Heterogeneous 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-008-16
August 2016

Abstract

As smartphones and tablets proliferate, there is a growing need to provide ways to combine multiple mobile systems into more capable ones, including using multiple hardware devices such as cameras, displays, speakers, microphones, sensors, and input. We call this multi-mobile computing. However, the tremendous device, hardware, and software heterogeneity of mobile systems makes this difficult in practice. We present M2, a system for multi-mobile computing across heterogeneous mobile systems that enable new ways of sharing and combining multiple devices. M2 leverages higher-level device abstractions and encoding and decoding hardware in mobile systems to define a client-server device stack that shares devices seamlessly across heterogeneous systems. M2 introduces device transformation, a new technique to mix and match heterogeneous input and output device data including rich media content. Example device transformations for transparently making existing unmodified apps multi-mobile include fused devices, which combine multiple devices into a more capable one, and translated devices, which can substitute use of one type of device for another. We have implemented an M2 prototype on Android that operates across heterogeneous hardware and software, including multiple versions of Android and iOS devices, the latter allowing iOS users to also run Android apps. Our results using unmodified apps from Google Play show that M2 can enable apps to be combined in new ways, and can run device-intensive apps 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 [23], and groups of users often have many mobile systems at their disposal. 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.

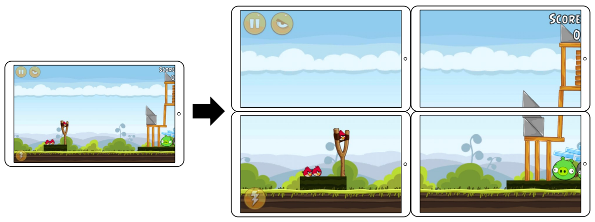

Figure 1: Multi-mobile computing using fused displays/input

Three simple examples help to illustrate just a few of the possibilities of multi-mobile computing. First, remote control of other devices such as cameras is often useful, but each app developer today is forced to reinvent the wheel each time to provide such functionality across systems. Multi-mobile computing makes it straightforward to have an app on one system control devices on another without additional implementation complexity. Second, rather than being limited to using a touchscreen for input to control a game, multi-mobile computing would enable smartphones to be used as Wii-like game controllers for a game running on another system to provide a richer gaming experience. Third, 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 combine their tablets together in a grid to provide a unified display and input surface across all the tablets, for a better viewing experience for all the users. Unlike simple one-to-one mirroring approaches such as AirPlay [2] 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 in new ways.

Although multi-mobile computing has the potential to provide a wide range of powerful new app functionality, two key challenges must be met to turn this potential into reality. First, smartphones and tablets are highly heterogeneous; on Android alone, more than 20,000 different systems available [19]. They are tightly integrated hardware platforms that incorporate a plethora of different hardware devices using non-standard interfaces. Many different versions of software run on these systems, including many versions of iOS and Android, especially the latter given the fragmentation of the Android market. This level of device, hardware, and software heterogeneity makes it difficult to combine multiple devices together across mobile systems while ensuring good performance. Second, smartphone and tablet devices consume and produce a wide range of disparate input and output data via heterogeneous interfaces, from a variety of sensor readings to rich audio and video media content. This level of data heterogeneity makes it difficult to combine and share smartphone and tablet devices so that different types of devices can be redirected, mixed, and matched together across mobile systems.

The lack of system support for combining multiple devices across heterogeneous mobile systems together forces each app developer who would like to provide such functionality to start from scratch, making such development 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.

To address these problems, we introduce M2, a system for multi-mobile computing that redirects and transforms heterogeneous device input and output across heterogeneous mobile systems to enable new ways of sharing and combining multiple devices. To solve the device heterogeneity problem, 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, allowing apps to make use of remote devices. 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.

To mix and match heterogeneous devices across different mobile systems, M2 introduces device transformation, a framework that enables disparate devices across different systems to be substituted and combined with one another. For example, rather than using the local camera for input, device transformation allows an app to use any remote video device for input, including a remote camera or remote display output. To enable unmodified apps to become multi-mobile, M2 introduces two types of device transformations: *fused devices* and *translated devices*.

Fused devices 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.

Translated devices allow devices to be reinterpreted in different but contextually meaningful ways. For example, a translated device could transform sensor data from the accelerometer into input touches, so a hand gesture can be considered as a left or right input swipe. M2 can thereby transform a smartphone into a Nintendo Wii-like remote and enjoy interactive, unmodified games on a nearby tablet.

M2 leverages higher-level device abstractions and widely deployed mobile system hardware features to optimize the transfer of device data 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 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 software and hardware systems, including iOS 8.2, iOS 9.3.1, and four recent and widely used major Android versions, Marshmallow, Lollipop, KitKat and Jelly Bean running on different tablet and smartphone systems. 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. Usage Model

M2 is designed to be simple to use. A mobile system is a *device server* if it has a device that is being shared with other systems, and is a *device 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, an M2 Android app from Google Play or an M2 iOS app from the Apple App Store. 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. Devices not listed in a profile are not shared. Device profiles can be disabled at any time. For simplicity, only one device profile can be enabled at a time on a given device server.

