Tap: An App Framework for Dynamically Composable Mobile Systems

Naser AlDuaij
Department of Computer Science
Columbia University
New York, New York, USA
alduaij@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 have become ubiquitous, there is a growing demand for apps that can enable users to collaboratively use multiple mobile systems. We present Tap, a framework that makes it easy for users to dynamically compose collections of mobile systems and developers to write apps that make use of those impromptu collections. Tap users control the composition by simply tapping systems together for discovery and authentication. The physical interaction mimics and supports ephemeral user interactions without the need for tediously exchanging user contact information such as phone numbers or email addresses. Tapping triggers a simple NFC-based mechanism to exchange connectivity information and security credentials that works across heterogeneous networks and requires no user accounts or cloud infrastructure support. Tap makes it possible for apps to use existing mobile platform APIs across multiple mobile systems by virtualizing data sources so that local and remote data sources can be combined together upon tapping. Virtualized data sources can be hardware or software features, including media, clipboard, calendar events, and devices such as cameras and microphones. Leveraging existing mobile platform APIs makes it easy for developers to write apps that use hardware and software features across dynamically composed collections of mobile systems. We have implemented a Tap prototype that allows apps to make use of both unmodified Android and iOS systems. We have modified and implemented various apps using Tap to demonstrate that it is easy to use and can enable apps to provide powerful new functionality by leveraging multiple mobile systems. Our results show that Tap has good performance, even for high-bandwidth features, and is user and developer friendly.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile devices**; **Ubiquitous and mobile computing systems and tools**; **Ubiquitous and mobile computing design and evaluation methods**; • **Software and its engineering** → **Software libraries and repositories**; **Development frameworks and environments**; **Middleware**; **Operating systems**; **Peer-to-peer architectures**; **API languages**; • **Computer systems organization** → *Client-server architectures*; *Peer-to-peer architectures*; • **Networks** → *Mobile ad hoc networks*.

KEYWORDS

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

ACM Reference Format:
Naser AlDuaij and Jason Nieh. 2021. Tap: An App Framework for Dynamically Composable Mobile Systems. In *The 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '21), June 24–July 2, 2021, Virtual, WI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458864.3467678

MobiSys '21, June 24–July 2, 2021, Virtual, WI, USA

ACM ISBN 978-1-4503-8443-8/21/06…$15.00
https://doi.org/10.1145/3458864.3467678

1 INTRODUCTION

With mobile systems evermore ubiquitous, users rely on them as essential accessories of modern day life to socialize, share, and interact with other people. Individual users often own multiple mobile systems [84] and groups of users have many mobile systems at their disposal. There is a growing demand to provide users with a seamless experience across multiple mobile systems, not just use them as separate, individual systems. Our work on M2 [1] demonstrates various examples of useful functionality that can be achieved with multiple mobile systems working together, including combining multiple mobile systems into a mobile multi-headed display surface to provide a big screen experience for all users and turning a pair of mobile systems into a portable motion-based gaming console. We refer to the ability to combine the functionality of multiple mobile systems as *multi-mobile* computing.

Although multi-mobile computing has the potential to provide a wide range of powerful new app functionality, there are two key challenges that stand in the way of further adoption. First, there is no general, easy-to-use mechanism to connect impromptu collections of mobile systems that works across heterogeneous systems and networks. Even basic information sharing across mobile systems often requires users to tediously exchange phone numbers or email addresses, a burdensome process especially for the types of ephemeral interactions that occur among mobile users, not to mention that users may not want to reveal such personal information. Some approaches require users to login to cloud infrastructure and connect those cloud accounts together, an inconvenient process at best that requires network infrastructure and fails otherwise [41, 45]. Other approaches such as Apple's AirDrop [8] remove some of this burden to enable media sharing across mobile systems, but do not generalize to allow apps to go beyond this limited functionality to perform a wider range of tasks.

Second, there is a lack of support for app developers who want to create multi-mobile apps that operate across multiple mobile systems. This forces each app developer who wants to provide such functionality to start from scratch, incurring the same recurring development costs and making development difficult and error prone at best. Each developer may come up with ad hoc approaches and conflicting user interfaces, resulting in unexpected app interactions and an inferior user experience. A key challenge is how to provide a general way for apps to programmatically connect hardware and software features across impromptu collections of mobile systems.

To address these problems to enable multi-mobile computing for the masses, we introduce Tap. Tap is an app framework that enables users to make use of impromptu and dynamically composable collections of mobile systems, and enables app developers to create apps that can take advantage of these collections. Tap introduces two key ideas. First, Tap provides a novel connectivity mechanism that combines tapping and NFC to enable connectivity across multiple different available communication mediums, even in the absence of Internet access. Second, Tap introduces data source virtualization which allows apps to access multiple remote software and hardware features across heterogeneous platforms via existing, familiar platform APIs.

Tap makes it easy to connect mobile systems together by leveraging a familiar user interaction paradigm, tapping a mobile system, which is increasingly used for mobile payments and public transportation [26, 42, 47]. Tap repurposes it to control a novel mechanism to enable connectivity across multiple heterogeneous systems operating across heterogeneous networks. Users simply tap their mobile systems together to allow them to be used in combination with one another. Tap leverages widely available NFC technology deployed on mobile systems to detect proximity of mobile systems as a result of a tap and then exchange connectivity information and security credentials. Unlike other approaches such as Bump [17], Tap is an entirely localized mechanism, making it possible to establish network connectivity across systems without any need to sign into user accounts or rely on cloud infrastructure or certificate authorities. The NFC information exchange includes local IP addresses which are then used to establish a high-bandwidth IP-based connection between systems via WiFi, cellular, or WiFi Direct [81] in a manner that works robustly for systems on completely different networks and in the presence of NATs and firewalls. Using Tap, mobile systems automatically connect after a tap and can then be used by apps collectively as though they were one, enabling quick and easy-to-use ephemeral interactions among users and their mobile systems.

Tap makes it easy to create multi-mobile apps by leveraging and repurposing existing mobile APIs already familiar to app developers such that the same APIs can be used across multiple mobile systems. This is made possible by introducing a novel idea, data source virtualization. We observe that existing mobile APIs provide ways to specify data sources to be used with the API. For example, using the camera API, there is a source identifier to specify whether it is the front or back camera that is requested. Tap introduces data source virtualization so that remote features on other mobile systems can be referenced locally via virtual data sources. Virtual data sources can be used in the same manner as any other data sources, making it possible to reuse existing APIs with remote features. Data source virtualization is accomplished using a data-centric approach to import and export the data associated with hardware and software features, avoiding the need to bridge API differences across heterogeneous systems. For example, a developer writing an Android app only needs to know about Android APIs but can still access hardware and software features on remote iOS systems. This is made possible because the data associated with features can be represented in standard formats portable across different systems.

