Heterogeneous Multi-Mobile Computing

Naser AlDuaij
Department of Computer Science
Columbia University
New York, New York, USA
alduaij@cs.columbia.edu

Alexander Van't Hof
Department of Computer Science
Columbia University
New York, New York, USA
alexvh@cs.columbia.edu

Jason Nieh
Department of Computer Science
Columbia University
New York, New York, USA
nieh@cs.columbia.edu

ABSTRACT

As smartphones and tablets proliferate, there is a growing demand for multi-mobile computing, the ability to combine multiple mobile systems into more capable ones. We present M2, a system for multi-mobile computing that enables existing unmodified mobile apps to share and combine multiple devices, including cameras, displays, speakers, microphones, sensors, GPS, and input. M2 introduces a new data-centric approach that leverages higher-level device abstractions and hardware acceleration to efficiently share device data, not API calls. To support heterogeneous devices, M2 introduces device transformation, a new technique to mix and match different types of devices. Example transformations include combining multiple displays into a single larger display for better viewing, or substituting accelerometer for touchscreen input to provide a Nintendo Wii-like experience with existing mobile gaming apps. We have implemented M2 and show that it (1) operates across heterogeneous systems, including multiple versions of Android and iOS, (2) can enable unmodified Android apps to use multiple mobile devices in new and powerful ways, including supporting users with disabilities and better audio conferencing, and (3) can run apps across mobile systems with modest overhead and qualitative performance indistinguishable from using local device hardware.

CCS CONCEPTS

• **Human-centered computing** → **Mobile computing**; **Ubiquitous and mobile computing systems and tools**; **Mobile devices**; • **Software and its engineering** → *Client-server architectures*; Operating systems; Peer-to-peer architectures.

KEYWORDS

Mobile computing; distributed computing; operating systems; mobile devices; remote display; Android; iOS

ACM Reference Format:
Naser AlDuaij, Alexander Van't Hof, and Jason Nieh. 2019. Heterogeneous Multi-Mobile Computing. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307334.3326096

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326096

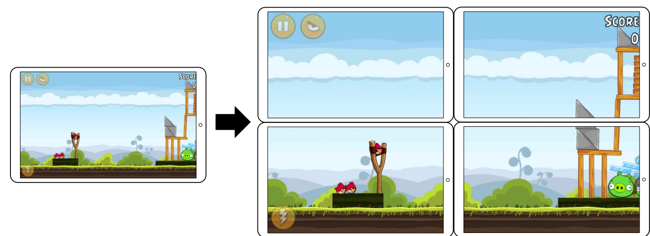

Figure 1: Multi-mobile computing using fused devices

1 INTRODUCTION

Users rely on tablets and smartphones for their everyday computing needs. Individual users often own multiple mobile systems of various shapes and sizes [79], and groups of users often have many mobile systems at their disposal. Since the vision of dynamic composable computing [72], there has been a growing demand to provide users with a seamless experience across multiple mobile systems, not just use them as separate, individual systems. We refer to the ability to combine the functionality of multiple mobile systems into a more capable one as *multi-mobile* computing [1, 2].

Three examples help illustrate some of the possibilities. First, multi-mobile computing makes it straightforward to remote control other devices such as cameras, and provide richer input modalities beyond touchscreen input such as motion-based game controllers or more accessible interfaces for users with disabilities. Second, multi-mobile computing makes it easy for users to combine their tablets together in a self-organizing multi-headed display and input surface for a big screen experience for all users anywhere, even when a big bulky screen is not available, as shown in Figure 1. Similarly, multiple smartphones can combine cameras together to provide panoramic video recording without specialized hardware. Third, multi-mobile computing makes it easy to use users' smartphones distributed across a room to leverage their microphones from multiple vantage points. Together, they can provide superior speaker-identifiable sound quality and noise cancellation for audio conferences, without costly specialized equipment. Unlike simple one-to-one I/O sharing approaches [4, 15, 30, 52] such as Apple AirPlay [10] which can display content from a smartphone to an Apple TV [11], multi-mobile computing envisions a broader, richer experience 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, three key challenges must be met to turn this potential into reality. First, mobile systems are highly heterogeneous; on Android alone, more than 24,000 different systems are available [55]. 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. Second, smartphone and tablet devices consume and produce a wide range of disparate input and output data in heterogeneous data formats, 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. Finally, good performance is essential to support high-bandwidth and time-critical devices across the network.

We introduce M2, the first system for heterogeneous, transparent, multi-mobile computing. M2 redirects and transforms heterogeneous device input and output across heterogeneous, commodity mobile systems to transparently enable new ways of sharing and combining multiple devices with existing unmodified apps. M2 is based on two key observations. First, unlike traditional desktop and server systems, mobile systems are highly vertically integrated, and each system has its own device-specific APIs. These APIs are often nonstandard and incompatible with other systems making heterogeneity difficult to support by forwarding or remotely calling these APIs. Second, although lower-level APIs are often device-specific, the higher-level semantics of device data are well-known and device data may often have a common format across different platforms, e.g., H.264 video data.

To solve the device and data heterogeneity problem, M2 builds on these observations and takes a novel data-centric approach: it shares device data, not device APIs. This is possible by importing and exporting data to and from each mobile system using common cross-platform device data formats, avoiding the need to bridge incompatible device APIs. This not only allows for simple heterogeneous device remoting, but also enables mixing and matching of different device types across heterogeneous multiple systems. M2 introduces *device transformation*, a framework that enables disparate devices across different systems to be substituted and combined with one another to support multi-mobile heterogeneity, functionality, and transparency. For example, M2 can match heterogeneous input devices with different coordinate systems. As another example, M2 can enable an app to use, in lieu of a local camera, any remote video device for input, including a remote camera or remote display output transparently without requiring any app modifications. Different types of transformations exist under this framework, such as *fused devices*, which fuse multiple devices as shown in Figure 1. M2's approach makes it possible for the first time to efficiently share and access devices across multiple heterogeneous mobile systems.

M2 leverages higher-level device abstractions and mobile system hardware features to optimize the transfer of device data across mobile systems with no user-perceived loss of fidelity. For higher-bandwidth devices, M2 takes advantage of encoding and decoding hardware widely deployed on mobile systems to efficiently compress device data before transferring it. 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 M2 on Android and demonstrate that it transparently provides new multi-mobile functionality for existing unmodified Android apps using both Android and iOS remote devices. We show various example use cases that utilize M2 and take advantage of multi-mobile computing. M2 allows any stock Android or iOS system to share its devices by running an app which can be made available in Google Play [26] or the Apple App Store [12]. It only requires modest user-level framework modifications to allow unmodified Android apps to access and combine local and remote devices from multiple mobile systems. We show that M2 operates seamlessly across heterogeneous mobile software and hardware systems, including multiple iOS and Android versions running on different smartphones and tablets. We demonstrate that M2 provides multi-mobile functionality with low latency and only modest performance overhead across even high-bandwidth devices such as camera, audio, and display, even for 3D graphics-intensive apps. Using both standard WiFi networks and WiFi Direct [73], our Institutional Review Board (IRB)-approved user studies show that the display performance using multiple remote devices with a wide range of popular apps from Google Play is well synchronized and 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 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 downloading the M2 app from their platform's app store, e.g. Google Play or 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. To share one or more devices using the app, the user creates a *device profile*, which consists of a profile name, a list of devices to share, an optional password, and optional access control options that can restrict the systems that can access the device.

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, which processes touchscreen input just like any other app and forwards 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 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.

To run apps that access remote devices on other mobile systems, the M2 native frameworks must be installed on the device client. In practice, we envision mobile ecosystem vendors such as Google and Apple would include these modifications in their frameworks to transparently provide multi-mobile functionality to their existing large installed base of apps; our current implementation fully supports Android but only supports iOS device servers. Once the M2 native frameworks are 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. Devices on a system are never shared without user approval. Multicast DNS (mDNS) [31] is used to facilitate peer discovery. Using the app on the client, the user can select a device profile on a server, input the required password if the device profile is being used for the first time, 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, 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. M2 therefore relies on users to decide how to share devices. 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 app basis so that different apps may use different usage profiles at a given time.