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 touch-screen 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 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 dumb peripheral, and any other Android app, treating the system as a full-fledged Android computer.

To run apps that access remote devices on other mobile systems, the M2 native frameworks must be installed on the device client. Once installed, a user can make remote servers accessible by downloading and running the M2 app on the client. The M2 app enables the user to see all available peers on the network currently offering to share devices. Multicast DNS (mDNS) [9] is used to facilitate peer discovery. 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. The M2 app shows both currently active device profiles as well as previously accessed device profiles that are not currently active, the latter to make them easy to access again in the future. Accessing device profiles can also be done by other apps in a programmatic fashion.

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 increase security risks with remote device access compared to the security 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. 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 [34], 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. 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.

Usage profiles can specify server devices directly or server devices via device transformation. Two common device transformations that M2 provides to support unmodified apps are *fused devices* and *translated devices*. Fused devices provide a single device abstraction based on fusing information from multiple devices, allowing existing apps designed for use with one device to see multiple devices as one. The idea is similar to fused location providers in Android [7], which combine GPS, WiFi, and cellular 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. Similarly, a fused camera device

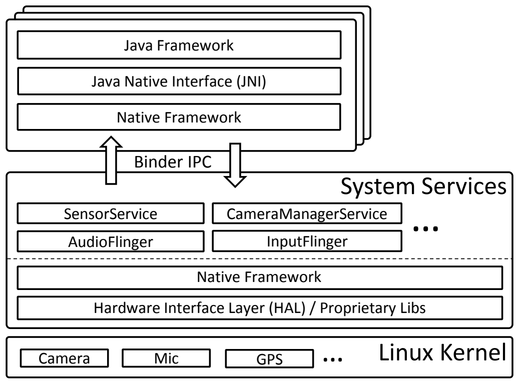

Figure 2: Android architecture

could be defined to combine the previews of multiple cameras together by tiling them in a single preview display.

Translated devices enable one type of device to be used as another, allowing existing apps designed for use with one type of device to use another as a substitute. For example, a translated accelerometer sensor to input would use linear acceleration as input to another device. Hand gestures would be interpreted as either input touches or swipes resulting in a Nintendo Wii-like experience. A similar example is to use visual input cues from the camera by translating the camera data into actual input touches.

3. M2 Architecture

The M2 architecture addresses three key issues to support multi-mobile computing. The first is how should device functionality be partitioned across systems to support the device, hardware, and software heterogeneity in mobile systems. The second is ensuring good performance even for using high-bandwidth devices across the network. The third is how to enable disparate devices across different systems with heterogeneous data formats to be mixed and matched.

3.1 Client-Server Device Stack

To understand how to partition device functionality across mobile systems in such a way that supports heterogeneous devices, hardware, and software, it is helpful to 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 call Java frameworks, which function as libraries that provide the core public APIs used by developers for 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.

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. Each type of device has an associated system service which provides its own specialized abstraction. Table 1 lists the major types of user-facing I/O devices supported in Android and the respective device abstractions used. For example, the SensorService manages sensor devices, AudioFlinger manages audio devices, and the CameraService 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.

Given this background regarding the Android device infrastructure, 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 forwards device interactions to the server. However, this approach will not work for heterogeneous systems for at least three reasons. First, since this is done below the HAL, it only works for systems which use the same vendor-specific hardware devices and any attempt to operate across heterogeneous hardware is problematic. Second, common device-level operations on mobile systems involve vendor-specific `ioctls` with variable inputs and possibly memory references which are difficult to identify and reproduce correctly across different systems. Finally, substantial device functionality may be implemented inside proprietary vendor libraries rather than the driver itself and these libraries may manipulate device state via shared memory in any opaque manner such that simply forwarding device files calls is insufficient to replicate device state.

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 problems supporting heterogeneous systems. First, because the HAL layer is low-level and Android dependent, providing device servers at the HAL layer would require firmware 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. Second, it may be important for performance and heterogeneity to be able to process device data before sending it over the network, but this is an issue at the HAL layer. Furthermore, since the HAL layer is low-level, higher-level abstractions provided by Android are unavailable, making it problematic to leverage higher-level semantics for processing data. Finally, system services expect to manage underlying devices on behalf of Android apps running on the server, so avoiding device conflicts with unmodified system services becomes problematic as they would be unaware of the devices being used by a remoting mechanism at the lower HAL layer.

device	abstraction	system service
sensor	event type	SensorService
input	source type	InputFlinger
location	provider name	LocationManagerService
mic	audiosource type	AudioFlinger
camera	camera id	CameraService
audio	implicit	AudioFlinger
display	surface name	SurfaceFlinger