Unlike prior systems and platform approaches, Tap is designed to support multi-mobile apps on stock mobile platforms without requiring any modifications. All necessary functionality can be encapsulated in a cross-platform framework that app developers can use in creating multi-mobile apps; the mobile apps can then simply be downloaded from the respective app store. This makes it easy to deploy multi-mobile apps as getting a multi-mobile app is no different than downloading any other app from the respective app store. In the same manner, Tap is designed to work across heterogeneous collections of mobile systems with different hardware and software versions based on the same principles that provide portability for existing non-multi-mobile apps.

We have implemented a Tap prototype for Android and iOS that allows apps to make use of multiple mobile systems with different hardware and software versions. Our experimental results show that Tap is easy to use and provides fast connectivity and performance among systems across heterogeneous networks. Writing a Tap app is similar to writing any other mobile app and only requires a one line import of the Tap framework. We have used Tap to build and install several multi-mobile apps, including a popular sword fighting game that can leverage multiple mobile systems to control the game via realistic sword movements instead of unintuitive touchscreen gestures, a music player that can leverage the audio output of multiple mobile systems to provide higher fidelity surround sound, a multi-channel audio recorder that leverages microphones across multiple mobile systems to provide higher-fidelity audio recording, a new group snap feature for Snapchat users across multiple mobile systems, and a cross-platform photo gallery app that makes it trivial to share photos among authenticated mobile systems with a simple select, tap, and swipe gesture. These various apps show that connecting mobile systems is easy for users to do, and building apps that leverage the Tap framework is straightforward and can provide powerful new multi-mobile computing functionality across unmodified Android and iOS systems.

2 USAGE MODEL

Tap is easy to use. Users simply download and install Tap-enabled apps from the respective app store and run them. The respective app is run on each mobile system that users would like to have interact using Tap. Figure 1 shows how to use a Tap app. A Tap app will have one or more app-defined tasks that are enabled by tapping systems together. Each task can involve sharing hardware and software features across systems. When the systems are tapped together and therefore connected, a notification is typically displayed to the user to indicate what systems are connected and what task is being performed. When the task completes, the systems are automatically disconnected. Note that Tap also supports contactless interactions and works when systems are just a few centimeters apart, for users

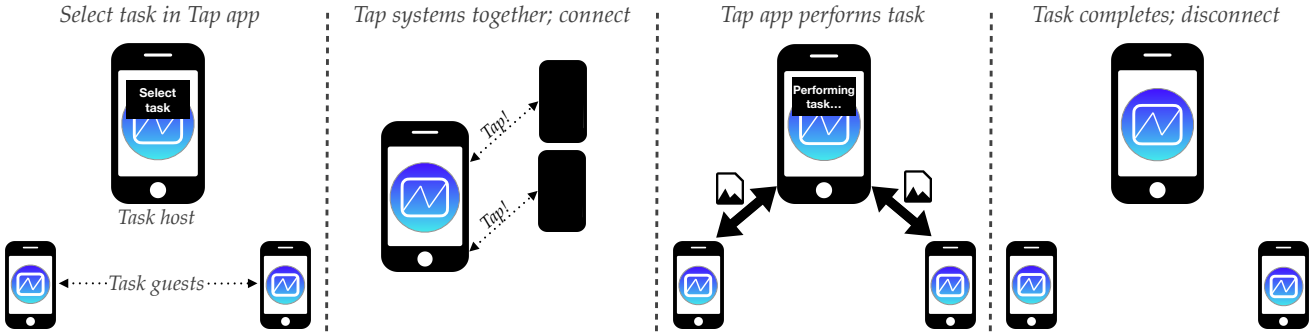

Figure 1: How to use a Tap app

who may want to avoid physically touching mobile systems together due to COVID-19 [19, 73, 78]. It also enables users to share features rather than physically sharing a system with other users.

Using photo sharing as a simple example, suppose user A wants to share a photo with user B. For simplicity, let's assume both users are using Android smartphones, which have the Android Gallery app for browsing photos. The Gallery app has a Share button that brings down a drop down list for ways to share photos. Both users would run the Gallery app on their systems. Assuming the Gallery app has been enabled to use Tap, user A would select a photo and click the Share button, then select the option to share via Tap. Users A and B would then tap their systems together, causing the photo to be shared from user A's Gallery app to user B's Gallery app. Note that there is no need for the app to show a user a list of systems or for the user to manually select from a list of systems; physically tapping systems together automatically connects them. Once the photo has been delivered to user B's smartphone, the systems are automatically disconnected. Although we use photo sharing as a simple example, Tap can be used to allow an app on one system to access a wide range of remote hardware and software features on other systems, such as clipboard, camera, microphone, display, and sensors.

With Tap, an app-defined task is the granularity at which systems are connected. A task can be long or short, though we expect most tasks to be short in practice. While alternative design choices could be made, such as having systems be connected until users explicitly disconnect, we wanted to avoid complicating the usage model by focusing users on the task they want to accomplish as opposed to keeping track of connectivity and requiring users to explicitly disconnect. We also wanted to enable developers to select the appropriate granularity best suited to the respective app and its functionality. A potential downside of this approach is that for some apps, there may be multiple tasks, each of which would require tapping systems together to accomplish the task. However, we believe that app developers who retain control over what constitutes a task are best suited to defining the appropriate granularity that makes the most sense for their respective apps, and Tap's goal is to empower those app developers. Furthermore, it is important to note that a task need not correspond to a single hardware or software feature. For example, for the Gallery app photo sharing example, a user could select multiple photos to share on a tap, not just one.

Tap allows multiple mobile systems to be involved in an app-defined task by tapping them together. We refer to the set of systems that have been tapped together to perform an app-defined task as a *collection*. Tap enables chaining, which allows any system in the collection to tap with another mobile system to include that system in the collection as well.

Tap apps are regular mobile apps and must obey all the same user and security permission settings already available on mobile platforms such as Android. For simplicity, Tap does not currently override any existing permissions. Any permissions granted for local features are granted similarly for remote features. For example, if a user denies a Tap app permission to access the local camera, the Tap app will not be able to share the local camera since these permissions are controlled by the platform. However, this does not prohibit the Tap app from accessing a remote camera as long as the app is permitted to access the network and the app located on the remote system has permission to access the camera on that system. This model is no different than non-Tap apps; once a user gives an app local access to a feature, the user has no control over whether that feature is shared remotely if network permissions have been granted. For example, Zoom [85] allows a user to share their local camera by only allowing local camera access to the app. Tap, therefore, does not increase security risks beyond non-Tap apps. Adopting a more fine grained permission model would require platform changes so Tap adopts a simpler permission model to work across unmodified mobile platforms.