3 M2 ARCHITECTURE

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

3.1 Background

We first provide a brief overview of the way devices are used in mobile systems, using Android as an exemplary system. Android can be thought of as having a tall interface to devices through multiple layers of software [14]. 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 via Inter-Process Communication (IPC) to Android system services, which are shared, long-running system processes that run in the background and are used to manage devices.

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 [27, 28]. 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. 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, many 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 IPC, and those system services then manage lower-level device functionality [6, 7].

Given this background regarding the Android device infrastructure, there are a number of ways in which device functionality can be provided from a device server to a device client. At the most basic level, this involves remoting device functionality from server to client. The common approach used for one-to-one remote device access is to forward calls from client to server, which can be done at different layers of the device stack. However, this approach has fundamental limitations in the context of multi-mobile computing.

For example, Rio [4] partitions the device stack at the kernel interface using traditional device files as the abstraction between the server, with the real device file, and the client, with the virtual device file. The virtual device forwards HAL library interactions to the server, bypassing the local I/O device. For each device, the approach requires modifying a potentially complex HAL library to add non-trivial device-specific and system-specific changes to support Rio. These modifications are not possible for devices with closed-source proprietary libraries, as is the case with modern devices. The client and server must also use the same modified device-specific HAL library, which is Android version specific and must be ported to each Android system. For example, whereas a standard Android system would have one HAL library for the GPS, Rio would require the system to have a separate HAL library for each different combination of remote GPS and Android version. Given the number of devices per system and the many different devices available across different Android systems, this approach does not scale to support device heterogeneity. Furthermore, Android is designed to use a single HAL library per device, making it impossible to support using multiple heterogeneous remote and local devices at the same time without significant restructuring of Android. For example, a user cannot share their audio with another system and still receive local notification sounds.

As another example, Mobile Plus [52] partitions the device stack at the IPC interface to system services by forwarding IPC calls remotely. This will also not work across heterogeneous systems since forwarding requires the exact same IPC interfaces to system services on both systems. These interfaces vary between Android versions making interoperability problematic across Android versions not to mention non-Android systems. IPC interfaces also 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.

device	system service	M2 mods. (LOC)
sensor	SensorService	123
input	InputFlinger	167
location	LocationManagerService	97
mic	AudioFlinger	86
camera	CameraService	201
audio	AudioFlinger	77
display	SurfaceFlinger	57

Table 1: Android system services and M2 modifications

Regardless of the partitioning approach or layer, matching APIs to forward device-related calls is problematic for multi-mobile computing. First, device APIs usually rely on the hardware model, driver, and device state, making it difficult if not impossible to match these APIs across heterogeneous mobile systems. Second, not all app-device communication can be captured by forwarding calls, such as data exchanged via shared memory, making it problematic to support full device functionality. Third, forwarding APIs usually involves exchanging raw API-specific data which is detrimental to performance for high-bandwidth devices, as evident by the performance problems and lack of high-bandwidth device support in approaches such as Rio and Mobile Plus. Finally, multi-mobile computing enables sharing, combining, and mixing multiple devices, not just one-to-one device sharing. Trying to match APIs and forward calls across multiple heterogeneous devices is an enormous problem especially since each call cannot represent more than a single hardware device or type.

3.2 Client-Server Device Stack

To solve the device heterogeneity problem, M2 takes a fundamentally different data-centric approach that leverages higher-level device abstractions and hardware acceleration to efficiently share device data, not device APIs. We observe that although lower-level APIs are often device-specific and tied to a given mobile platform, the higher-level semantics of device data are well-known and often have a common format across different platforms. For example, PCM is used for audio and JPEG is used for camera images. Instead of attempting to match heterogeneous APIs and forward device calls, M2 imports and exports device data while using each platform's own APIs. M2 converts, manipulates, forwards, and injects device data, not APIs, across systems without the need to match APIs. This allows M2 to handle device heterogeneity, portability, and composition effectively as long as each platform is able to export device data as well as import and process device data.

We make three observations regarding the device software stack in mobile systems that suggests an approach for exporting and importing device data across heterogeneous devices, hardware, and software. First, for hardware devices of interest such as input, camera, audio, display, sensors, microphone, and GPS, mobile apps do not access such hardware devices directly but go through system services. Anything below the level of system services is irrelevant as far as apps are concerned. Second, system services manage all access to these hardware devices and represent device data in standard or well-known formats. This makes device data manipulation feasible and universal to all apps on a system. Finally, the primary interface between apps and devices is at user-level, not kernel-level, suggesting that a user-level approach for handling device data is sufficient.

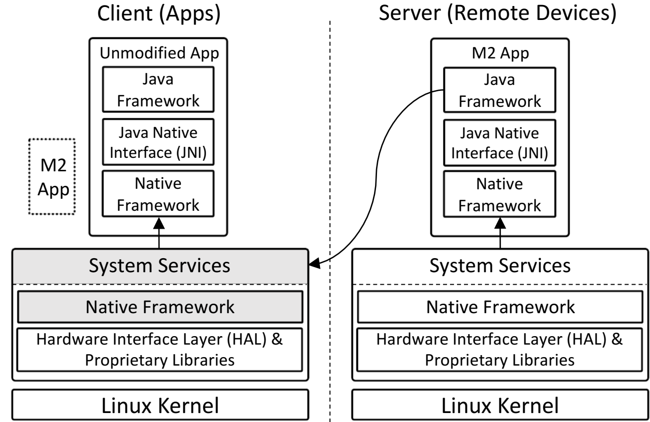

Figure 2: M2 architecture; Android modifications in grey

Based on these observations, M2 takes a unique approach to partitioning device functionality between device server and client that leverages the characteristics of mobile systems. On the server, an M2 app is used to leverage the entire device stack to access device data via the same public APIs used by all mobile apps. This completely avoids the need to modify the software stack and makes importing or exporting device data as easy and portable as writing an app on any platform to access local device data.

On the client, M2 introduces user-level virtual devices that import or export remote device data and appear just like normal local devices via system services. By introducing virtual devices via system services, M2 avoids the need to implement low-level device interfaces that are not used by apps and leverages the same platform's own higher-level device interfaces to create virtual devices that import or export device data. Virtual devices do not need to be interface compatible with any other platform other than the one they are executed on, avoiding interface compatibility issues across heterogeneous systems. Because data formats are well-known, implementing virtual devices on each platform is straightforward. Furthermore, virtual devices can also be created to compose data involving multiple devices and device types. Figure 2 shows an overview of the M2 architecture in Android.

In addition to importing and exporting device data, auxiliary information about the data is often included. This information is used to indicate what the format is for the device data and the functionality it represents. Table 2 lists some of the auxiliary options. Functionality to distinguish between, for example, camera preview and picture data is reflected in the auxiliary information in the device data. This auxiliary information only ever consists of a few bytes, resulting in negligible overhead while facilitating heterogeneous device support.

Implementing virtual devices requires minimal modifications to a device's respective system service and supporting native frameworks, as discussed in Section 4. Each system service is modified such that it can receive common M2 data formats over the network and convert them to the format the platform expects. Receiving device data, system services can then encapsulate them as additional virtual devices presented to apps. Table 2 provides a summary of formats and types used by M2 for each hardware device. The definition of these types and formats under M2 is essential to provide a portable

device	identifiers, data types, and formats
sensor	sensor type, data fields (x, y, z), aux data (e.g., rate)
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, bit/sample rate
camera	function (pic vs. video), media format, resolution, aux data (e.g., flash, timer, HDR, orientation)
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

translation medium across different Android versions, hardware devices, and platforms (e.g., iOS to Android and vice versa).

As an example, following Figure 2, assume an M2 touchscreen input device client and server. On the server, with the M2 app in the foreground, when the user touches the screen, the input event is captured by the M2 app. The M2 app then standardizes and packages the input event details, e.g. the coordinates, into the M2 input format. The event is then forwarded to the input system of the client along with auxiliary information, notably the server's display resolution. Upon receipt, the client's input system service unpackages the input event and converts it to the client platform's input format, scaling the coordinates if necessary based on the provided auxiliary information.

This device data-centric architecture solves the key heterogeneity challenges of multi-mobile computing. As evidence of this, our M2 prototype has been tested to work, reusing the exact same code, across five recent and widely used major Android versions.

3.3 Device Transformations

M2 introduces a device transformation framework that makes it possible for existing apps to transparently use and compose 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 M2 data format. The transformation function is used to convert the input device abstraction to the output device abstraction, and can operate on device data or control information.