Table 1: Android device abstractions

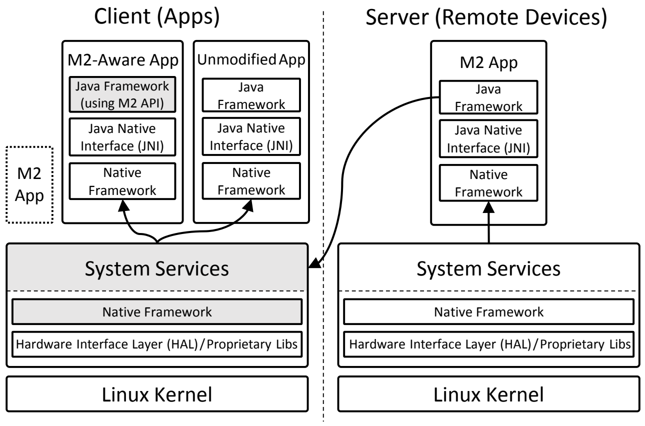

Figure 3: M2 architecture; Android modifications in grey

A third approach would be to partition at the IPC interface to system services by forwarding IPC calls remotely, but this will not work for heterogeneous systems. First, forwarding requires the exact same IPC interfaces to system services on both systems, but these interfaces vary between Android versions making interoperability problematic across Android versions not to mention non-Android systems. Second, IPC interfaces do not encapsulate all necessary communication between apps and devices. Device data may be exchanged via shared memory, the file system, or Unix domain sockets as opposed to IPC callback functions. For example, apps write display data directly to surfaces backed by shared graphical memory, sensor data is sent to apps via Unix domain sockets, and apps can request that the CameraService write pictures directly to disk on their behalf.

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 to address the problem of device, hardware, and software heterogeneity. 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, for those similar APIs, they are likely to remain relatively stable across different Android versions.

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 accessing local devices, system services and their supporting native frameworks are modified to also access remote devices. Figure 3 shows an overview of the M2 architecture in Android, which supports new multi-mobile apps using an M2 API and existing unmodified apps.

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. As demonstrated by our support of iOS and given that servers are built on public APIs typical of most mobile ecosystems, we expect that building servers for other non-Android systems is relatively straightforward.

On the client, because device access is implemented within system services, remote devices are made available to apps through the existing system service interfaces used for local devices. This makes it easy for apps to use remote devices in the same manner as they use local devices. By implementing remote devices at the system service layer, 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 leverages those same frameworks to reduce implementation complexity rather than having to reimplement low-level interfaces such as the HAL. These native frameworks often provide similar functionality to the Java frameworks that provide the public API for apps. This makes it easy to map from one to the other, often 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 four recent and widely used major Android versions, Marshmallow, Lollipop, KitKat, and Jelly Bean.

By relying on the decoupling of apps from devices provided by system services, M2 treats remote devices as dumb peripherals. 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.

3.2 Data and Network Communication

M2 clients and servers communicate over standard network sockets and are designed to interact over WiFi and WiFi Direct networks. For control messages and 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 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 same timestamps are combined with NTP and best practices to ensure media synchronization across devices [12, 17, 22, 24]. The microphone is an interesting case as it can operate in both ways where TCP is used for recording and UDP for streaming. Note that we currently do not use UDP multicast with multiple remote devices as existing 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. An alternative is MicroCast [10] which uses pseudo-broadcast, but it requires changes to WiFi drivers/firmware and the use of custom Android firmware.

To optimize network performance and security, M2 leverages common hardware features of mobile systems to provide security and high performance. M2 provides secure client-server communications using 128-bit AES encryption, leveraging widespread adoption of AES hardware acceleration support in mobile devices. M2 leverages video and audio encoding and decoding hardware available on mobile systems to efficiently support high-bandwidth devices. M2 encrypts data with separate session keys for each device as opposed to separate session keys for every client system. This way, device data is only encrypted once regardless of the number of clients, but still prevents, for example, one system with access to a remote sensor device from accessing display data being shared with a different system. Should the user alter the mobile systems allowed to access a device, that device session key is regenerated and retransmitted to all clients.

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 outputted. 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 [27]. 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. Since hardware is needed for real-time encoding, each system can encode and decode a limited number of data streams. Given the limited number of encoder/decoder streams supported, M2 encodes the complete data and transmits the same encoded data to all remote devices even if each device only uses a portion of the data, such as when multiple displays are used as one and each device only displays a portion of the data. Although this uses some additional bandwidth, it saves on the number of encoders used at the client. The display data can then be scaled and resized appropriately for viewing at the remote display based on the hardware characteristics of the respective screen.

