the widely-used Android PassMark benchmark [23]. PassMark conducts a wide range of resource intensive tests to evaluate CPU, memory, I/O, and graphics performance. In cases when display clients drop frames, the reported benchmark results may not reflect the performance perceived at the clients since the app does not account for this in reporting benchmark performance. To account for this difference, we use metrics introduced by slow-motion benchmarking [21] by scaling the performance results based on the percentage of frames displayed at the server. For example, if only half of the frames are displayed by the server, then the benchmark measurement reported by the app, assuming higher is better, is reduced by half.

We ran M2 with PassMark in seven system configurations using a Nexus 9 to run the app: (1) stock Android Lollipop, (2) M2 installed but idle, (3) M2 displaying locally on the same system, (4) using two Nexus 9 systems, splitting the display across the Nexus 9 client and another Nexus 9 display server, (5) using four Nexus 9 systems in a 2x2 configuration combined as one display, splitting the display across the Nexus 9 client and three other Nexus 9 display servers, (6) using a mix of four heterogeneous systems, one-to-many mirroring the Nexus 9 display to a Nexus 9, a Nexus 7, and a Nexus 4, and (7) using a mix of four heterogeneous systems, one-to-many mirroring the Nexus 9 display to a Nexus 7, a Nexus 4, and an iPad mini. We used the full 1536x2048 native display resolution for all Nexus 9 experiments; display encoding was done at 30 fps and a 10 Mbps bit rate. The high resolution and bit rate were used to stress the system. For the mixed cases we used a 720x1280 display resolution for all experiments since there is a resolution limit imposed by the Nexus 7 H.264 hardware decoder; display encoding was done at 30 fps and a 4 Mbps bit rate. To demonstrate the ability to run M2 without additional network infrastructure, all tests were done using WiFi Direct, except for the last one, which used the WiFi router since the iPad mini does not support WiFi Direct.

Figure 4 shows the PassMark benchmark measurements normalized to stock Android Lollipop performance; lower is better. M2 idle is omitted since it performed essentially the same as stock Android. Due to space constraints, the individual tests are grouped under CPU, disk, and memory using PassMark's overall score for those categories, while the 2D and 3D individual tests are shown separately. For the two and four system experiments, we present results for the worst remote device; in all cases except using iOS, the remote devices performed similarly. Figure 4 shows that M2 incurs some additional overhead as the number of remote display devices increases, but it is modest and in some tests uncorrelated with the number of devices used. This suggests that for modest numbers of displays, M2 performance does not degrade as additional displays are added. In all cases, the network was not a performance bottleneck and dropping frames or packets was not an

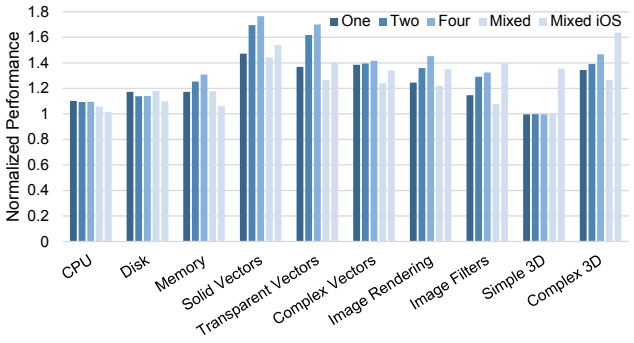

Figure 4: PassMark performance; lower is better

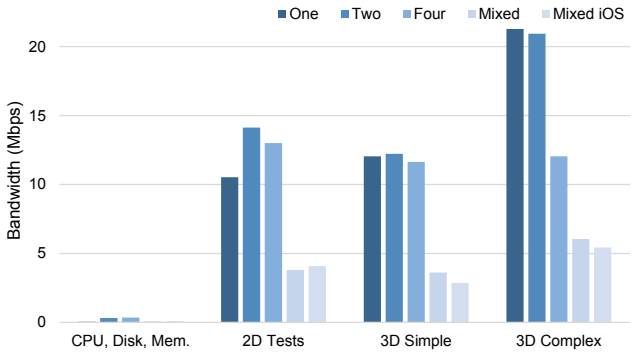

Figure 5: PassMark per device bandwidth

issue. In comparing the mixed display and homogeneous display measurements, both using four Android systems, the performance is better for the mixed display system because of the lower resolution and bit rate used, imposing less work on the client running the app. When using multiple displays, the display quality across the devices appeared qualitatively the same, except for using the iPad mini, which performed well for all tests except the 3D tests, which were noticeably worse than other systems. The iOS test ran on an older 2012 iPad mini and only uses software decoding because we did not have time to implement hardware decoding, so for the 3D tests, it struggled to decoding display data fast enough and had to skip frames, up to 35% of them for the 3D complex test. If accepted for publication, we will include iOS results for hardware decoding.