To support fused devices, translated devices, and other transformations, M2 provides a transformation plugin framework that operates in conjunction with the M2 app. Transformation plugins can be installed and run on either a device server or client. Plugins provide a way for vendors or developers to provide transformations, which can be integrated into M2. In Android, a plugin is a standalone Android Application Package (APK); it can be downloaded through Google Play like other Google services. The M2 app exposes a Remote Procedure Call (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 devices. 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 devices or exposed as a new shareable device. On the device server, which only needs to run the M2 app, plugins can have access to most devices at any time, but input only when the M2 app is in the foreground. On the device client, plugins can have access to devices at any time by leveraging the 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 [9] to support plugins for iOS.

This framework provides four key benefits. First, it provides a way to enhance how users interact with existing apps without the need to modify them. Second, by allowing plugins to operate on the server or client, it provides maximum flexibility to support a wide range of transformation functionality. Third, by leveraging standard plugin functionality via APKs, it makes it possible for vendors and 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, Table 3 shows a non-exhaustive list of different types of device transformations, along with a description of each and some examples that illustrate the versatility of transformations in M2. One example is a fused display/input 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, the display client, and displaying content on its display and the three other display servers. Additionally, the app is receiving touch input from the four touchscreen devices exposed by M2. To provide fused display, each system 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 display client is simultaneously providing display output to other display servers while its own screen is also part of the fused display.

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 all four input devices, one local and three remote, 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.

Replacement devices can be used to support devices available at a device server but unknown to a device client. Although M2 allows each system to use its own device APIs to process device data, it requires an API to exist. If a device client does not have any support for a given type of device, it will not be able to access it remotely. Replacement devices can be used to bridge this gap by translating

transformation types	description	examples
fused	one to many of the same type	multi-headed display/input, mirroring speakers
translated	one to another of different types	camera (eye tracking) to input, audio to display (visualization), sensor to input (Wii-like)
piped	output to input device	display to cam (Netflix recording), audio to mic (high fidelity audio recording)
replacement	translated dev from an unknown dev	3D touch (iOS) to touchscreen input, pressure sensor to system without a pressure sensor
merged	merging data components of devices	different channels from fused mic (record surround sound), fused camera with merged frames (e.g., green screen effect to superimpose objects from one frame onto another)

Table 3: M2 device transformation types

an unsupported device into a supported one. For example, Android does not have existing APIs for pressure sensitive touchscreens such as Apple's 3D Touch [8]. M2 can run a device server transformation extension on iOS that uses the iOS 3D Touch API to translate force to input touch duration which can then be processed as a regular touchscreen input device on Android.

To support various types of device transformations, M2 implements a modified version of BeepBeep [57] for indoor localization. This mechanism can be used to automatically configure device transformations that are location dependent, such as determining relative positioning of devices for fused devices. For example, a fused display grid can be automatically configured based on the locations of the systems in the grid. Other localization techniques [41, 71] may also be used for M2.

3.4 Network Communication

M2 clients and servers communicate over standard network sockets and are designed to interact over WiFi and WiFi Direct networks, the latter supporting groups of mobile users anywhere without being tethered to network infrastructure. M2 leverages common hardware features of mobile systems to optimize network performance and security, and is designed to operate effectively even in the presence of intermittent network failures.

Performance Since high-bandwidth devices send visual and audio data, M2 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. 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 frames per second (fps) frame rates since they are visually indistinguishable from higher frame rates for end users [67]. 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 only encode and decode a limited number of data streams simultaneously. Given this, 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. This applies to fused display when 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 efficiently scaled and resized appropriately for viewing at the remote display based on the hardware characteristics of the respective screen.

Security To optimize network security, M2 leverages AES acceleration hardware on mobile systems to provide secure client-server communications using 128-bit AES encryption. At install time, the M2 app generates a public/private key pair to facilitate secure sharing of runtime generated AES session keys. These public keys are exchanged between relevant mobile systems once the user specifies a device profile to use (the key's fingerprints are visible to the user on all systems allowing the user to verify their authenticity). M2 encrypts device 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.

Reliability To ensure operational reliability, M2 maintains all app state on the device client and tailors the transport mechanism used based on the device data being delivered. By relying on the decoupling of apps from devices provided by system services, M2 treats remote devices as dumb peripherals. App state is entirely on the client and network disconnections do not cause app failures. A remote device can be disconnected and reconnected at any time and an app that was using the device will continue to function properly in the presence of such device disconnections. 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. This is useful in the presence of intermittent network disconnections between server and client due to system mobility or other environment conditions affecting wireless networks.

Network disconnections are managed by system services, which can provide transparent support for apps. For example, if a sensor device becomes persistently disconnected, the `SensorService` can simply replace the original sensor data source with a local sensor device instead to provide sensor data. M2 distinguishes between intermittent and persistent network disconnections based on heartbeats and timeouts. By default, M2 does nothing on a persistent network disconnection, so apps continue normal operation but receive no input and remote devices receive no output. Unmodified apps are resilient to failures from receiving no input since they rely on callbacks for device data. For example, if a connected remote sensor does not exist locally and a disconnection occurs, the app will continue to function whilst waiting for remote sensor data to trigger the app's callback function.

For control messages and devices which expect lossless data, M2 uses TCP to ensure reliable data delivery. For devices such as display, audio, and the camera preview, UDP is used since some loss can be

tolerated and timing is important for these streaming devices. 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 [40, 48, 62, 64]. Multicast or pseudo-broadcast [33] may become viable options to use in the future, but existing implementations either have poor performance or require firmware changes.

4 IMPLEMENTATION

We implemented M2 in the Android Open Source Project (AOSP) [25], and also as an M2 device server app for iOS, enabling remote device access across Android and iOS. The implementation has three parts: an M2 app, minor modifications to Android system services, and an M2 support library. The modifications required for each Android system service are modest, as shown in Table 1, averaging fewer than 120 lines of code (LOC). Since these changes are so minimally invasive, supporting newer platform versions often requires no additional modifications. Similarly, any newly introduced devices can be easily supported so long as the device is used via a system service. An M2 format for the device will need to be created, the system service modified to support this format, and the M2 app updated to access the new device.

The bulk of the code is system service independent and confined to the M2 support library, which provides networking, encoding, encryption, and configuration APIs. The latter is used to get configuration-related profile information such as what and how devices should be shared. This is useful for system services to decide where the data should go. M2 also uses the Mongoose embedded networking library [18] and a JSON parser library [42]. Except for Android system services modifications, all of the code is portable across Android and iOS. Due to space constraints, we only provide a brief overview of how M2 interfaces with Android to capture and deliver device data. These same interface points are also used to support device transformation plugins.

For input, location, and sensor devices, the M2 app obtains device data on the device server by using public Android APIs to register listener classes: an `OnTouchListener` for input events, a `LocationListener` for location, and a `SensorEventListener` for sensors. Upon receipt of device data, the M2 app uses the M2 library APIs to package the data and send it to device clients. On the device client, system services are modified to listen for this data and upon receipt, provide this data to apps. For input devices, `InputFlinger` is modified to call Android's `InputDispatcher` `injectInputEvent` function, providing the app with the input event. For location devices, the `LocationManagerService` is modified such that upon receipt of a location update from a device server, an app's `LocationListener` callback function is called and given the data as a `Location` argument. For sensor devices, unlike location devices, it is not sufficient to call an app's listener callback function directly because apps can also use an alternative queue-based approach for obtaining sensor data via Android's Native Development Kit (NDK). Since both the Java framework and NDK use SensorService's `SensorEventQueue` to obtain events, we modify it to also deliver remote sensor data from device servers when it is read.

Supporting audio output and input involved similar methods. To support output devices such as speakers, the M2 app on the device server listens for audio data sent from a device client. The app then plays the received audio via Android's `AudioTrack` API. To capture audio on the device client, `AudioFlinger`'s `FastMixer` main loop, `onWork`, where audio is mixed and played, is modified to intercept audio buffers. The audio is then encoded and forwarded to the device server. Similarly, to support microphone, the M2 app on the device server uses Android's `AudioRecord` API to record audio, encode it, and send it to a device client. On the device client, `AudioFlinger`'s `RecordThread`'s main loop, `threadLoop`, where most audio is recorded, is modified to write received audio into the receiving app's `RecordTrack` buffer. Equivalent modifications to `AudioFlinger` for audio input support were also made for the low-latency `FastCapture` thread.