3.3 Device Transformations

M2 provides a straightforward extension of the Android APIs for sensors, input, location, microphone, camera, audio and display devices to support new multi-mobile apps that desire explicit control of multiple remote and local devices. The API makes multiple devices available so that apps can programmatically access and combine them. However, existing apps are generally not designed to use multiple devices of a given type, so the M2 extension API cannot be used by them unless they are modified. Given the large number of existing apps that can benefit from multi-mobile computing, M2 introduces a device transformation framework that makes it possible for existing apps to transparently use mixes of local and remote devices, making them multi-mobile without modification. For example, M2 supports fused devices, a device transformation that combines multiple devices of the same type together into one. An existing app built to use one display device can instead use a fused display device to be able to display to multiple displays instead of just one.

A transformation consists of an input device abstraction, an output device abstraction, and a transformation function. The device abstraction includes its type and data format. The trans-

device	identifiers, data types, and formats
sensor	sensor type, data fields (x, y, z)
input	gesture (e.g. touch vs swipe), coords. (x, y)
location	location type, coords. (x, y, z), aux data (e.g., speed)
mic	media format (e.g. AAC), channels, sample rate
camera	functionality (pic vs. video), media format, resolution
audio	media format (e.g. AAC), channels, bit/sample rate
display	media format (e.g. H.264), resolution, bit rate, fps, position, scaling

Table 2: M2 formats and types per hardware device

formation function is used to convert the input device abstraction to the output device abstraction, and can operate on device data or control information. Table 2 provides a summary of formats and types for each hardware device. The definition of these types and formats under M2 is essential in order to provide a portable translation medium across different Android versions, hardware devices, and platforms (e.g., iOS to Android and vice versa).

To support fused devices, translated devices, and other custom transformations, M2 provides a transformation plugin framework that operates in conjunction with the M2 app on either a device server or client. Plugins provide a way for developers to write their own transformations, which can be integrated into M2. In Android, a plugin is a standalone Android Application Package (APK) that can be downloaded through the Google Play Store. The M2 app exposes an RPC interface implemented via the Android Interface Definition Language (AIDL) that allows these plugins to register with the M2 app to receive device control and data information from desired device(s). The plugin can then transform this data and return the result to the M2 app as a new output device abstraction to be directed to local or remote device(s) or exposed as a new shareable device. With regard to input device abstractions, plugins have access to devices whenever the M2 app would have access to them. On the device server, where only the app runs, plugins can have access to most devices at any time, but input only when the app is in the foreground. On the device client, plugins can have access to devices at any time by leveraging their modified frameworks for background access. Client-side transformation output is passed back to system services via the loopback network interface, appearing to system services in the same manner as remote devices. In iOS, transformations are currently implemented within the M2 app, but we envision using iOS action app extensions [3] to support plugins for iOS.

This framework provides three key benefits. First, by allowing plugins to operate on the server or client, it provides maximum flexibility to support a wide range of transformation functionality. Second, by leveraging standard plugin functionality via APKs, it makes it possible for developers to make use of higher-level application interfaces and semantics to ease programming of transformation functionality. Finally, since plugins communicate with devices via the M2 app using the same mechanisms that support remote device server functionality, they are similarly isolated. The risk of misbehaving plugins is therefore limited to only the apps that are actively using them as specified in the user profile, while mitigating the risk to the system as a whole.

To illustrate how the framework can be used, we describe a few example device transformations. One example is a fused display device, illustrated in Figure 1, in which four tablets are combined in a 2x2 matrix to provide a larger display and input surface. The app is running on one tablet and displaying content on four M2 provided remote display devices, with the local client accessing its "remote" display device through the loopback network interface. Additionally, the app is receiving touch input from the four touchscreen devices exposed by M2. To provide fused display, each device runs a plugin with a display input device abstraction and a display output abstraction. The plugins use the display ID information, which identifies each display's position in the 2x2 grid, to adjust the output abstraction such that it is scaled to be four times as large and positioned relative to which quarter needs to appear. Since the plugin only manipulates control information while leaving the data processing to system services and the underlying hardware, the display fusion is fast and efficient. Note that in this scenario the local client is simultaneously providing display output while its screen part of the fused display. To achieve this, M2 makes a small modification to the client's SurfaceFlinger to allow for recording only those surface layers below the M2 app.

To provide fused input, M2 runs a plugin on just the device client where the app runs. The plugin registers to receive input device abstraction data from the four remote input devices and scales the coordinates of each input device so that each touchscreen appears to only cover a quarter of the input surface. This is done based on the display ID information to determine each touchscreen's position in the 2x2 grid. It then combines the input data into a single fused input device which is seen by the app.