3 DEVELOPER API

App developers make use of the Tap API to program tasks in apps to make use of features across multiple mobile systems. The Tap API is available as a downloadable SDK that developers can use in their apps and release in the respective app store. The API is designed to be similar to existing mobile platform APIs to make it easy for developers to use. For simplicity, we describe the API as it would be used by Android app developers; similar methods would be used by iOS app developers.

The Tap API enables developers to define app-specific tasks to express how hardware and software features are shared across a collection of mobile systems. A *task* is a series of operations that can make use of one or more features. Tap defines tasks in a master-slave model in which one system, the *task host*, is responsible for controlling and coordinating tasks while other systems that are part of the collection, *task guests*, follow the direction of the task host. Task hosts generally use remote features and task guests generally provide those features for use. Task guests only connect directly to the host, not directly to each other, as they are merely participants in

Feature	System service/level	Abstraction	Interface	Number of streams	Local	Remote N==1	Error report	LOC
Sensor	SensorService	Event type (int32)	Callback	1 sensor event stream	0-26	27-53	No update	109
Input	InputFlinger	Source type (int32)	Callback	1 input event stream	0-4	5-9	No update	125
Location	LocationMgrSvc	Provider name (string)	Callback	1 location data stream	"gps"	"gps-TP1"	No update	272
Microphone	AudioFlinger	Source type (int32)	Method	Per source	0-10	11-21	Return error	376
Camera	CameraService	Camera ID (int32)	Method	One camera at a time	0-1	2-3	Callback	1817
Audio	AudioFlinger	Implicit (int32)	Method	Mixes audio streams	0-3	4-7	Return error	147
Display	SurfaceFlinger	Surface name (string)	Method	Per surface	App based	Append "-TP1"	Callback	2105
Clipboard	ClipboardService	Label/item (string)	Method	System wide buffer	App based	Append "-TP1"	Exception	153
Notification	NotificationMgrSvc	Tag (string)	Method	App initiated	App based	Append "-TP1"	Callback	239
Intent	Activity/Context	Data (URI string)	Method	App initiated	App based	Append "-TP1"	Exception	312
Calendar	ContentResolver	ID (URI string)	Method	System wide database	"calendar"	"calendar-TP1"	Return error	142
Contacts	ContentResolver	ID (URI string)	Method	System wide database	"contacts"	"contacts-TP1"	Return error	421
Media	ContentResolver	ID (URI string)	Method	System wide database	"media"	"media-TP1"	Return error	627

Table 1: Tap features on Android

the dynamic collection controlled by the host. An app may designate different task hosts for different tasks, but each task has only one task host. The master-slave model simplifies the programming paradigm for the developer by enabling centralized control and coordination of a task instead of having to manage more complex distributed control and coordination, especially given that tasks are typically short-lived. The role of a host versus a guest does not define the data flow or how much work the guest does compared to the host, only that the host initiates the task and controls it, deciding what it does, its duration, and how the guests participate.

To initialize an app to use Tap, Tap provides a simple API, `setSystemListener` as shown in Listing 1, to register an app-defined callback class and indicate whether it is a task host or a task guest:

```
boolean setSystemListener(TapSysListener listener,
                boolean taskHost);
int getSource(System sys, String feature, int local);
String getSource(System sys, String feature, String local);
```

Listing 1: Tap API

Registering this class initializes the underlying Tap framework code, which is executed in response to taps to establish a successful connection. Only when Tap successfully connects two systems together, the callback is called to inform the app of the availability of another system in the collection. The callback provides a read-only Tap `System` object that contains information about the connected system, such as its unique ID, available features, and feature options.

Once systems are discovered and connected via Tap and the app is informed via the callback method, features can be shared across the connected collection of systems. Tap defines a *feature* as a set of developer public APIs that provide access to software or hardware user-facing data, such as device location, media, sensors, or clipboard. These are high-level public developer APIs in Java for Android and Objective-C or Swift for iOS. Tap focuses on making features available across multiple systems that relate to receiving and sending user-facing data. It does not provide multi-system functionality for other generic app APIs, programming constructs, or helper APIs such as the settings APIs. Tap does not share generic system resources unrelated to user-facing data, such as CPU, WiFi, or Bluetooth. Table 1 lists the features Tap supports in Android.

Leveraging existing APIs for accessing remote features allows developers to avoid a steep learning curve and to easily convert apps to Tap-aware apps. Using Tap, developers need to simply include Tap package classes instead of Android or iOS classes or header files. Tap introduces data source virtualization, described in Section 4, to enable reusing existing APIs for remote features.

As shown in Listing 1, Tap provides an API, `getSource`, to return virtualized data source identifiers that works for both string and integer identifiers. It takes a `System` object referring to a remote system, a feature name, and the equivalent local type, and returns the virtualized remote type. Tap reuses existing methods for reporting errors with local data sources to report errors with remote data sources as well. For example, if a remote system disconnects, its respective feature would be unavailable and Tap would simply reuse Android's existing and available error reporting functionalities for the feature to inform the app of failures. Table 1 lists the data source abstractions for various hardware and software features on Android.

To illustrate how the Tap API works, we provide an example of an app requesting remote sensor data. First, we show how local sensors are accessed on Android:

```
int S1 = Sensor.TYPE_LINEAR_ACCELERATION; // Accelerometer
int d = SensorManager.SENSOR_DELAY_NORMAL;
SensorManager mSMgr = getSystemService(SENSOR_SERVICE);
mSMgr.registerListener(this, mSMgr.getDefaultSensor(S1), d);
```

Listing 2: Registering for local Android sensors

Listing 2 shows the sensor service is requested first and the sensor source type, `S1`, is defined as the accelerometer. The app then registers its class, which is of type `SensorEventListener`, to receive accelerometer sensor data at a normal rate. The app overrides the `onSensorChanged` method, shown in Listing 3:

```
@Override // Override onSensorChanged with our own code
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == S1)
      // accelerometer, data in event.values[]
}
```

Listing 3: Receiving sensor data

This callback method gets called by Android to provide the app with sensor events. In this case, it will only provide the accelerometer data. The sensor type can be accessed through the provided event argument and is checked against the equivalent type. Using `unreg`

isterListener, the app can unregister from the sensor service to stop receiving updates.