To support display, the M2 app on the device server receives video data from the device client, decodes it, and draws it on a full screen `Surface`, a buffer that Android's screen compositor displays to the user. To send the video data, the device client uses Android's `VirtualDisplay` to add an additional mirrored output. The frames Android sends to the `VirtualDisplay` are then encoded and sent to a device server. We also modified `SurfaceFlinger` to support ignoring tagged `Surfaces`. For example, supporting fused display involves drawing full screen video data on one `Surface` that is sent to the `VirtualDisplay`, then hiding that `Surface` behind another on which is drawn the portion of the screen that should be visible locally. The top `Surface` is only visible locally and is tagged so that it is ignored and not sent to the `VirtualDisplay`.

To support cameras, the M2 app on the device server provides a `Surface` on which the camera preview displays camera frames. The `Surface` is recorded by the M2 app by taking the raw frames, encoding them, and sending them to the device client. To enable apps on the device client to use the remote camera preview, the device client's `CameraService`'s `CameraClient` and `Camera2Client` are modified such that the received preview frames are drawn to an app's preview `Surface` instead of activating and using local camera frames. To take a photo, the M2 app on the device server is sent the request, takes the photo, and sends it to the device client. On the client, the `CameraService` is modified to call an app's `onPictureTaken` callback, used for receiving a taken picture, upon receipt of a photo from a device server. This callback is implemented by apps as part of the Android photo taking workflow. Camera options such as HDR, flash, orientation, etc., are reflected in the auxiliary information. To take a video, the M2 app on the device server is sent the request, starts recording video, and sends the file in chunks.

5 EVALUATION

We measured the performance of M2 across a range of Android and iOS hardware and software, including both tablets and smartphones. We first describe ways in which we have used M2 with popular unmodified Android apps in various user studies, then present some quantitative performance measurements, and finally compare M2 with other approaches. We also provide a video demo [3].

As listed in Table 4, we ran M2 across heterogeneous smartphone and tablet configurations running five different SoCs and nine different versions of Android and iOS, including Android versions Jelly Bean (4.3), KitKat (4.4), Lollipop (5.0), Marshmallow (6.0.x), and Nougat (7.1.1). We conducted experiments with both WiFi Direct

name	system	type	display	SoC	OS
N4J	Nexus 4	tablet	768x1280	Snapdragon S4 Pro	Android 4.3
N4	Nexus 4	tablet	768x1280	Snapdragon S4 Pro	Android 4.4
N5	Nexus 5	phone	1080x1920	Snapdragon 800	Android 6.0.1
N7	Nexus 7	tablet	1200x1920	Snapdragon S4 Pro	Android 6.0
N9	Nexus 9	tablet	1536x2048	Tegra K1	Android 5.0
N9N	Nexus 9	tablet	1536x2048	Tegra K1	Android 7.1.1
iPd	iPad mini	tablet	768x1024	Apple A5	iOS 8.2
iPh9	iPhone 6S	phone	750x1334	Apple A9	iOS 9.3.1
iPh	iPhone 6S	phone	750x1334	Apple A9	iOS 10.3.1

Table 4: Heterogeneous systems used for running M2

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; the other systems use IEEE 802.11n.

5.1 Example Use Cases

iOS and Android apps on iPhone First, we simply made iOS remote devices available to Android apps, including sending iOS touchscreen input to the apps and displaying app output to the iOS display, allowing an iPhone 6S with an unmodified iOS to effectively run unmodified iOS and Android apps from the same system for the first time.

Self-organizing multi-headed display and input Second, we used fused devices for display and input to allow four systems in a 2x2 layout to be combined as one, as shown in Figure 1. The systems automatically self-configure using localization to identify which system was positioned where in the 2x2 grid, and can be rearranged at any time on the fly. Display and input are split across all screens providing a larger multi-headed display experience that is easily portable and available anywhere. Because displaying across multiple devices can be bandwidth-intensive, we used this M2 configuration for many of our performance measurements in Section 5.2, including using popular display-intensive Android apps. As a useful sysadmin alternative, we also did the reverse, allow one system to split its screen in a 2x2 layout to control four other systems instead. We did a small user study by asking 20 users, half of them computer-savvy users and half of them not, to compare the display quality while watching Big Buck Bunny [17], the widely used open movie project, on the 2x2 fused display versus a single Nexus 9 local display. All of the users said that the fused display performance was well synchronized and visually indistinguishable from using a single local display, except for being clearly larger and multi-headed with some bezel separation between the displays. The visibility of bezels will only continue to diminish as mobile systems increasingly have higher screen-to-body ratios [36, 61, 68, 74].

Audio conferencing Third, we created a fused microphone device to allow microphone input from the microphones of multiple systems to be combined as one, providing a 3D-like audio experience useful for Skype [43] or audio conferencing apps. Microphone data from multiple sources was mixed to stereo channels. The same sampling rate, timestamps, and an initial beep at a specified frequency were used for audio synchronization across systems. As an experiment, we placed a Skype VoIP call to a conference room and had the person in the room move around while speaking. The conference room was set up with three smartphones placed at different locations in the room and a Polycom SoundStation IP 7000 teleconferencing system in the center of the room. We recorded the call audio from the conference room when using one smartphone, fused microphone with the three smartphones, and the dedicated teleconferencing system. We then did a small user study by asking 20 users, half of them computer-savvy users and half of them not, to listen to the three recordings. All of them said that the fused microphone from the smartphones provided better audio clarity than the other approaches. It also provides the same ease of use of just having to connect one system in the room to the audio conference.

Wii-like Android gaming Fourth, 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 V shape, which match various first-person Android game experiences. We used this translated device configuration for two Android games, Epic Swords 2 and 3D Tennis. We did a small user study with 20 users, half of them computer-savvy users and half of them not, and all of them said that using a smartphone to control the game running on a tablet via M2 provided a much more intuitive, realistic gaming experience than the native game used with touchscreen input. Epic Swords 2 feels more realistic swinging a smartphone to control sword movements during a sword fight than swiping across the touchscreen, especially using a realistic stabbing movement to stab the sword instead of an unnatural V swipe using the touchscreen. 3D Tennis also feels more realistic swinging a smartphone as a tennis racquet to play tennis instead of swiping across the touchscreen.

Eye movement input for hands-free users Fifth, we used a translated device to transform eye movements into touchscreen input. This is a useful accessibility feature for disabled users without good use of their hands or users whose hands are full, e.g., while cooking. The device server used the camera for eye tracking via face detection functionality provided by OpenCV [54], an open source library for real-time computer vision. The movements were then transformed into input swipes and sent as input data so that the device client receives touchscreen input data instead. We were able to use this successfully with a number of Android apps ranging from reading an ebook using the Amazon Kindle app to playing Temple Run by simply looking left, right, up, and down to effectively swipe in those directions.

High-quality media recording Sixth, we used a translated device 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. This allows any camera app on the remote system to view and record the audio and video output of another system. The camera app perceives the display and audio data streams as camera and microphone input, but not via the physical microphone or camera lens. 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 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. Similarly, we recorded a video being played by the YouTube app. The recordings were high quality, well synchronized, and playable using the Android stock video player, allowing the user to play them at a later time, e.g., on an airplane.