As another example, consider a translated input device in which accelerometer sensor data is instead translated into touchscreen input to provide a Wii-like remote controller in place of normal touchscreen input for an app. Providing this translated device can be accomplished using either a server or client plugin, but for brevity, we just discuss the server plugin option. M2 runs a plugin on the device server which converts sensor data into different touchscreen inputs. For example, accelerometer data showing a left-to-right movement of the device is translated into a left-to-right touchscreen swipe. The output device abstraction from the device server is a touchscreen input device, which is sent to the device client where a tennis or swordfight app can run to take advantage of the more natural Wii-like gaming experience. The device client simply sees a remote input device and does not need to know that the input was translated from an actual accelerometer device.

4. 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 app for iOS systems, demonstrating the ability to access remote devices across Android and iOS. We first describe some ways in which we have used M2, then present some quantitative performance measurements.

We ran M2 across four different major versions of Android and two different major versions of iOS on five different smartphones and tablets, namely Nexus 4 (768x1280 display, Qualcomm Snapdragon S4 Pro 1.5GHz quad-core CPU) smartphones with Android 4.3 Jelly Bean and Android 4.4 KitKat, Nexus 7 (1200x1920 display, Qualcomm Snapdragon S4 Pro 1.5GHz quad-core CPU) tablets with Android 6.0 Marshmallow, Nexus 9 (1536x2048 display, Nvidia Tegra K1 2.3GHz dual-core CPU) tablets with Android 5.0 Lollipop, a 1st generation iPad mini (768x1024 display, Apple A5 1GHz dual-core CPU) tablet with iOS 8.2, and an iPhone 6S (750x1334 display, Apple A9 1.85GHz dual-core CPU) running iOS 9.3.1. 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 and iPhone 6S support and use IEEE 802.11ac, while the other systems use IEEE 802.11n.

4.1 Example Use Cases

We first downloaded and installed various unmodified apps from Google Play to use with M2 in four different configurations. First, we simply made iOS remote devices available to Android apps, allowing an iPhone 6S with the latest iOS to effectively run unmodified iOS and Android apps from the same system for the first time.

Second, we used fused devices for display and input to allow four systems in a 2x2 layout to be combined as one. Display and input are split across all screens providing a larger multi-headed display experience. Because displaying across multiple devices can be bandwidth-intensive and therefore is a useful stress test of the system, we used this M2 configuration for many of our performance measurements in Section 4.2. Section 4.2 also lists ten display-intensive apps that we used with a fused display and input device configuration.

Third, we used a translated device from accelerometer sensor data to input touches to provide a Wii-like experience for various unmodified Android games. We map five movements based on accelerometer changes, right to left, left to right, up, down, and forward, to five respective touchscreen gestures, swipe left, swipe right, swipe up, swipe down, and swiping in a Z shape. The mappings were chosen to match various first-person Android game experiences. We used this translated device configuration for two Android games, Epic Swords 2 and 3D Tennis. Our experience using a smartphone to control the game running on a tablet showed that M2 provided a much more realistic gaming experience than the native game used with touchscreen input. The Epic Swords 2 game feels more realistic swinging a smartphone to control sword movements during a swordfight than swiping across the touchscreen. This is especially true for stabbing with the sword, which corresponds to a realistic stabbing movement by moving the smartphone forward and back with M2 but requires an unnatural Z swipe in the case of using the touchscreen due to the practical limitations of what a touchscreen can do.

The 3D Tennis game also feels more realistic swinging a smartphone to control the tennis racquet while playing tennis than swiping across the touchscreen. For example, swinging the smartphone from right to left is similar to a tennis forehand stroke, unlike a confusing left swipe on a touchscreen. The same is true for swinging from left to right for a backhand, swinging down to hit a serve, swinging up to add topspin to a stroke, and swinging down to slice. Because it is designed for use with a touchscreen, a limitation of the 3D Tennis game is that it is not possible to both control the stroke and spin of the tennis ball, and the speed of the swing has no impact on the speed of the ball. We did not have access to the game's source code, otherwise modifying the game directly to use the M2 API would provide an even better multi-mobile gaming experience. This example illustrates both the useful ways in which M2 can make existing apps multi-mobile as well as the limits of what can be done without any app modifications.

Fourth, we used translated devices to record movies playing on another system. Instead of having the audio and video data go to remote audio and display devices, we mapped the audio to go to the remote microphone and the video to go to the remote camera device. In this scenario, both systems act as M2 device clients. This effectively allows an existing unmodified camera app on the remote system to view and record the audio and video output of another system. The transformation plugin we used for video ignores the secure flag for display surfaces so that it can be viewed on non-secure displays and later recorded. We used this configuration to play a movie on one Android system using the Netflix app, and record it on another using the stock Android camera app. We also recorded a video being played by the YouTube app in the same manner. The recordings were high quality with audio and video well synchronized, and were playable using the Android stock video player, allowing the user to play them at a later time without needing to be online.