Performance overhead for the Android remote devices was qualitatively good visually, but quantitatively shows a range of performance overhead from less than 1% for the 3D simple test to more than 70% for the 2D solid vectors test. Some of this performance degradation is due to the extra work that M2 does in this case with the substantially higher resolution display of 1536x2048, but most of the performance degradation is due to an unoptimized implementation of how M2 displays content on the local display device, thereby slowing the execution of the benchmark for some tests. In the current implementation, M2 displays the content on the local device by treating it the same way as a remote device, incurring all the same costs of encoding and decoding as well as copying the data multiple times as though it were being passed to a remote display server. An obvious local device optimization is to remove all of this additional overhead and simply display the visual content directly rather than going through encoding, decoding, sending data, and the other steps used for displaying on clients. To provide a rough measure of the overhead that would be expected using M2 with this optimization, we ran the same experiments but without drawing the content on the Nexus 9 local device, therefore skipping the decoding and copying steps on the server. For example, the results for the one device case showed the same performance as stock Android, eliminating over 40% of the overhead for the 2D solid vectors tests; carrying over to the multi-device experiments brings the performance overhead overall below 30% in the worst case. We note that PassMark is designed to stress test the system, and we do not expect users of real apps to experience any qualitative performance degradation.

Figure 5 shows the per device average network bandwidth required while running the PassMark tests, aggregated into the minimally graphical CPU, disk, and memory tests, 2D tests, 3D simple test, and 3D complex test. Note that the network bandwidth required on the client running the benchmark is the bandwidth shown times the number of remote devices as it sends the display data to each of the remote devices. For the CPU, disk, and memory tests, the bandwidth required was less than .1 Mbps as the only display updates are for a progress bar for each test and the display of the test results. For the 2D and 3D simple tests, the bandwidth required was up to 14 Mbps for the 1536x2048 remote display tests and up to 4 Mbps for the 720x1280 remote display tests. Note that the 2D test bandwidth exhibits more differences than the 3D test as there was more variation in the performance results and therefore what display data was actually encoded. For the 3D complex test, the bandwidth required ranged from 13 to 20 Mbps for the 1536x2048 remote display tests and was roughly 6 Mbps for the 720x1280 remote display tests. The lower per device bandwidth consumption for the four system 3D complex test was due to somewhat worse benchmark performance and the resulting differences in the display data encoded, not packet loss. Our results show that WiFi networks can meet the bandwidth requirements for even 3D graphics-intensive display data, providing good M2 performance.

We next focus on measuring camera latency performance. Unfortunately, there is a lack of standardized Android camera performance app benchmarks, so we simply ran the default Android camera app for each system and instrumented it to measure the time to take a picture including committing it to persistent storage, and in the case of using a remote camera, the bandwidth requirements for both the camera preview and transferring the picture taken from the remote camera to the default local storage for the app. We measured the performance using stock Android on all three Android systems, the Nexus 9, Nexus 7, and Nexus 4, and compared to four different remote camera scenarios, the Nexus 7 using a remote Nexus 4 camera, the Nexus 7 using a remote Nexus 9 camera, the Nexus 4 using a remote Nexus 9 camera, and the Nexus 4 using a remote Nexus 7 camera. The first two remoting scenarios illustrate using a higher quality remote camera to take pictures as both the Nexus 4 and Nexus 9 cameras are higher quality than the Nexus 7. The second two remoting scenarios