Display performance (PassMark)	
Stock	Stock AOSP on N9
Idle	M2 installed on N9 but idle
One	M2 displaying locally on the same N9
Two	M2 w/ 2 N9s in a 2x1 configuration
Four	M2 w/ 4 N9s in a 2x2 configuration (Figure 1)
Mixed	M2 w/ 1-to-many mirroring from N9 to N9N, N7, and N5
Mixed iOS	Same as mixed but mirroring to iPd instead of N9N
Camera latency (take photo)	
N4 AOSP	Stock AOSP on N4, 8MP camera
N7 AOSP	Stock AOSP on N7, 5MP camera
N9/N9N AOSP	Stock AOSP on N9/N9N, 8MP camera
iPh	iPhone 6s, 12MP camera
N7 using N4	M2 N7 using remote N4 camera
N7 using N9	M2 N7 using remote N9 camera
N4 using N9	M2 N4 using remote N9 camera
N4 using N7	M2 N4 using remote N7 camera
N4 using N7	M2 N4 using remote N7 camera
N9N using iPh	M2 N9N using remote iPh camera
Audio latency (Zoiper)	
N7 AOSP	Stock AOSP on N7, local mic and speaker
N7 Idle	M2 installed on N7 but idle, local mic and speaker
N7 w/ N7 speaker	M2 N7 using local mic and remote N7 speaker
N7 w/ N7 mic	M2 N7 using local speaker and remote N7 mic
N7 w/ N7 mic+speaker	M2 N7 using remote N7 mic and speaker
N7 w/ N4J mic	M2 N7 using local speaker and remote N4J mic
N7 w/ N9 speaker	M2 N7 using local mic and remote N9 speaker
Multi-mobile Android apps (7 system multi-mobile configuration)	
7 Android systems combined in a multi-mobile setup with 1 N9 client running the apps, 60 fps, 10 Mbps bit rate, 1 N4 providing remote sensor and touchscreen input, 2 N7s as remote speakers, separate left and right audio channels, and 3 N9s as display servers in 2x2 large display grid with N9 client	
Power consumption	
Stock	Stock AOSP on N5
One	M2 N5 local device usage with no remote devices
Two	M2 N5 client plus an N7 device server
Four	M2 N5 client, 2 N7 and 1 N9 device servers
Remote	M2 N5 device server with an N7 client

Table 5: M2 configurations for running benchmarks

Panoramic video recording Seventh, we used fused devices to enable panoramic video recording using the cameras of multiple smartphones without any specialized hardware and using the existing unmodified stock Android camera app. Smartphones provide panoramic photo functionality, but no panoramic video. We fuse the camera inputs from two systems to create a wider panoramic view which can be recorded as video or photos. This is useful, for example, when parents seated next to each other want to record a child's school or sports performance. Often, neither parents' smartphones can individually capture the whole performance, but taken together with M2, they can provide a complete panoramic video recording. The prototype requires the two systems to be relatively fixed in position with respect to each other and video quality is sensitive to camera angle and positioning. Better methods for stitching video data from multiple cameras can be used [58].

5.2 Performance Measurements

We ran benchmarks and unmodified Android apps from Google Play to quantify M2 performance using various combinations of tablets and smartphones serving as different remote devices, as listed in Table 5. Systems are named as listed in Table 4.

Display performance We first measure display performance since it is crucial and a key performance challenge. Using the multi-headed display and input configuration from Section 5.1, we ran the widely used Android PassMark benchmark [56], a set of resource intensive tests to evaluate CPU, memory, I/O, and graphics performance. To account for frame drops at display servers not captured by the benchmark app, we scale the results based on the percentage of frames displayed at the server, similar to slow-motion benchmarking [50, 77]. 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 listed in Table 5, all using an N9 to run the app and various display servers, including an N9N, N7, N5, and iPd. 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 show that M2 can run 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 3 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, the remote devices performed similarly. Figure 3 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 comparing the mixed 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 slightly better performance since it does not send packets out to the network, but includes encryption/decryption and encoding/decoding costs. This is more apparent with the solid vectors, image filters, 3D complex 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. PassMark is designed to stress test the system, so its quantitative performance is a conservative measure of real app performance.

Figure 4 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. 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 0.3 Mbps due to display updates only for a progress bar

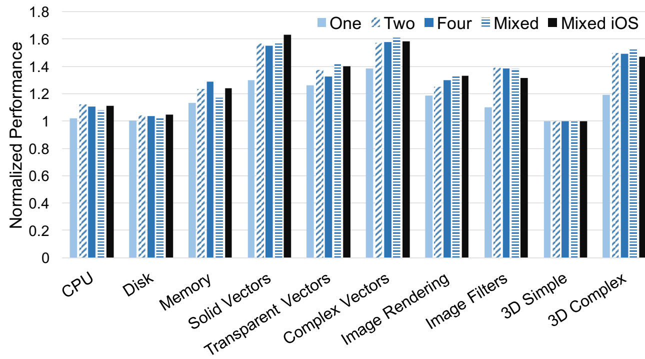

Figure 3: PassMark performance; lower is better

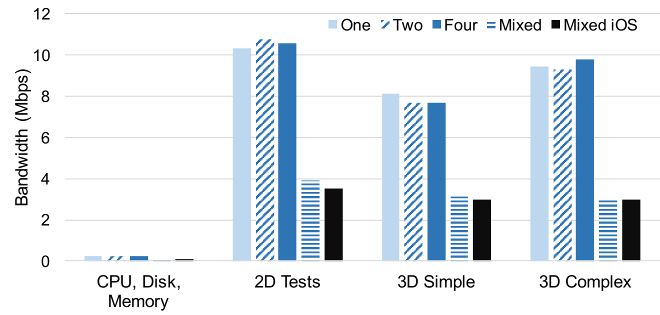

Figure 4: PassMark per device bandwidth

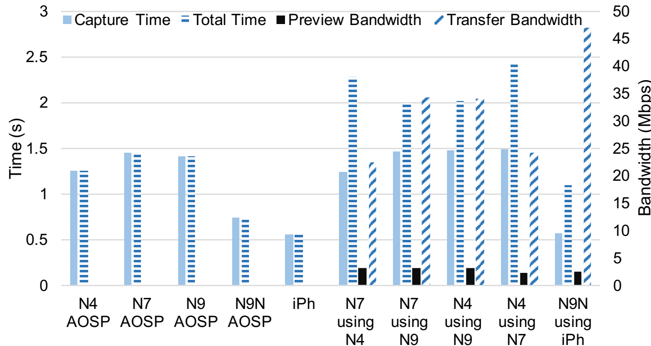

Figure 5: Camera latency for taking/storing pictures

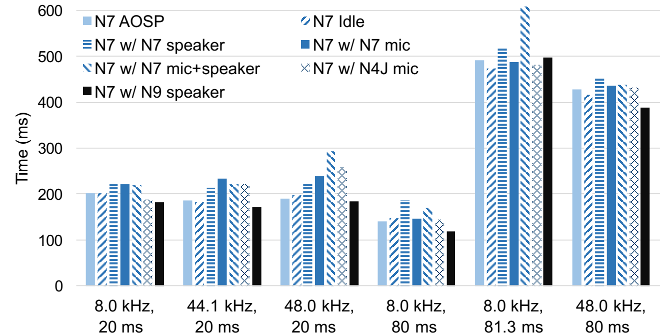

Figure 6: Audio latency using Zoiper

and test results. For the 2D tests, the bandwidth required was up to about 10 Mbps for the 1536x2048 remote display tests and 4 Mbps for the 720x1280 remote display tests. Our results show that WiFi can meet the bandwidth requirements for 3D graphics-intensive display data, providing good M2 performance.

Camera latency We next measure camera latency performance. Due to a lack of Android camera performance app benchmarks, 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. When using a remote camera, the picture was transferred from the remote camera to the default local storage for the app. We measured the performance using ten camera configurations listed in Table 5, five systems, N9, N9N, N7, N4, and iPh, and five different remote camera scenarios. 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. The final scenario illustrates using a different platform's higher quality remote camera.

Figure 5 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. 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 take less than 1.5 seconds in all cases with the iPh 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 iPh 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 5 also shows the bandwidth requirements for taking a picture, including both the camera preview and transferring the picture taken from the remote camera to local storage. The camera preview runs at a lower resolution than the native display resolution, so its bandwidth requirement is 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, up to about 45 Mbps for the faster N9.