4.2 Performance Measurements

We next ran benchmarks and unmodified Android apps from Google Play [6] to quantify M2 performance. 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 [20]. PassMark conducts a wide range of resource intensive tests to evaluate CPU, memory, I/O, and graphics performance. If display servers drop frames, the reported benchmark results may not reflect the performance perceived at the servers since the app does not account for this in reporting benchmark performance. To account for this difference, we scale the performance results based on the percentage of frames displayed at the server, similar to metrics from slow-motion benchmarking [18]. For example, if only half of the frames are displayed by the server, then the benchmark measurement reported by the app is reduced by half.

We ran M2 with PassMark in seven system configurations using a Nexus 9 (N9) to run the app: (1) stock Android Lollipop, (2) M2 installed but idle, (3) M2 displaying locally on the same system, (4) using two N9 systems, splitting the display across the N9 client and another N9 display server, (5) using four N9 systems in a 2x2 configuration combined as one display, splitting the display across the N9 client and three other N9 display servers, (6) using a mix of four heterogeneous systems, one-to-many mirroring the N9 display to a N9, a Nexus 7 (N7), and a Nexus 4 (N4), and (7) using a mix of four heterogeneous systems, one-to-many mirroring the N9 display

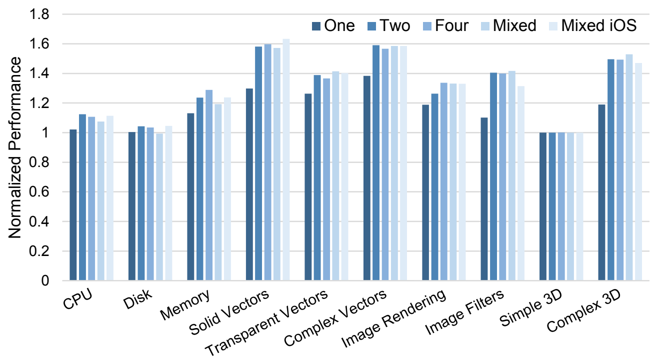

Figure 4: PassMark performance; lower is better

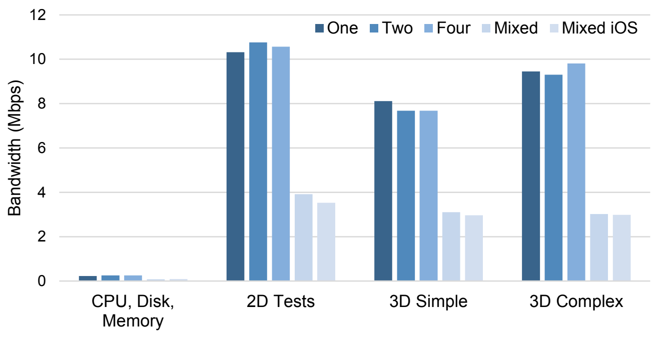

Figure 5: PassMark per device bandwidth

to a N7, a N4, and an iPad mini. We used the full 1536x2048 native display resolution for all N9 experiments; display encoding was done at a variable fps limited to 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 N7 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. 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, 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. In all cases, the network was not a performance bottleneck and dropping frames or packets was not an issue. In fact, we also ran the Mixed iOS scenario with a dozen other tablets and laptops connected to the WiFi router playing YouTube HD videos, and the performance was the same as shown in Figure 4. In comparing the mixed display and homogeneous display measurements, both using four Android systems, the performance is similar even though the homogeneous case uses a much higher resolution and bit rate, showing that M2 scales well with increasing devices and higher video quality. When using multiple displays, the display quality across the devices appeared qualitatively the same. The lone device case shows only slightly better performance since it does not send packets out to the network. This is more apparent with the solid vectors, image filters, complex 3D tests which use more bandwidth due to the higher number of changes and therefore, encoded frames.

Performance for the remote display devices was visually indistinguishable from stock Android, but quantitatively shows a range of performance overhead from less than 1% for the 3D simple test to around 60% for the 2D solid vectors test. This is the worst case quantitative performance due to the extra encoding, decoding, encryption, and other steps involved. PassMark is designed to stress test the system, so its quantitatively performance is a conservative measure of real app performance.

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 .3 Mbps as the only display updates are for a progress bar for each test and the display of the test results. For the 2D tests, the bandwidth required was up to a little over 10 Mbps for the 1536x2048 remote display tests and up to 4 Mbps for the 720x1280 remote display tests. 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 N9, N7, and N4, and compared to four different remote camera scenarios, the N7 using a remote N4 camera, the N7 using a remote N9 camera, the N4 using a remote N9 camera, and the N4 using a remote N7 camera. The first two remoting scenarios illustrate using a higher quality remote camera to take pictures as both the N4 and N9 cameras are higher quality than the N7. The second two remoting scenarios illustrate using a small form factor system, the N4, to control cameras on larger form factor tablets, the N9 and N7.