For an app to use Tap to receive remote accelerometer data instead, the code in Listing 2 is changed to:

```
// Listen on connectivity+update sys (a List of System objs)
Tap.setSystemListener(localListener, /*host*/true);
// After connecting to a remote system:
int S1 = getSource(sys.get(0), "Sensors",
                   Sensor.TYPE_LINEAR_ACCELERATION);
int d = SensorManager.SENSOR_DELAY_NORMAL;
SensorManager mSMgr = getSystemService(SENSOR_SERVICE);
mSMgr.registerListener(this, mSMgr.getDefaultSensor(S1), d);
```

Listing 4: Registering for remote sensors using Tap

Listing 4 shows the app registering for connectivity updates and hosting a task using the Tap API. The `localListener` is implemented by the app for when systems connect, disconnect, or get updated. The connected status is only reported to the app when a successful tap occurs and the system is fully connected. A disconnected status is reported to the app if the system is no longer connected. These are reported via Tap app callbacks, allowing an app to receive updates on what systems are available for use. Tap guests can now connect to this system. These guests are tracked by an app-defined global list variable, `sys`. Once the app discovers a connected system, it gets the source type of the sensor by calling Tap's `getSource` method on the remote `System` object. This method retrieves the virtualized data source identifier for the remote feature type. Similar to the local sensors code, the app then registers to listen for the remote system sensor updates. If `getDefaultSensor` is called before a Tap app registers a listener, it will simply return the same sensor used by the local system.

The app then overrides the `onSensorChanged` method, resulting in the same exact code as requesting the local sensor shown in Listing 3, with `S1` referring to remote sensor data. The app can then use the sensor data. The app can stop receiving remote sensor data by using the same exact Android method to stop receiving sensor updates from local sensors, `unregisterListener`. Alternatively, the app can disconnect all systems and destroy the task by passing `null` to the `setSystemListener` method. For guests trying to disconnect, setting `setSystemListener` to `null` will suffice as well.

4 ARCHITECTURE

To support the Tap usage model and developer API without changes to existing mobile platforms, Tap provides a novel system architecture encapsulated in a set of libraries used by apps that includes two key components, a connectivity mechanism to facilitate sharing features and a data source virtualization mechanism to access and utilize these shared features. Figure 2 shows an overview of the Tap architecture on Android.

Connectivity Mechanism Tap provides a mechanism to connect mobile systems together to share features and to meet several key requirements. First, it must be easy-to-use. Second, it must be quick to support common ephemeral user interactions among users who may not know each other's contact information. Third, it should protect user privacy; users should not receive data they do not want to receive, be discoverable when they do not want to be, or require

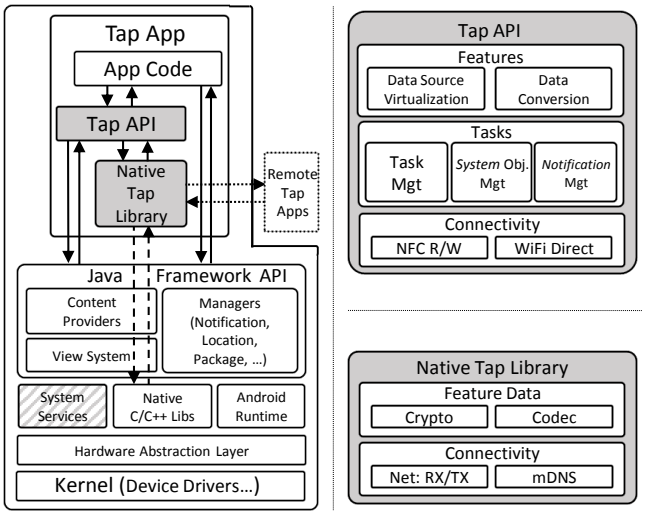

Figure 2: Tap architecture; Tap components in solid grey

their data to go through third-party cloud infrastructure to be potentially collected. Fourth, it must work across multiple heterogeneous systems connected across heterogeneous networks, even with firewalls and NATs. Fifth, it should work even when there is limited or no network infrastructure. Finally, it should provide a secure and high-performance network communication path to support accessing bandwidth-intensive data sources.

Various approaches exist that address some but not all of these requirements. For example, Bump [17] provided an app to share contacts and files by bumping systems together, but required use of and routed user data through cloud infrastructure to work, risking user privacy. Bump was eventually acquired by Google. Google's Android Beam [33] leveraged Bump's bumping mechanism together with NFC for discovery, but suffered from poor performance due to being limited to using Bluetooth for data communications. Bump and Beam could only be used to connect two systems together at a time. Apple's AirDrop uses Bluetooth and WiFi Direct to provide a localized high bandwidth connection, but is known to suffer from usability issues [23, 46, 63, 75]. It either requires users to sift through preexisting contacts or set up new ones to work, a burdensome process for ephemeral user interactions, or allow anyone within a large radius to send them potentially undesirable content, violating user privacy. All of these approaches are limited to simple contact or file sharing; none of them provide a general mechanism for apps to use for data communications across mobile systems.

To meet all of the connectivity requirements, Tap leverages an NFC-based mechanism triggered by tapping systems together, but uses it to connect systems via an IP-based data communication mechanism that takes advantage of the multi-network capabilities of mobile systems. NFC is used to exchange local IP addresses which are then used to establish a high-bandwidth IP-based connection between systems via WiFi, cellular, or WiFi Direct in a manner that works robustly for systems on completely different networks and in the presence of NATs and firewalls. The simple tapping interaction is quick and easy-to-use, and protects user privacy by only allowing discovery for systems tapped together and avoiding the need to exchange phone numbers, email addresses, or transmit user

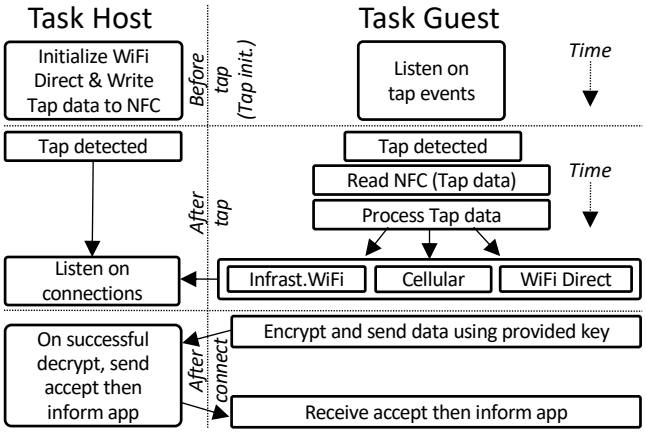

Figure 3: Tap host/guest tap and connectivity flow

data through the cloud. The IP-based communication mechanism provides a high-bandwidth communication path that works across heterogeneous systems and networks, even in the absence of network infrastructure by falling back to WiFi Direct availability. This combination of NFC with IP-based technologies provides a significant advantage over existing solutions to provide a cross-platform, robust, and user-friendly way to connect systems for feature sharing even in the absence of network infrastructure.