Audio latency We next measure audio and microphone latency using Zoiper [65], an audio benchmark. It 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

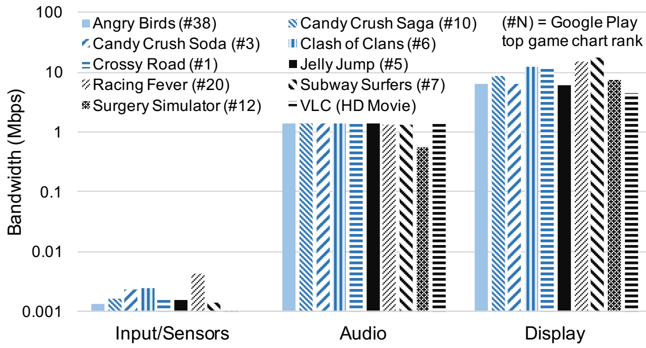

Figure 7: Android apps in a seven system M2 configuration

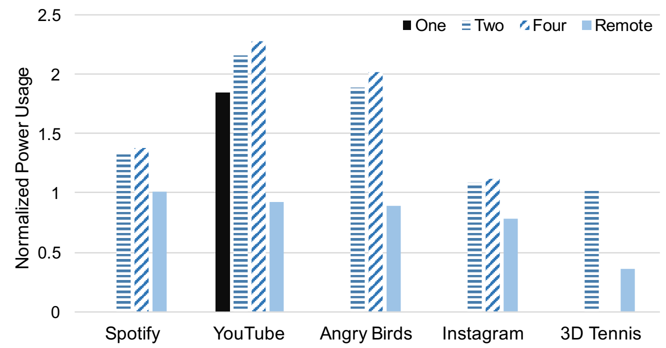

Figure 8: Power measurements using M2; lower is better

the audio path. We tested seven configurations of local and remote speakers and microphones listed in Table 5.

Figure 6 shows the audio latency measurements. For most tests, M2 adds negligible latency compared to stock Android, even for using remote microphones and speakers. 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.

Multi-mobile Android apps To measure using multiple remote devices with audio and display streaming, we used seven Android systems together in the multi-mobile setup listed in Table 5. 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 [70] movie player app for comparison purposes. 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 [17].

M2's qualitative performance for all of the apps was indistinguishable from running on an N9 with stock Android Lollipop. Audio was clear with no drops, and display was smooth with no noticeable skipped frames or display degradation. Figure 7 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.

Power consumption To measure power consumption, we connected an N5 to a Monsoon power monitor [47] and recorded its overall power consumption as a device client and server when sharing several individual hardware devices, sharing a combination of hardware devices, and using a transformation plugin. Using various popular apps, we ran M2 in five configurations listed in Table 5 with five workloads, each run for three minutes: (1) Spotify for sharing audio, (2) YouTube (browser) running a 1080p video for sharing a mirrored display, (3) Angry Birds for sharing a split display, audio, and input in a fused scenario, (4) Instagram using remote camera for sharing the camera, (5) 3D Tennis for testing sensor to input transformation. Display encoding was done at 1920x1080 resolution, the maximum supported by N5, with 30 fps and a 10 Mbps bit rate.

Figure 8 shows the power measurements normalized to stock Android running the respective app. For remote, power consumption is lower using M2 as a device server instead of running the respective Android app. For running the N5 with multiple remote devices, display sharing is the most expensive, though the cost in all cases for the device client does not increase much with more systems. There is negligible power overhead associated with sharing camera, sensors, input, GPS, and supporting a transformation, as shown by 3D Tennis and Instagram. While there are some additional power costs when using M2, they are not prohibitive and can be quite modest while delivering powerful new functionality and a much better user experience, as in the case of 3D Tennis.

5.3 Comparison with Other Approaches

Since I/O sharing is one aspect of M2, we compared this basic functionality against other systems that also provide I/O sharing. Table 6 lists the devices supported by M2 versus other I/O sharing systems, including Rio [4], Mobile Plus [52], Apple AirPlay [10], Google Chromecast [24], Microsoft RDP [46], and VNC [30], which are discussed further in Section 6. None of the other systems support the range of devices possible with M2, and device support is limited in many cases for systems that provide it. For example, RDP and VNC support display sharing, but at best provide poor quality for display-intensive apps such as 3D gaming or video streaming. Rio and Mobile Plus support camera sharing, but only at 2-3 fps frame rates. M2's approach for device data sharing supports heterogeneity, shows good performance, and is more efficient than device API sharing approaches.

We also ran a direct I/O sharing performance comparison with Rio since it supports the most devices other than M2. Rio runs on Android and is open source [5], but does not work on any current Android systems as it was designed for the discontinued Galaxy Nexus smartphone and CyanogenMod [20] 10.1.3-RC1, based on Android JellyBean 4.2. Updating Rio to work on any current Android system is fundamentally problematic due to Rio's low-level approach as discussed in Section 3. It requires modifications to vendor-specific HAL libraries which are proprietary and no longer open source for modern Android systems. We instead ported M2 to run on the old Galaxy Nexus with JellyBean 4.2. When using the Galaxy Nexus local camera natively, the camera preview only ran at 15 fps at the default 480p resolution as measured by HAL statistics. M2 delivered the same 15 fps at the same resolution when camera sharing from one Galaxy Nexus to another, while Rio delivered only 1.6 fps at 480p

device \ system	M2	Rio	Mobile+	AirPlay	Chromecast	RDP	VNC
display	✓			✓	✓	✓	✓
camera	✓	✓	✓				
audio	✓	✓		✓	✓	✓	
mic	✓	✓					
accelerometer	✓	✓	✓				
other sensors	✓						
input	✓					✓	✓
location	✓						

Table 6: Comparison of basic I/O sharing device support

resolution, almost an order of magnitude worse performance than M2. M2 delivered 30 fps at 720p resolution, even better than native Galaxy Nexus performance, when camera sharing from an N7 to a Galaxy Nexus. This was not possible with the Rio implementation. It does not work with an N7 as it only supports camera sharing between homogeneous Galaxy Nexus systems.

6 RELATED WORK

At a conceptual level, M2's goal of offering to apps, the semantic of one, capable system out of multiple, heterogeneous, less capable systems is similar to the goal of any distributed system, including distributed storage [21] and distributed shared memory [37]. M2 addresses the challenges specific to dynamic, composable integration of mobile devices. Unlike dynamic composable computing which provides mobile systems a more PC experience by replacing devices [72], M2 composes the devices of mobile systems together. For example, instead of replacing the small screen of a mobile system with a TV [72], M2 can additionally combine the displays of multiple mobile systems together into a better display when a TV is not available.

Various previous approaches have explored remote device access, though they do not address the important device heterogeneity and composition challenges solved by M2. Rio [4] provides mirroring for some devices, but has the performance, portability, and homogeneity limitations discussed in Section 5.3. Android HAL layer approaches, such as [32] which only supports sharing sensors, also suffer from the same platform and version specificity issues. None of these approaches support the degree of heterogeneity supported by M2.

Remote display systems, such as VNC [30], RDP [46], THINC [15, 34], GoToMyPC [19], and X [39, 75], introduce custom display commands to transport display content over the network, resulting in poor performance for display-intensive content such as 2D and 3D graphics [49, 76, 78]. Other approaches remote graphics by sending OpenGL commands [29], but require substantial network bandwidth and do not support the myriad of OpenGL extensions for mobile systems. These approaches do not work well for display sharing and do not support the broad range of devices available on mobile systems. Systems such as Apple AirPlay [10] and Google Chromecast [24] enable display mirroring using video encoding hardware similar to M2, but do not provide display mirroring between tablets and smartphones and lack support for other devices.

Some one-off apps exist for sharing a specific device. For example, mosaic app [23] allows a user to share an image across multiple displays. Similarly, Samsung Group Play [59] and MobiUS [66] only allow display sharing with a pre-recorded video that has to be downloaded on every device. Pinch [53], allows display sharing using only a Pinch-enabled app. Unlike M2, these apps are device-specific and do not enable other apps to access the shared device. Pursuits [69] and SideWays [80] track eye movements as input, but require a specific app and specialized hardware. M2 enables unmodified apps to transparently share and combine multiple devices and introduces new and powerful device transformations.

Sharing features such as clipboard, calls, messages, or switching views are being made available by Apple (Continuity) [13], Microsoft (Continuum) [44], various Android apps [60, 63], BlackBerry Blend [16], and Nintendo Switch [51]. These approaches often require the same OS to avoid the heterogeneity problem, and do not support general device sharing across multiple mobile systems. Similarly, Mobile Plus [52] mostly shares features, but not devices such as display or audio, by sharing IPC calls between systems. This does not work even across different Android versions as discussed in Section 3.2. Unlike M2, none of these systems support device transformations or multi-mobile functionality.