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

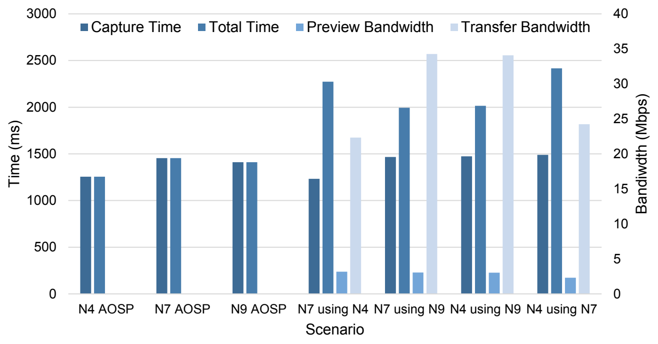

Figure 6: Camera latency for taking/storing pictures

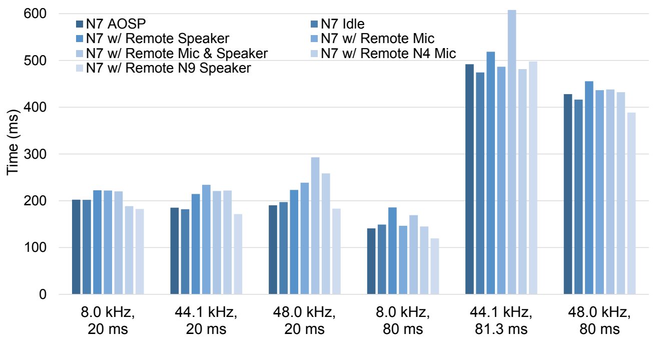

Figure 7: Zoiper audio latency

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 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 N4 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 N7 using the N4 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 N9 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 N9.

We next focus on measuring audio and microphone latency performance. We used the Zoiper [25] 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 N7, (2) local speaker and microphone with M2 idle on N7, (3) local microphone with N7 and remote speaker with another N7, (4) local microphone with N7 and remote speaker with N9, (5) local speaker with N7 and remote microphone with another N7, (6) local speaker with N7 and remote microphone with N4, and (7) N7 using remote speaker and microphone on another N7.

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 N9 client running the apps, a N4 providing remote sensor and touchscreen input, three other N9s and the N9 client in a 2x2 configuration for a combined larger display, and two N7s 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 a variable 60 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 [31] 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 N9 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 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.

5. Related Work

Some aspects of remote device sharing have been previously explored, both for mobile and desktop computing, but not across the broad range of heterogeneous devices and systems supported by

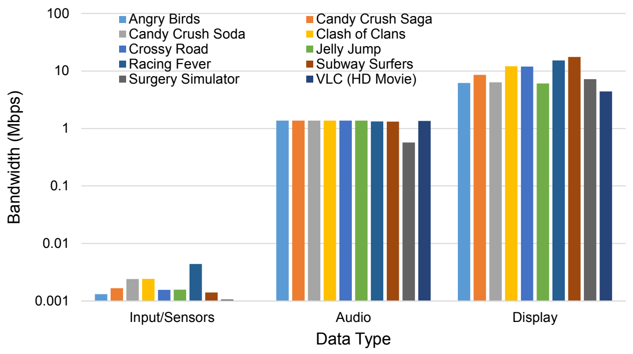

Figure 8: 3D games on seven Nexus 4/7/9 using M2

M2. Rio [1] provides device mirroring for homogeneous Android systems, so that for example, the camera from one system can be remotely accessed from another. However, Rio partitions the device stack at the kernel interface using traditional device files as the abstraction between client and server, making it not workable for heterogeneous mobile systems as discussed in Section 3. As a result, Rio only works across Galaxy Nexus phones with the exact same version of Android, and even in that case, has no display device support and poor remote audio and camera performance. This is problematic in practice given the heterogeneity of the Android market. Unlike M2, Rio does not support multiplexing or using multiple devices simultaneously.

Although Rio does not support display sharing, a number of other approaches have explored this in the context of desktop computing, including VNC [21], Microsoft's Remote Desktop Protocol (RDP) [16], GoToMyPC [5], THINC [4], X [28, 33], and the general emergence of Virtual Desktop Infrastructure (VDI) [15]. Some of these systems support remoting audio content as well, but none of them support the broad range of devices available on mobile systems. 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 [8], 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.

Newer versions of Apple's AirPlay [2] 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. Some apps such as MobiUS [26] share a display, in the case of MobiUS by splitting a video across mobile devices by using software decoding on both devices. MobiUS only works with two mobile devices, has weak performance due to software decoding, supports video files only, and is restricted to display sharing only within the app. 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) [29] is a standard of network protocols used to stream media content from a server to a UPnP capable system such as an Xbox 360. 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 previously announced CrossMount [13, 14], which 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, and the latest documentation still lists it as becoming available in late 2015, which has past.

Various approaches explore extended protocols typically associated with traditional cabling to connect systems to output devices. For example, Miracast [32] 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 recently been explored for using multiple mobile systems, such as Flux [30]. 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.