Figure 3 and Listing 5 show how Tap combines NFC for discovery, authentication, and security, and traditional IP wireless networking for data communication. Each system sends via NFC its infrastructure WiFi IP address, cellular IP address, its public key for a self-generated key pair, and app ID. The task host also sends temporary and randomized potential WiFi Direct network credentials, namely an SSID/password pair. The systems then use that information to connect via IP networking for data communication through all three potential network paths concurrently, accepting the highest priority successful connection and saving the host provided symmetric key for potentially encrypted feature data. The default priority ordering used mirrors how smartphones use network infrastructure today, providing a familiar user experience and taking advantage of what is usually the fastest and most available medium on mobile systems. Infrastructure WiFi is prioritized first, cellular is second, and WiFi Direct is last, the latter typically taking the longest to establish connectivity and having the most limited range. Lower priority networks will connect if timeouts are exceeded for higher priority network connections, currently 300 ms for infrastructure WiFi, and an additional 300 ms for cellular to provide reasonable responsiveness with sufficient time to allow for connections for each medium. Quick connectivity is important, hence the focus on faster connectivity methods such as infrastructure WiFi and cellular instead of WiFi Direct, since many Tap tasks are ephemeral. Tap provides a modular design for connectivity, so that other mediums such as Bluetooth can be added, mediums can be prioritized differently, and future technologies can be incorporated into Tap.

Once a connection is established and verified with the guest, the task host shares its self-generated symmetric key with the guest by encrypting it using the guest public key. This shared symmetric key is used for encrypting feature data only once for all guests in the task. If a guest is removed from a task session, the host generates a new symmetric key that is only sent to the remaining guests, encrypted using their public keys. If the connection later fails, Tap will actively try all mediums once again. If it still fails, the tap is deemed a failure and the Tap apps are informed. Since a task host binds to different network mediums, additional guests can connect through different mediums on the same active task. This is possible since Tap apps are decoupled from networking. No other existing approach provides this combination of multiple mediums, medium offloading, and secure security credential exchange.

Tap allows chaining where one guest taps with a task guest already connected to a task host to join that task. Once guests connect to the host, they switch to an advertising mode similar to the host to allow guests to connect to the main task host via chaining. Since the task guest already has the NFC data from the host, it exchanges it via NFC with the new guest and relays the public key and connectivity information of the new guest to the host through its IP connection. The new guest is then able to use the same task host mechanism described.

```
// TAP TASK GUEST:
main thread at t=0ms:
  spawn medium threads:
    infrastructure WiFi, cellular, WiFi Direct
  wait on medium threads or t>5s
  if t>5s then
    end all medium threads, notify user and return fail
  else if signaled with a connected medium then
    notify user and return success

infrastructure WiFi thread:
  if connected, then goto guest check
cellular thread:
  if connected, wait until t=300ms then goto guest check
WiFi Direct thread:
  if connected, wait until t=600ms then goto guest check
guest check:
  1. end all other medium threads and their connections
  2. encrypt session communication via NFC exchanged keys
  3. send encrypted data to host
  4. wait on symmetric key (decrypt with guest private key)
  5. signal main thread

// TAP TASK HOST:
main thread at t=0ms:
  spawn medium threads:
    infrastructure WiFi, cellular, WiFi Direct
  wait on medium threads or t>5s
  if t>5s || signaled with no connection then
    end all medium threads, notify user, and return fail
  else if signaled with a connected medium then
    notify user and return success

infrastructure WiFi, cellular, WiFi Direct threads:
  listen on connection and if connected then goto host check
host check:
  1. wait to receive encrypted data from guest
  2. decrypt+verify data, otherwise signal main thread
  3. send symmetric key (encrypted with guest public key)
  4. end all other medium threads and their connections
  5. signal main thread
```

Listing 5: Tap connectivity

Limitations arise with network address translation (NAT) across different networks. In these cases, a Tap system may fail to reach another Tap system. If one system is behind a NAT and the other is not, the two systems will not be able to connect using the exchanged IP addresses. This issue is mitigated though in practice for several reasons. First, Tap is being used to connect physically local systems, so it is commonly the case that both systems may be on the same infrastructure WiFi network if such a network is available. In that case, both systems are likely to be on the same subnet, NAT or no NAT, and the local IP addresses will likely enable a successful connection. Second, the vast majority of cellular networks in developed countries use non-NATed IPv6 [40], which will facilitate a connection using the exchanged IP addresses. Finally, locally provided WiFi Direct is always a fallback in case other mediums fail. A simple connectivity test is used to check if the network is reachable followed by heartbeats to ensure that the systems are still connected. Tap does not resend data on a reconnection.

Data Source Virtualization Mechanism Tap's goal is to allow remote features to be used the same way as local features without API or system changes. This provides two advantages: First, app developers can reuse the same existing APIs to access both local and remote features, minimizing development effort and cost. Second, encapsulating Tap within apps allows for easier deployment for both app developers and users.

To accomplish this goal, Tap introduces a new idea, *data source virtualization*. We observe that many features provide the ability to select the data source for the feature. For example, the camera feature provides a way to select the back camera or the front camera as the data source for the object representing a camera. Each data source has an associated identifier, typically either an integer or a string representing a name. For example, given a system with back and front cameras, the camera feature data source identifiers are 0 and 1, respectively. Although any valid integer can be an identifier, each feature makes use of only a limited range of identifiers; for example, the camera feature only needs as many integers as there are cameras available. Tap leverages this observation by segmenting the data source identifier namespace for the respective feature and mapping remote features to separate segmented portions of the namespace that are unused, effectively providing virtual identifiers for remote features. Remote features can then be referenced using these virtual identifiers in the same way as local features, the only difference being that the virtual identifiers map to a different range of the data source namespace. Continuing with the camera example, if a remote system has back and front cameras identified on the remote system using respective identifiers 0 and 1, the remote cameras can now be referenced on the local system in an independent virtual namespace with the unused integer range 2 to 3. Because these ranges and naming are small and limited, Tap can create segments of data source ranges for a large number of remote systems. Tap effectively expands data source namespaces into a set of independent virtual namespaces that can be accessed as seamlessly as the original local namespaces.