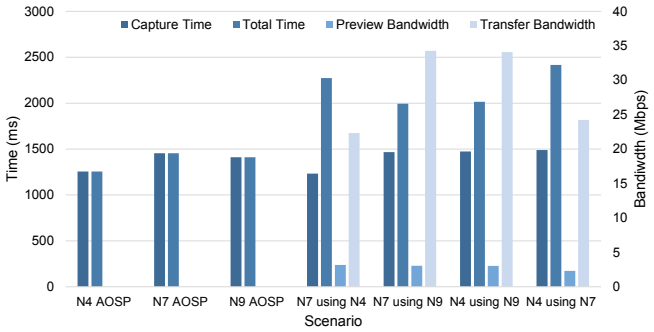

Figure 6: Camera latency for taking/storing pictures

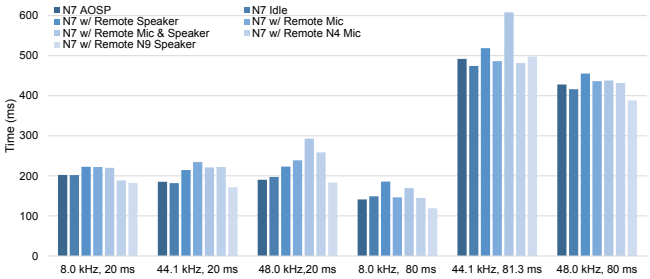

Figure 7: Zoiper audio latency

illustrate using a small form factor system, the Nexus 4, to control cameras on larger form factor tablets, the Nexus 9 and Nexus 7.

Figure 6 shows the camera performance measurements. For the time to take a picture, we show capture time, the time from the button press until the picture is saved to storage, and total time, the time until the picture is synced to persistent storage, including transferring it over the network in the case of remote devices. Note that the capture time is not the same as the time it takes for the user interface to indicate that it is ready to take another picture, which is faster but not a true measure of actual camera performance. For the stock systems using the local camera, the capture and total time are roughly the same, taking less than 1.5 seconds in all cases with the Nexus 4 (N4) camera being the fastest; syncing time is negligible compared to capture time. For the remote camera scenarios, capture time is comparable to the respective local camera capture time, with the remote Nexus 4 camera being the fastest as well. The capture times show that M2 incurs negligible additional latency versus local camera use. Total time for the remote camera scenarios is much higher because of the time it takes to transfer the picture over the network to the default local storage of the app on the client. In the worst case of the Nexus 7 using the Nexus 4 remote camera, the total time is almost a second more than the capture time due to transfer time. In contrast, the difference between the capture and total time for the remote Nexus 9 camera scenarios was only half a second because it uses the faster 802.11ac networking standard. Figure 6 also shows the bandwidth requirements for the camera preview and picture transfer. The camera preview runs at a lower resolution than the native display resolution, so its bandwidth requirements are less than 3 Mbps. The picture transfer bandwidth is higher simply because M2 sends the picture as fast as it can from the remote camera server to the client, so it uses as much bandwidth as possible, almost 35 Mbps for the faster Nexus 9.

We next focus on measuring audio and microphone latency performance. We used the Zoiper [27] audio benchmarking app, which measures the time from playing a beep through the speaker, recording it through the microphone, and retrieving the audio buffer. Zoiper tests different sample rates for recording the audio, from 8 to 48 KHz, and different audio buffer sizes for storing the audio, from recording 20 to 80 ms of audio through the microphone. The results depend on the native sample rate of the respective system along with echo cancellers and filters in the audio path. We tested seven combinations of local and remote speakers and microphones: (1) local speaker and microphone with stock Android on Nexus 7, (2) local speaker and microphone with M2 idle on Nexus 7, (3) local microphone with Nexus 7 and remote speaker with another Nexus 7, (4) local microphone with Nexus 7 and remote speaker with Nexus 9, (5) local speaker with Nexus 7 and remote microphone with another Nexus 7, (6) local speaker with Nexus 7 and remote microphone with Nexus 4, and (7) Nexus 7 using remote speaker and microphone on another Nexus 7.

Figure 7 shows the audio latency measurements. For most of the tests, M2 adds negligible latency compared to stock Android, even for using remote microphones and speakers, indicate that M2's push model and device partitioning architecture provide good low latency performance. The one case in which M2 incurs higher performance overhead is when running the benchmark with both remote speaker and microphone at the 44.1 KHz sample rate and 81.3 ms buffer size settings for Zoiper, resulting in roughly 100 ms of additional latency and almost 20% overhead.