Various cloud-based approaches aggregate some device functionality across multiple mobile systems. For example, by creating new apps that obtain sensor data to predict earthquakes [35], creating sensor maps for temperatures or pressure [45], or a mobile resource sharing system [38]. These solutions only share a limited set of devices, without display, via cloud. M2 does not require any cloud infrastructure.

Approaches such as Recombinant Computing [22] envision creating generic broad interfaces for a limited set of shared resources, enabling developing new apps and scenarios such as using a mobile system to project or print a document. M2 takes a very different data centric approach and introduces a novel device transformation framework that works under existing unmodified mobile apps.

7 CONCLUSIONS

We have built and evaluated M2, the first system for heterogeneous, transparent multi-mobile computing. M2 makes three key contributions. First, by observing how mobile systems use higher-level abstractions and taller interfaces, M2 introduces a new user-level data-centric approach by exporting and importing device data instead of forwarding device related calls. This results in a system that is highly portable and can easily support different hardware devices and software platforms, as we demonstrate by operating M2 across heterogeneous hardware running multiple versions of Android and iOS. Second, M2 introduces device transformation to transparently mix and match disparate devices across different systems with heterogeneous data formats. This enables unmodified apps to share and combine devices in new and powerful ways such as multi-headed displays, Wii-like gaming, better audio quality teleconferencing, user interfaces for disabled users, and running Android apps on iOS systems. Finally, M2 leverages commodity mobile encoding and encryption hardware to enable a user-level remoting approach to provide qualitative performance over wireless networks similar to local device hardware even for 3D games.

8 ACKNOWLEDGMENTS

Roxana Geambasu and Mahadev Satyanarayanan provided helpful comments on earlier drafts of this paper. This work was supported in part by a Google Research Award, and NSF grants CNS-1717801 and CNS-1563555.

REFERENCES

[1] Naser AlDuaij, Alexander Van't Hof, and Jason Nieh. 2015. *M2: Multi-Mobile Computing.* Technical Report CUCS-005-15. Department of Computer Science, Columbia University.

[2] Naser AlDuaij, Alexander Van't Hof, and Jason Nieh. 2016. *Heterogeneous Multi-Mobile Computing.* Technical Report CUCS-008-16. Department of Computer Science, Columbia University.

[3] Naser AlDuaij, Alexander Van't Hof, and Jason Nieh. 2019. M2: Heterogeneous Multi-Mobile Computing. https://youtu.be/BzQ_YBA7kUU.

[4] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. Rio: A System Solution for Sharing I/O Between Mobile Systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2014).* Bretton Woods, NH, 259–272.

[5] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. Rio: A System Solution for Sharing I/O Between Mobile Systems. https://www.ruf.rice.edu/~mobile/rio.html.

[6] Jeremy Andrus, Naser AlDuaij, and Jason Nieh. 2017. Binary Compatible Graphics Support in Android for Running iOS Apps. In *Proceedings of the 2017 ACM/IFIP/USENIX International Middleware Conference (Middleware 2017).* Las Vegas, NV, 55–67.

[7] Jeremy Andrus, Alexander Van't Hof, Naser AlDuaij, Christoffer Dall, Nicolas Viennot, and Jason Nieh. 2014. 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 2014).* Salt Lake City, UT, 367–381.

[8] Apple Inc. 3D Touch - iOS - Apple Developer. https://developer.apple.com/ios/3d-touch/. Accessed: 2017-04-20.

[9] Apple, Inc. App Extensions - Apple Developer. https://developer.apple.com/app-extensions/. Accessed: 2018-03-21.

[10] Apple Inc. Apple - AirPlay - Play Content from iOS Devices on Apple TV. https://www.apple.com/airplay/. Accessed: 2014-12-07.

[11] Apple Inc. TV - Apple. https://www.apple.com/tv/. Accessed: 2018-08-07.

[12] Apple Inc. App Store - Apple. https://www.apple.com/ios/app-store/. Accessed: 2019-03-23.

[13] Apple Inc. 2019. Use Continuity to Connect Your Mac, iPhone, iPad, iPod touch, and Apple Watch. https://support.apple.com/en-us/HT204681.

[14] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys 2016).* London, UK, 19:1–17.

[15] Ricardo Baratto, Leonard Kim, and Jason Nieh. 2005. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005).* Brighton, UK, 277–290.

[16] BlackBerry. BlackBerry Blend - Desktop Software for BlackBerry. https://us.blackberry.com/software/desktop/blackberry-blend. Accessed: 2017-03-15.

[17] Blender Foundation. Big Buck Bunny. https://peach.blender.org/. Accessed: 2018-08-07.

[18] Cesanta Software. 2019. Mongoose Embedded Web Server Library. https://github.com/cesanta/mongoose.

[19] Citrix Systems, Inc. Remote Access | GoToMyPC. https://www.gotomypc.com/remote-access/. Accessed: 2015-02-12.

[20] CyanogenMod Open-Source Community. 2016. CyanogenMod. https://web.archive.org/web/20161224194030/https://www.cyanogenmod.org/.

[21] James C. Corbett and et al. 2012. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI 2012).* Hollywood, CA, 251–264.

[22] W. Keith Edwards, Mark W. Newman, Jana Sedivy, Trevor Smith, and Shahram Izadi. 2002. Challenge: Recombinant Computing and the Speakeasy Approach. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking (MobiCom 2002).* Atlanta, GA, 279–286.

[23] Darrell Etherington. 2013. Mosaic Lets You Weave A Single Display From Multiple iPhones And iPads, Offers SDK For Developers. https://techcrunch.com/2013/04/02/mosaic-lets-you-weave-a-single-display-from-multiple-iphones-and-ipads-offers-sdk-for-developers/.

[24] Google Inc. Chromecast - Google. https://www.google.com/chromecast. Accessed: 2017-04-20.

[25] Google Inc. Android Open Source Project. https://source.android.com/. Accessed: 2019-03-23.

[26] Google Inc. Google Play. https://play.google.com. Accessed: 2019-03-22.

[27] Alexander Van't Hof, Hani Jamjoom, Jason Nieh, and Dan Williams. 2015. Flux: Multi-Surface Computing in Android. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys 2015).* Bordeaux, France, 24:1–17.

[28] Alexander Van't Hof and Jason Nieh. 2019. AnDrone: Virtual Drone Computing in the Cloud. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 2019).* Dresden, Germany, 6:1–16.

[29] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. 2002. Chromium: A Stream-processing Framework for Interactive Rendering on Clusters. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2002).* San Antonio, TX, 693–702.

[30] Internet Engineering Task Force (IETF). 2011. RFC 6143 - The Remote Framebuffer Protocol. https://tools.ietf.org/html/rfc6143.

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

[32] Yu-Wen Jong, Pi-Cheng Hsiu, Sheng-Wei Cheng, and Tei-Wei Kuo. 2016. A Semantics-aware Design for Mounting Remote Sensors on Mobile Systems. In *Proceedings of the 53rd Annual Design Automation Conference (DAC 2016).* Austin, TX, 140:1–6.

[33] Lorenzo Keller, Anh Le, Blerim Cici, Hulya Seferoglu, Christina Fragouli, and Athina Markopoulou. 2012. MicroCast: Cooperative Video Streaming on Smartphones. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys 2012).* Low Wood Bay, Lake District, UK, 57–70.

[34] Joeng Kim, Ricardo Baratto, and Jason Nieh. 2006. pTHINC: A Thin-Client Architecture for Mobile Wireless Web. In *Proceedings of the 15th International World Wide Web Conference (WWW 2006).* Edinburgh, Scotland, 143–152.

[35] Qingkai Kong, Qin Lv, and Richard M. Allen. 2019. Earthquake Early Warning and Beyond: Systems Challenges in Smartphone-based Seismic Network. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications (HotMobile 2019).* Santa Cruz, CA, 57–62.

[36] Richard Lawler. 2018. iOS 12 Developer Beta Points to Bezel-Less iPad with Face ID. https://www.engadget.com/2018/08/02/ipad-pro-2-ios-12-beta-leak-bezel-faceid/.

[37] Kai Li and Paul Hudak. 1986. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing (PODC 1986).* Calgary, Alberta, Canada, 229–239.

[38] Yong Li and Wei Gao. 2017. Interconnecting Heterogeneous Devices in the Personal Mobile Cloud. In *Proceedings of the 36th IEEE Conference on Computer Communications (INFOCOM 2017).* Atlanta, GA, 1–9.