To expand the local data source namespace to include sets of independent virtual namespaces for each feature, Tap overrides and intercepts feature related API calls and examines the respective data source. If the data source identifier refers to a local namespace, such as 0 or 1 for cameras, it is sent to the local framework as usual. If the data source refers to a remote namespace, such as 2 or 3 for a remote system's cameras, Tap initiates communication with the remote system to access the remote feature. If the app then starts the camera preview or takes a picture, the Tap library redirects these requests to retrieve remote data and feed it to the Tap app. Note that the Tap app uses the same exact API for both local and remote cameras, but in the case of the latter, it passes in a data source identifier associated with a remote virtual namespace when initializing the camera object. Only a small subset of feature APIs are intercepted to be able to support data source virtualization in Tap. Table 1 lists the namespace virtualization that is done for various hardware and software features on Android.

Tap's novel data source virtualization mechanism must address four key issues. First, there must not be any virtual namespace inconsistencies or conflicts. Second, network intermittence or failures must not cause apps to behave badly since features are generally designed to run locally, not remotely. Third, features must be usable across heterogeneous systems even though they may have different APIs. Finally, users and their experience should not be adversely affected using remote instead of local features, so performance is crucial.

To provide naming consistency and avoid virtual namespace conflicts, Tap provides a unique system identifier for each connected remote system which it uses to determine the range for each virtual namespace. For example, the data source for the local sensor feature is an integer numbered 0 to 26 representing an event type. Tap uses the remote system identifier to shift the range of the data source identifiers for each remote system. Table 1 shows that for a remote system with identifier N=1, its sensor data sources are numbered 27 to 53. If N=2 instead, sensor data sources would be numbered 54 to 80, and so forth. Using `getSource` API method discussed in Section 3, apps can pass in the intended feature type for the equivalent local data source along with the remote system identifier. The method returns the data source for the specific feature on the specified remote system which can then be used with existing feature APIs. Note that the conflict-free namespace ranges are tracked internally and not exposed to developers; developers simply use the same feature APIs for both local and remote features without needing to be aware of the specific virtual namespace mappings.

To adapt to the dynamic nature of mobile systems and intermittent connectivity, Tap reuses existing feature API options to report failures. Existing feature APIs provide a way of informing apps of any failures with data sources, as shown in Table 1. Tap observes that errors are reported when local features are unavailable or inaccessible through a specific return value or a callback. Tap utilizes the same error reporting by applying and extending the reported errors to also cover remote data source errors and disconnections.

To allow local feature APIs to be used even when remote systems use different APIs, we make two observations. First, mobile systems are highly vertically integrated, and each system has its own system-specific APIs. These APIs are often nonstandard and incompatible with other systems making heterogeneity difficult to support by forwarding or remotely calling these APIs, especially in the context of lower-level hardware features. Second, although mobile APIs are often system-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. Leveraging these observations, Tap makes no attempt to match APIs

across different systems, but instead imports and exports data to and from each mobile system using common cross-platform data formats, avoiding the need to bridge incompatible APIs. In other words, each system converts its data sources into an intermediate cross-platform format when exporting that data from one system to another, and imports data from that intermediate format into its own APIs.

Tap intercepts the API methods used for accessing feature data to control data flow, formats, and duration of sharing. There are generally three main methods: a read method for some features to read feature data from the system, a write method for some features to write feature data to the system, and a callback interface for some features to asynchronously provide apps with feature data. Tap simply intercepts these three types of methods and converts the data into a common format in a data conversion layer, as shown in Figure 2.

A potential disadvantage of interposing on the public developer API layer is the complexity of the API. If the API is complex and many methods must be intercepted at which to import and export feature data, and the volume of data that must be manipulated is high, the result can be too complex and inefficient. Our experience with the software and hardware features for mobile systems indicates that these disadvantages do not exist for most software and hardware features, except for OpenGL, as discussed in Section 5.

Because the feature data is in well-known common formats, Tap can use hardware acceleration mechanisms available to apps to manipulate feature data to aid with importing and exporting feature data in an efficient manner. For example, Tap uses real-time hardware video and audio encoding to compress display and audio data before transmitting it. Tap uses the commonly available H.264 video encoding and AAC audio encoding for display and audio devices, respectively, though other encoding formats can be used. These encoders can be dynamically configured to use different resolutions, bit rates, and frame rates; Tap by default uses 30 frames per second (fps) frame rates since they are visually indistinguishable from higher frame rates for end users [72]. In the case of camera, Tap 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. This type of feature data approach shows that the performance is indistinguishable even with respect to high-bandwidth features such as display, camera, and audio [1].

For control messages and features which expect lossless data, Tap uses TCP. For streamed features such as display, audio, and camera preview, UDP is used. Data that is late or not delivered due to packet loss is discarded. Timestamps are combined with NTP and best practices to ensure media synchronization across systems [51, 56, 64, 70].

5 IMPLEMENTATION

We implemented Tap for Android and iOS. The Tap API SDK was in Java for Android and Objective-C for iOS, and also supported by a native Tap library in C. Table 1 lists the lines of code for each supported feature across both the SDK and native library for Android; iOS details are omitted due to space constraints. Most of the native library code consists of networking, crypto, codec, mDNS, and configuration code. The code for the library is in C and was easily portable.

To support NFC, Tap uses Host Card Emulation [30] and Reader [32] modes on Android and the NFC ISO 7816 [7] implementation on iOS. Android and iOS share data over NFC using the Application Protocol Data Unit (APDU). Currently, platforms require apps to be in the foreground to receive NFC events. Unlike Android, the default behavior on iOS when registering for NFC events forces a user prompt. Since task guests need to listen on NFC, we only register for NFC on iOS when we detect a magnetometer sensor spike, indicating a potential task host nearby. Immediately after the spike, Tap listens on NFC events and the shown prompt is dismissed by Tap after the data exchange, after a time out, or by the user.

To support feature sharing, Tap reuses the existing interfaces provided by the mobile platform and interposes on some of these existing APIs. Data sources of API calls are examined to route requests and data from and to relevant systems. Android apps using Java Native Interface (JNI) for features also include data sources that can be virtualized. Tap adds a virtualized data source field for some iOS feature APIs without a data source argument. Figure 2 shows the local and remote data flow between a Tap app to a Tap library. Table 1 lists the interface for each feature. For callbacks, the local Tap library requests the data; once received, it is unpacked, converted for the local platform, and the app's callback is fired by Tap with the data. For write methods, Tap forwards the data from the app to the remote Tap library, where it converts it and applies it to its system. For read methods, the Tap library requests the data from the remote system, which gets forwarded in a common format to be received, converted, and provided to the Tap app. Note that Tap adds an argument to the input feature method for a data source and also revives a deprecated `setRouting` API for setting audio source. Android does not provide error reporting for callbacks since the app never blocks. However, disconnections are reported through Tap callback API instead.