To measure audio performance in terms of audio streaming along with using other remote devices, we used seven Android systems together in a multi-mobile setup with a Nexus 9 client running the apps, a Nexus 4 providing remote sensor and touchscreen input, three other Nexus 9s and the Nexus 9 client in a 2x2 configuration for a combined larger display, and two Nexus 7s providing remote speakers with separate left and right audio channels, respectively, for stereo output across two devices. Remote display used full 1536x2048 native display resolution with video encoding at 30 fps and a 10 Mbps bit rate, and remote audio was unencoded PCM. To stress the system, we ran ten Android apps from Google Play, nine of the most popular gaming apps along with the VLC movie player app for comparison purposes. The gaming apps and their respective Google Play top game chart ranking were Angry Birds (#38), Candy Crush Saga (#10), Candy Crush Soda (#3), Clash of Clans (#6), Crossy Road (#1), Jelly Jump (#5), Racing Fever (#20), Subway Surfers (#7), and Surgery Simulator (#12). Each game was played intensively for a minute, and the VLC movie player was used to play and skip around for a minute of Big Buck Bunny, the widely used open movie project.

M2's qualitative performance for all of the apps was indistinguishable from running on a Nexus 9 with stock Android Lollipop. Audio was clear with no drops, and display was smooth with no noticeable skipped frames or display degradation. Figure 8 shows the per device average bandwidth consumption for running

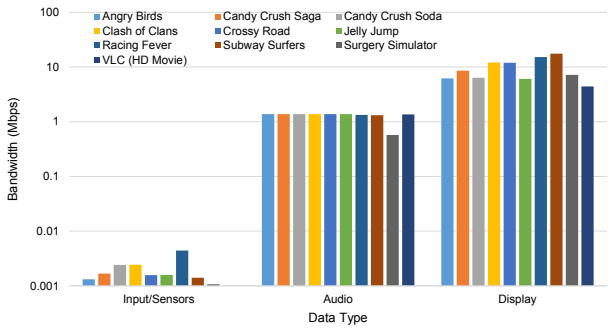

Figure 8: 3D games on seven Nexus 4/7/9 using M2

the various apps. Input and sensor remoting requires only a few Kbps of bandwidth even for intensive gaming. Audio remoting required 1 Mbps of bandwidth for PCM raw data, though AAC encoding would reduce this further. Display remoting for gaming required the most average bandwidth per device, ranging from 6.3 Mbps for Candy Crush Soda to 17.6 Mbps for Subway Surfers. By comparison, VLC only required 4.4 Mbps.

7. Related Work

Perhaps the most closely related approach to M2 is Rio, a system for sharing I/O devices between mobile systems and lays out a broad vision of its potential benefits. Rio is restricted to device mirroring, so that for example, the camera from one system can be remotely accessed from another, but there is no support for multiplexing or using multiple devices simultaneously, such as splitting the display of content across multiple displays. Rio partitions the device stack at the kernel interface using traditional device files as the abstraction between client and server. The client where the device resides has the real device file, and the server has a virtual device file which essentially forwards device interactions to the client. This approach suffers from a number of crucial limitations as discussed in Section 5. For example, since this is done below the HAL at the level of vendor-specific devices, it only works for systems which use the same vendor-specific hardware devices, and in practice, only works when the device implementations are sufficiently open, often not the case with vertically integrated mobile systems. Operating across different hardware is problematic, and making Rio work for each different system even assuming homogeneous devices is a complex and enormous porting effort. As a result, Rio has only been demonstrated to work on a single system, the Galaxy Nexus, and even in that case, has no display device support and poor remote audio and camera performance. M2 takes a completely different approach that allows it to support device heterogeneity and a richer model of using and multiplexing multiple devices simultaneously.

Although Rio does not support display sharing, also referred to as screencasting, a number of other approaches have explored this in the context of desktop computing, including VNC [25], Microsoft's Remote Desktop Protocol (RDP) [20], GoToMyPC [6], THINC [5], X [30, 34], and the general emergence of Virtual Desktop Infrastructure (VDI) [19]. A number of these systems support remoting audio content as well. These approaches use various forms of display commands to transport display content over the network, but do best with non-graphics intensive workloads and are inadequate in providing good display performance for 2D and 3D graphics intensive content, if such content is viewable at all. Other approaches have considered remoting graphics by sending OpenGL commands over the wire [16], but require substantial network bandwidth and do not support the myriad of OpenGL extensions used in mobile systems. None of these approaches work effectively if at all for display sharing across tablets and smartphones. In contrast to these display command and graphics primitive approaches, more recently, newer versions of Apple's AirPlay [3] enable display mirroring from iOS mobile systems to AppleTV by taking advantage of H.264 encoding hardware available on those systems to simply encode and decode raw display frames, but AirPlay is proprietary, only works on Apple hardware, and does not provide display mirroring between tablets and smartphones. M2 takes a similar approach of using video encoding and decoding hardware to make it possible to efficiently enable display sharing across WiFi networks between mobile systems with excellent display quality even for 3D graphics-intensive workloads. Unlike previous approaches, M2 goes beyond display mirroring to enable using multiple display devices simultaneously and provides support for the broad range of heterogeneous devices other than display available on mobile systems.