[39] The Linux Information Project. 2006. An Introduction to X by the Linux Information Project (LINFO). http://www.linfo.org/x.html.

[40] Alexander Löffler, Luciano Pica, Hilko Hoffmann, and Philipp Slusallek. 2012. 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) (ICAT/EGVE/EuroVR 2012).* Madrid, Spain, 37–44.

[41] Wenguang Mao, Jian He, and Lili Qiu. 2016. CAT: High-precision Acoustic Motion Tracking. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking (MobiCom 2016).* New York, NY, 69–81.

[42] James Alastair McLaughlin. 2018. Very Low Footprint JSON Parser Written in Portable ANSI C. https://github.com/udp/json-parser.

[43] Microsoft Corporation. Skype | Communication Tool for Free Calls and Chat. https://www.skype.com. Accessed: 2019-03-21.

[44] Microsoft Corporation. Windows Continuum for Windows 10 Phones and Mobile. https://www.microsoft.com/en-us/windows/continuum. Accessed: 2017-03-15.

[45] Microsoft Corporation. 2008. SenseWeb - Microsoft Research. https://www.microsoft.com/en-us/research/project/senseweb/.

[46] Microsoft Corporation. 2018. Remote Desktop Protocol (Windows). https://msdn.microsoft.com/en-us/library/aa383015.aspx.

[47] Monsoon Solutions, Inc. Monsoon Solutions | Printed Circuit Board Design & Manufacturing. https://www.msoon.com. Accessed: 2019-03-21.

[48] Sungwon Nam, Sachin Deshpande, Venkatram Vishwanath, Byungil Jeong, Luc Renambot, and Jason Leigh. 2010. Multi-application Inter-tile Synchronization on Ultra-high-resolution Display Walls. In *Proceedings of the 1st Annual ACM SIGMM Conference on Multimedia Systems (MMSys 2010).* Phoenix, AZ, 145–156.

[49] Jason Nieh and S. Jae Yang. 2000. Measuring the Multimedia Performance of Server-Based Computing. In *Proceedings of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video.* Chapel Hill, NC, 55–64.

[50] Jason Nieh, S. Jae Yang, and Naomi Novik. 2003. Measuring Thin-Client Performance Using Slow-Motion Benchmarking. *ACM Transactions on Computer Systems (TOCS)* 21, 1 (Feb. 2003), 87–115.

[51] Nintendo Co., Ltd. Nintendo Switch. https://www.nintendo.com/switch. Accessed: 2017-03-15.

[52] Sangeun Oh, Hyuck Yoo, Dae R. Jeong, Duc Hoang Bui, and Insik Shin. 2017. Mobile Plus: Multi-device Mobile Platform for Cross-device Functionality Sharing. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2017).* Niagara Falls, NY, 332–344.

[53] Takashi Ohta and Jun Tanaka. 2012. Pinch: An Interface That Relates Applications on Multiple Touch-screen by 'Pinching' Gesture. In *Proceedings of the 9th International Conference on Advances in Computer Entertainment (ACE 2012).* Kathmandu, Nepal, 320–335.

[54] OpenCV team. OpenCV library. https://www.opencv.org. Accessed: 2017-04-18.

[55] OpenSignal. 2015. Android Fragmentation Visualized. https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf.

[56] PassMark Software, Inc. PassMark PerformanceTest - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.passmark.pt_mobile. Accessed: 2015-03-10.

[57] Chunyi Peng, Guobin Shen, Yongguang Zhang, Yanlin Li, and Kun Tan. 2007. BeepBeep: A High Accuracy Acoustic Ranging System Using COTS Mobile Devices. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys 2007)*. Sydney, Australia, 1–14.

[58] Federico Perazzi, Alexander Sorkine-Hornung, Henning Zimmer, Peter Kaufmann, Oliver Wang, Sharon Watson, and Markus H. Gross. 2015. Panoramic Video from Unstructured Camera Arrays. *Computer Graphics Forum* 34, 2 (May 2015), 57–68.

[59] Pocketnow. 2013. Samsung Group Play Video Sharing Demo at IFA 2013. https://www.youtube.com/watch?v=hhvu9ugtVY4.

[60] Pushbullet. Pushbullet - SMS on PC - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.pushbullet.android. Accessed: 2017-03-15.

[61] Claire Reilly. 2018. Samsung's New Galaxy Phone Patent Is a Bezel-Less, Notch-Free Vision of the Future. https://www.cnet.com/news/samsung-galaxy-phone-patent-is-a-bezel-less-notch-free-slice-of-the-future/.

[62] Kay Römer. 2001. Time Synchronization in Ad Hoc Networks. In *Proceedings of the 2nd ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2001)*. Long Beach, CA, 173–182.

[63] Sand Studio. AirDroid: Remote Access and File - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.sand.airdroid. Accessed: 2017-03-15.

[64] Arne Schmitz, Ming Li, Volker Schönefeld, and Leif Kobbelt. 2010. Ad-Hoc Multi-Displays for Mobile Interactive Applications. In *Proceedings of the 31st Annual Conference of the European Association for Computer Graphics (Eurographics 2010)*. Norrköping, Sweden, 45–52.

[65] Securax LTD. Zoiper Audio Latency Benchmark - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.zoiper.audiolatency.app. Accessed: 2015-03-05.

[66] Guobin Shen, Yanlin Li, and Yongguang Zhang. 2007. MobiUS: Enable Together-viewing Video Experience Across Two Mobile Devices. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services (MobiSys 2007)*. San Juan, Puerto Rico, 30–42.

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

[68] Liz Stinson. 2017. What's the Big Deal With All These Bezel-Free Phones? https://www.wired.com/story/whats-the-big-deal-with-all-these-bezel-free-phones/.

[69] Mélodie Vidal, Andreas Bulling, and Hans Gellersen. 2013. Pursuits: Spontaneous Interaction with Displays Based on Smooth Pursuit Eye Movement and Moving Targets. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2013)*. Zurich, Switzerland, 439–448.

[70] VideoLAN Organization. VideoLAN - Official page for VLC media player. https://www.videolan.org. Accessed: 2016-05-10.

[71] Wei Wang, Alex X. Liu, and Ke Sun. 2016. Device-free Gesture Tracking Using Acoustic Signals. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking (MobiCom 2016)*. New York, NY, 82–94.

[72] Roy Want, Trevor Pering, Shivani Sud, and Barbara Rosario. 2008. Dynamic Composable Computing. In *Proceedings of the 9th Workshop on Mobile Computing Systems and Applications (HotMobile 2008)*. Napa Valley, CA, 17–21.

[73] Wi-Fi Alliance. Wi-Fi Direct | Wi-Fi Alliance. https://www.wi-fi.org/discover-wi-fi/wi-fi-direct. Accessed: 2019-03-23.

[74] Raymond Wong. 2018. If You Hate the iPhone X 'Notch,' These Phones Have Some Good News for You. https://mashable.com/2018/02/26/smartphones-true-bezel-less-displays/.

[75] X.Org Foundation. X.Org. https://www.x.org. Accessed: 2015-02-27.

[76] S. Jae Yang, Jason Nieh, Shilpa Krishnappa, Aparna Mohla, and Mahdi Sajjadpour. 2003. Web Browsing Performance of Wireless Thin-Client Computing. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*. Budapest, Hungary, 68–79.

[77] S. Jae Yang, Jason Nieh, and Naomi Novik. 2001. Measuring Thin-Client Performance Using Slow-Motion Benchmarking. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX ATC 2001)*. Boston, MA, 35–49.

[78] S. Jae Yang, Jason Nieh, Matt Selsky, and Nikhil Tiwari. 2002. The Performance of Remote Display Mechanisms for Thin-Client Computing. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX ATC 2002)*. Monterey, CA, 131–146.

[79] Katie Young. 2017. Digital Consumers Own 3.2 Connected Devices - GlobalWebIndex Blog. https://blog.globalwebindex.com/chart-of-the-day/digital-consumers-own-3-point-2-connected-devices/.

[80] Yanxia Zhang, Andreas Bulling, and Hans Gellersen. 2013. SideWays: A Gaze Interface for Spontaneous Interaction with Situated Displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2013)*. Paris, France, 851–860.