To support camera preview, the passed `Surface`, a graphical frame buffer, is examined for a remote source. If found, it encodes the remote camera preview and forwards the frames. The preview `Surface` is provided to the decoder to receive, decode, and display the encoded frames from the remote camera preview. Finally, to support display, Tap reimplements the Android native C `setName` method for `Surface` and adds it to its Java equivalent class. `Surface` related APIs are overloaded and examined for a remote source to forward the data.

Tap provides support for remoting most display related APIs, such as playing a video and recording a view. Tap apps may also locally use OpenGL. However, without platform modifications, Tap does not support writing frames to remote displays using OpenGL [74] commands. Previous approaches to remoting OpenGL have forwarded OpenGL commands or data, both of which are very inefficient and not easily portable [3, 5]. By modifying the platform, Tap can solve this problem by changing the display system service, `SurfaceFlinger` in Android, to record display surfaces and forward the encoded display frames instead of OpenGL commands. This change is only required in case a Tap developer wants to write frames using OpenGL to a remote display with the Tap API, and not when OpenGL is locally used. If there is demand to write frames remotely using OpenGL, we would envision that mobile platforms could include this modification.

6 EVALUATION

To demonstrate the effectiveness of Tap, we measured its performance, implemented new multi-mobile apps and modified existing apps to be multi-mobile aware using Tap, and conducted various usability studies. We show Tap's cross-platform functionality by

Network configuration (host/guest):	P3a/P3a	P3a/iPh
NAT WiFi (same network)	14	11
Public WiFi (same network)	14	20
NAT WiFi to public WiFi	19	12
NAT WiFi to cellular	96	86
Cellular to public WiFi	45	49
Cellular (same carrier)	67	65
Cellular (different carriers)	96	99
WiFi Direct	2441	4155

Table 2: Tap connect times (ms); lower is better

conducting our experiments across a wide range of smartphones and tablets running stock Android and iOS, including Pixel 3a (P3a) smartphones running Android 9.0 Pie, Nexus 9 (N9) tablets running Android Nougat 7.1.1, Nexus 7 Flo (N7) tablets running Android Marshmallow 6.0.1, iPhone 11 Pro (iPh) running iOS 13.3.1, and iPhone 6S (iPh6) running iOS 12.2. The N9, P3a, and iPh6 support IEEE 802.11ac, the iPh supports IEEE 802.11ax, and the N7 uses IEEE 802.11n. Only the iPh and P3a have cellular (LTE) capability.

6.1 Performance Measurements

Connectivity Measurements We first measured how long it takes from a tap to establishing a TCP connection to quantify how quickly users can connect mobile systems together. We performed our experiments across eight network configurations: (1) two systems connected to an ASUS RT-AC66U WiFi router with NAT (NAT WiFi), (2) two systems connected to a university campus public WiFi network (Public WiFi), (3) one system behind the ASUS RT-AC66U WiFi router with NAT connecting to another system on a university campus public WiFi network (NAT WiFi to public WiFi), (4) one system behind the ASUS RT-AC66U WiFi router with NAT connecting to another system on T-Mobile cellular service (NAT WiFi to cellular), (5) one system on T-Mobile cellular service connected to another system on a university campus public WiFi network (Cellular to public WiFi), (6) two systems on T-Mobile cellular service (Cellular same carrier), (7) one system on T-Mobile connected to another system on Verizon cellular service (Cellular different carriers), and (8) two systems connecting via WiFi Direct hosted by one of the systems.

Table 2 shows the connect times for each of the eight network configurations when using two Android systems and an Android and iOS system; there was not much difference among different pairs of Android systems, so we only show P3a results due to space constraints. Each measurement was averaged over a hundred experiments. Infrastructure WiFi generally connects the fastest at less than 20ms, cellular takes longer but less than 100ms, and WiFi Direct takes the longest at roughly 2.5 to 4s. These measurements show the benefits of Tap's multi-network approach to achieve connect latencies for both infrastructure WiFi and cellular of less than 100ms, a commonly used threshold below which users view the response time as negligible [72]. WiFi Direct provides an effective fallback when network infrastructure is unavailable, but takes substantially longer because it requires the guest to also connect to the host's new WiFi Direct network. Due to its proprietary implementation, the iPh cannot host a cross-platform WiFi Direct network for an Android guest, but it can connect to an Android host's WiFi Direct network, which was the measurement performed here. We also performed a similar measurement using Apple AirDrop and Android Beam, which had connect times between 3 to 3.5s, comparable to the connect times for WiFi Direct when using Tap.

Method	Touches	Windows	Time (s)
Tap	3	1	8.7
Android Beam	3	1	25.3
AirDrop	4	3	13.5
Text Msg	8	5	23.7
Email	9	3	27.6

Table 3: User-perceived latency for sharing an image

To provide a more complete measure of the time a user would experience in practice when connecting two systems together, we compared Tap against other approaches via a small IRB-approved user study for performing a very simple task between two systems, sharing an image from one system to another. Since the same task is performed in all cases, the differences in latency for performing the task can be loosely attributed to differences in establishing a connection among different approaches. We compared Tap against AirDrop and Beam again, but also compared against two other common approaches, texting and also emailing the image using Gmail. As a conservative measure, we assumed that the sending user already had the contact info for the recipient. We measured time using a stopwatch and included all user interface actions in the time measured from selecting the image in the photo gallery app for sending to receiving the image in the photo gallery app on another system. We report average measurements for three different users, each of which performed the experiment five times; a 3.4 MB image was used. All systems were connected to a public university campus WiFi network. Because the iOS and Android measurements were similar, we only report Android measurements using P3a systems except for AirDrop, which can only be done using iPh systems. All reported measurements used the standard Android Gallery app, with Tap modifications as discussed in Section 6.2, except AirDrop, which used the iOS Photos app.

Table 3 shows the latency measurements along with a count of the number of screen touches required to perform the task and the number of dialog windows used. Tap was the fastest in terms of time and the number of user interface actions required. Tap was 2 to 3 times faster than all other Android methods and more than 1.5 times faster than AirDrop. Tap and Android Beam performed similar user interface actions, but Beam suffers on performance due to its use of Bluetooth for data communications. AirDrop is slower than Tap both because of its much slower connect time via WiFi Direct and the more limited bandwidth of WiFi Direct compared to taking advantage of available infrastructure WiFi. Text messaging and email are much slower because they require many more user interface actions to set up connectivity between systems. Texting and email require sharing the image with text messaging or email app, choosing a contact and/or a new or existing conversation, then eventually hitting the send button. Even then, end-to-end connectivity is not complete as on the receiving side, the user must still touch on the receiving side notification, touch on the image in the received message or email, then touch the corresponding button to saving it. In contrast, Tap uses a simple physical interaction that for the first time can be programmed and connected to the native functionality