Universal Plug and Play (UPnP) [31] is a standard of network protocols to allow systems to discover each other on the network and access services, increasingly used to stream media content from a server to a UPnP capable system such as an XBox 360. However, we are not aware of any UPnP solutions that operate between tablets and smartphones. UPnP focuses on network discovery and access of services and is complementary to M2. MediaTek's recently announced CrossMount [18], to become available in late 2015, builds on UPnP to connect systems wirelessly so that for example, you can be streaming video on a TV then switch to watching it on a tablet. No actual demonstrations or public technical details are available, so this appears at this point to be a proposed technical specification to be implemented. Based on available mock-ups and advertising, CrossMount at best provides some form of device mirroring.

Various approaches explore extended protocols typically associated with traditional cabling to connect systems to output devices. For example, Miracast [33] defines a protocol to connect a TV monitor to a device over WiFi. These approaches are typically limited to a particular class of device and do not support general device sharing across multiple mobile systems.

Other approaches have more recently been explored for using multiple mobile systems, such as Flux [32]. It migrates apps across Android systems to enable a user using an app on a smartphone to continue using it on another tablet. Unlike M2, Flux does not support combining multiple mobile systems together for use.

Storage and networking on Android are not comparable to other devices M2 implements since they do not have relative system services. Existing applications on Android allow for sharing storage at the application layer using protocols such as Server Message Block or Common Internet File System [8]. Ori [17], PodBase [24],

and Eyo [29] are storage systems that take advantage of multiple devices to share files across different devices. Cloud based services such as Dropbox [7] and iCloud [4] provide location transparency of files and allow backing up of files. These systems are mostly concerned with sharing files rather than sharing storage devices as CIFS or NFS. Sharing networking can be achieved through WiFi Direct, in the case of Android, or Personal Hotspots in iOS.

8. Conclusions

We have designed, implemented, and evaluated M2, a system for multi-mobile computing by enabling remote sharing of heterogeneous devices across multiple mobile systems. By observing differences in the tall device stack of mobile systems, we split device functionality between client and server at user-level across app frameworks and system services to provide device remoting across heterogeneous mobile hardware and software. We show how fused devices can transparently enable existing apps to become multi-mobile, and extend the Android app API to allow developers to create new multi-mobile apps. Our experimental results across multiple versions of Android running on heterogeneous hardware as well as using iOS remote devices show that M2 can deliver good remoting performance even for device-intensive apps such as 3D games. We show that M2 runs unmodified Android apps across multiple systems combined together into one more capable one, and can enable users to run Android apps on iOS systems. If accepted for publication, we will do a live demo of the system at the conference.

9. Acknowledgments

This work was supported in part by NSF grants CNS-1162447, CNS-1422909, and CCF-1162021.

References

[1] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A system solution for sharing i/o between mobile systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 259–272, Bretton Woods, NH, 2014.

[2] J. Andrus, A. Van't Hof, N. AlDuaij, C. Dall, N. Viennot, and J. Nieh. Cider: Native Execution of iOS Apps on Android. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 367–382, Salt Lake City, UT, Mar. 2014.

[3] Apple Inc. Apple - AirPlay Play content from iOS devices on Apple TV. `http://www.apple.com/airplay/`, 2014. Accessed: 12/07/2014.

[4] Apple Inc. iCloud. `https://www.icloud.com`, 2015. Accessed: 03/25/2015.

[5] R. A. Baratto, L. N. Kim, and J. Nieh. Thinc: A virtual display architecture for thin-client computing. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 277–290, New York, NY, USA, 2005. ACM.

[6] Citrix Systems, Inc. Remote Access — GoToMyPc. `http://www.gotomypc.com/remote-access/`, 2015. Accessed: 02/12/2015.

[7] Dropbox Inc. Dropbox. `https://www.dropbox.com`, 2015. Accessed: 03/24/2015.

[8] S. M. French. A New Network File System is Born: Comparison of SMB2, CIFS, and NFS. `https://www.kernel.org/doc/ols/2007/ols2007v1-pages-131-140.pdf`, 2007. Accessed: 03/21/2015.

[9] Google Inc. Android Low-Level System Architecture. `http://source.android.com/devices/`, 2014. Accessed: 12/05/2014.