Various cloud-based approaches have also been used to aggregate some device functionality across multiple mobile systems, for example by creating new apps that obtain sensor data from multiple systems to predict earthquakes [11]. M2 provides multi-mobile functionality locally across the broad range of devices available on mobile systems, including those that are highly data-intensive, without additional cloud infrastructure.

6. Conclusions

We have designed, implemented, and evaluated M2, the first system for multi-mobile computing across heterogeneous mobile systems. By observing how mobile systems use higher-level abstractions and taller interfaces, M2 splits device functionality between client and server at user-level across app frameworks and system services to share devices remotely across heterogeneous mobile hardware and software. By leveraging widely available mobile encoding and encryption hardware, M2 provides encrypted device remoting across WiFi networks even for data-intensive devices with good performance. By introducing device transformations such as fused and translated devices, M2 transparently enables existing apps to share, redirect, and combine devices. Our experimental results across multiple versions of Android and iOS with heterogeneous hardware show that M2 enables new multi-mobile functionality for existing apps such as multi-headed displays, Wii-like gaming, and running Android apps on iOS systems, incurs only modest overhead, and provides qualitative performance similar to local device hardware even for 3D games.

7. Acknowledgments

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

References

[1] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin 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] Apple Inc. Apple - AirPlay Play content from iOS devices on Apple TV. `http://www.apple.com/airplay/`, 2014.

[3] Apple, Inc. App Extension Programming Guide: App Extensions Increase Your Impact. `https://developer.apple.com/library/ios/documentation/General/Conceptual/ExtensibilityPG/`, March 2016.

[4] Ricardo A. Baratto, Leonard N. Kim, and Jason 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.

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

[6] Google Inc. Google play. `http://play.google.com`, 2015.

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

[8] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James 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.

[9] Internet Engineering Task Force (IETF). RFC 6762 - Multicast DNS. `https://tools.ietf.org/html/rfc6762`, February 2013.

[10] Keller, Lorenzo and Le, Anh and Cici, Blerim and Seferoglu, Hulya and Fragouli, Christina and Markopoulou, Athina. MicroCast: Cooperative Video Streaming on Smartphones. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, Low Wood Bay, Lake District, UK, June 2012.

[11] Qingkai Kong, Richard M. Allen, Louis Schreier, and Young-Woo Kwon. Myshake: A smartphone seismic network for earthquake early warning and beyond. *Science Advances*, 2016.

[12] Alexander Löffler, Luciano Pica, Hilko Hoffmann, and Philipp Slusallek. Networked displays for VR applications: Display as a Service (DaaS). In *Virtual Environments 2012: Proceedings of Joint Virtual Reality Conference of ICAT, EuroVR and EGVE (JVRC)*, Oct 2012.

[13] 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/`, March 2015.

[14] MediaTek Inc. Unite your devices: Open up new possibilities. `http://www.mediatek.com/en/features/crossmount/`, October 2015.

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

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

[17] Sungwon Nam, Sachin Deshpande, Venkatram Vishwanath, Byungil Jeong, Luc Renambot, and Jason Leigh. Multi-application inter-tile synchronization on ultra-high-resolution display walls. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10. ACM, 2010.

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

[19] OpenSignal. Android Fragmentation Visualized. `http://www.opensignal.com/reports/2014/android-fragmentation/`, August 2014.

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

[21] Tristan Richardson. The RFB Protocol. `http://www.realvnc.com/docs/rfbproto.pdf`, November 2010.

[22] Kay Römer. Time synchronization in ad hoc networks. In *Proceedings of the 2nd ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '01. ACM, 2001.

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

[24] A. Schmitz, M. Li, V. Schnefeld, and L. Kobbelt. Ad-hoc multi-displays for mobile interactive applications. 2010.

[25] Securax LTD. Zoiper audio latency benchmark - android apps on google play. `https://play.google.com/store/apps/details?id=com.zoiper.audiolatency.app`, November 2014.

[26] Guobin Shen, Yanlin Li, and Yongguang Zhang. 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, New York, NY, USA, 2007. ACM.

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

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

[29] UPnP Forum. UPnP Forum. `http://www.upnp.org/`, 2015.

[30] Alexander Van't Hof, Hani Jamjoom, Jason Nieh, and Dan Williams. Flux: Multi-Surface Computing in Android. In *Proceedings of the*

10th European Conference on Computer Systems (EuroSys 2015), Bordeaux, France, April 2015.

[31] VideoLAN Organization. VideoLAN - Official page for VLC media player. `https://www.videolan.org`, 2016.

[32] 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`, September 2012.

[33] X.Org Foundation. X.Org. `http://www.x.org`, 2015.

[34] Zengbin Zhang, David Chu, Xiaomeng Chen, and Thomas 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.