[10] Google Inc. Audio. `https://source.android.com/devices/audio/index.html`, 2014. Accessed: 02/04/2015.

[11] Google Inc. Audio Terminology. `https://source.android.com/devices/audio/terminology.html`, 2014. Accessed: 02/04/2015.

[12] Google Inc. Chromecast. `http://www.google.com/chrome/devices/chromecast/`, 2014. Accessed: 12/07/2014.

[13] Google Inc. Graphics architecture. `https://source.android.com/devices/graphics/architecture.html`, 2014. Accessed: 01/29/2015.

[14] Google Inc. Google play. `http://play.google.com`, 2015. Accessed: 03/03/2015.

[15] Google Inc. Location APIs — Android Developers. `https://developer.android.com/google/play-services/location.html`, 2015. Accessed: 03/25/2015.

[16] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 693–702, San Antonio, Texas, 2002. ACM.

[17] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières. Replication, history, and grafting in the ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 151–166, New York, NY, USA, 2013. ACM.

[18] MediaTek Inc. MediaTek Introduces a New Convergence Standard for Cross-device Sharing with CrossMount. `http://mediatek.com/en/news-events/mediatek-news/mediatek-introduces-a-new-convergence-standard-for-cross-device-sharing-with-crossmount/`, Mar. 2015. Accessed: 03/10/2015.

[19] Microsoft, Corp. Virtual Desktop Infrastructure overview. `http://www.microsoft.com/en-us/server-cloud/products/virtual-desktop-infrastructure/default.aspx`, 2015. Accessed: 08/09/2014.

[20] Microsoft Corporation. Remote Desktop Protocol (Windows). `http://msdn.microsoft.com/en-us/library/aa383015.aspx`, 2014. Accessed: 12/07/2014.

[21] J. Nieh, S. J. Yang, and N. Novik. Measuring thin-client performance using slow-motion benchmarking. *ACM Transactions on Computer Systems (TOCS)*, 21(1):87–115, Feb. 2003.

[22] OpenSignal. Android Fragmentation Visualized. `http://www.opensignal.com/reports/2014/android-fragmentation/`, Aug. 2014. Accessed: 03/17/2015.

[23] PassMark Software, Inc. Passmark performancetest - android apps on google play. `https://play.google.com/store/apps/details?id=com.passmark.pt_mobile`, June 2013. Accessed: 03/10/2015.

[24] A. Post, J. Navarro, P. Kuznetsov, and P. Druschel. Autonomous storage management for personal devices with podbase. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 36–36, Berkeley, CA, USA, 2011. USENIX Association.

[25] T. Richardson. The RFB Protocol. `http://www.realvnc.com/docs/rfbproto.pdf`, Nov. 2010. Accessed: 12/07/2014.

[26] SC Magazine. 2013 mobile device survey. `http://www.scmagazine.com/2013-mobile-device-survey/slideshow/1222/`, 2013. Accessed: 03/26/2015.

[27] Securax LTD. Zoiper audio latency benchmark - android apps on google play. `https://play.google.com/store/apps/details?id=com.zoiper.audiolatency.app`, Nov. 2014. Accessed: 03/05/2015.

[28] B. Shneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition)*. Pearson Addison Wesley, 2004.

[29] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and F. Kaashoek. Eyo: Device-transparent personal storage. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[30] The Linux Information Project. An introduction to X by The Linux Information Project (LINFO). `http://www.linfo.org/x.html`, 2006. Accessed: 01/05/2015.

[31] UPnP Forum. UPnP Forum. `http://www.upnp.org/`, 2015. Accessed: 03/21/2015.

[32] A. Van't Hof, H. Jamjoom, J. Nieh, and D. Williams. Flux: Multi-Surface Computing in Android. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys 2015)*, Bordeaux, France, Apr. 2015.

[33] Wi-Fi Alliance. Wi-Fi CERTIFIED Miracast: Extending the Wi-Fi experience to seamless video display. `http://www.wi-fi.org/system/files/wp_Miracast_Industry_20120919.pdf`, Sept. 2012. Accessed: 12/20/2014.

[34] X.Org Foundation. X.Org. `http://www.x.org`, 2015. Accessed: 02/27/2015.

[35] Z. Zhang, D. Chu, X. Chen, and T. Moscibroda. Swordfight: Enabling a new class of phone-to-phone action games on commodity phones. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 1–14, New York, NY, USA, 2012. ACM.