Feature	Data unit	Avg unit size (KB)	Local call (ms)	Remote call (ms)	Bandwidth (kbps)	Max freq. (Hz)	Tested settings
Sensor (callback)	Sensor event	0.10	0.03	1.10	496.94	700	All sensors, "fastest" rate
Input (callback)	Touch event	1.19	0.03	1.10	582.84	200	Rapid multi-touches
Location (callback)	Location event	0.72	0.03	2.80	9.04	2	GPS, GLONASS
Microphone	Data buffer	3.84	38.93	39.87	769.35	26	Mono 16-bit PCM, 48KHz
Camera (preview)	Encoded frame	11.09	0.40	4.27	2666.67	30	720x1280, 4mbps
Camera (picture)	Image	1197.70	594.85	961.00	9570.41	1	2592x1944, JPEG
Audio (raw)	Data buffer	0.97	4.44	4.66	1545.71	200	Stereo 16-bit PCM, 48KHz
Display (preview)	Encoded frame	43.06	3.51	3.97	9967.06	30	720x1280, 10mbps
Clipboard	Clipboard item	0.25	1.61	2.98	2.10	1	Copying text
Notification	Notificat. post	0.23	16.13	19.27	1.82	1	Notification with an icon
Intent	Intent activity	0.21	19.23	20.82	1.61	1	Launching a URL
Calendar	Calendar event	0.17	9.25	12.83	1.56	1	Calendar event
Contacts	Contact item	43.08	27.07	276.28	328.22	1	Contact with a photo
Media	Image	1170.37	385.22	932.59	9508.68	1	Image from gallery

Table 4: Tap Android feature benchmarks

of the app to replace most of that complexity of launching a separate communication app, finding or entering contacts, setting up the actual message for communication, then responding to various notifications on the receiving side to get the data to the right place.

API Measurements We measured the cost of various data source virtualization operations using Tap to quantify its overhead and performance. Because Tap adds some overhead to replicate and intercept APIs via its framework, we first measured the baseline cost of this API redirection by calling an empty method to give a worst case measure of performance. Calling an empty method via Tap versus directly without Tap incurs a 10% overhead on N7, 16% on P3a, and 6% on iPh. The latency is roughly 30μs on N7, so the time overhead from redirection is small and negligible in practice once the cost of sending and receiving data is included.

We then measured the cost of accessing various Android features locally versus remotely using Tap. Table 4 shows these measurements using N7s connected via WiFi Direct. All discrete event measurements were averaged based on multiple runs and streaming event measurements, microphone, camera preview, audio, and display, were averaged based on one minute of streaming. These measurements provide a conservative worst case measure of Tap performance. Tap has the highest overhead for remotely accessing the features with the lowest local latency, such as sensors which take only 30μs to access locally but 1ms to access remotely due to network latency. While the percentage overhead is high, the actual latency remains modest. For features that are more data-intensive such as getting an image from the camera or local storage, the additional overhead for Tap is due to network bandwidth limits. However, other than the features involving sending large images, the absolute latencies for accessing other remote features using Tap are generally small. The latencies are small enough that they are much less than the response time needed to meet the maximum frequency requirements for each data source. For example, video data that must be delivered at 30 fps takes no more than 4.27 ms per frame with Tap, roughly eight times faster than needed. Tap is able to deliver good user-perceived performance for all of the features, including data sources such as audio and video with real-time requirements.

6.2 Example Apps

To demonstrate that Tap makes it easier to build multi-mobile apps, we describe some Tap Android apps. Table 5 shows each app, their total lines of code (LOC), LOC for Tap functionality, and the percentage of total LOC needed for Tap functionality. For Epic Swords 2 and Snapchat, the LOC are calculated from decompiling the APK into Java source files since they are closed source. The LOC show that Tap makes it simple to include multi-mobile functionality in an app with modest implementation effort, in all cases constituting a small percentage of the overall implementation complexity. For existing apps, this is much less than 1% of the app LOC. For our own apps built from scratch, the proportion is higher, since we supported fewer features in the apps, but it is still relatively low compared to the total app LOC. For all apps, Tap LOC were less than 250.

Media Sharing Gallery We modified the stock Android Gallery app by adding an option to share media via Tap and to listen on hosts sharing media. A user can select media from the Gallery app, press "Share via Tap", and tap any number of remote systems. Guest systems running the same gallery app would prompt their users to preview, accept, or reject the media. We also developed an iOS version of the app to do the same, allowing media sharing across iOS and Android using Tap.

Epic Swords 2 is an Android game available through Google Play that allows users to fight opponents with a sword by swiping on the touchscreen. Swiping on a touchscreen to control the sword is not intuitive, especially for actions such as a stab which requires a V swipe. Using Tap, we modified the app to use a smartphone as a Wii-like controller so that the physical movement of the smartphone was not only used to control the sword, but the damage inflicted by the sword was correlated with the speed of the physical swing; a faster swing caused more damage. Since the game was closed source, we modified it by decompiling it to smali [14], adding our Tap and app code, and recompiling and installing it. The new code requests remote sensor data from another system that has been tapped with the one running the app, converts it to input touches, and incorporates the damage inflicted based on swing speed. This app-specific use

App	Total	Tap	Tap/Total
Media Sharing Gallery	72593	68	0.09%
Epic Swords 2	219274	191	0.09%
Surround Sound Music Player	2792	245	8.78%
Multi-channel Audio Recorder	2897	228	7.87%
Snapchat Group Snap	2303299	235	0.01%

Table 5: LOC for apps and Tap app modifications

of feature data to control the damage inflicted by a sword strike inherently requires app modifications to provide the required logic.

Surround Sound Music Player Android does not support anything above stereo audio on mobile systems, limiting audio fidelity. Using Tap, we implemented a music player app that provides a higher fidelity multi-channel surround sound system by sharing audio across multiple systems. A host mobile system can tap with other mobile systems, which in turn share their speakers with the host system. The app examines the number of channels on the music file to be played and the surround sound is then automatically configured based on the positions of the mobile systems acting as speakers using indoor localization. Configuration hints are provided to users on how to place the systems if an optimum surround sound configuration cannot be achieved with the current collection. Systems can be dynamically added or removed, but testing was mostly performed with a 5.0 surround sound configuration.

Multi-channel Audio Recorder Mobile systems are not capable of recording multi-channel audio such as 3D audio. We implemented a Tap audio recorder that utilizes its own and other remote systems' microphones to combine them into a single multi-channel recording. Systems are introduced by tapping. Once the task host app user presses record, An initial beep is emitted from the app to synchronize the recordings.

Snapchat Group Snap Snapchat is an existing app that allows users in the same room to take a photo or video and add filters using a single camera. Using Tap, we added a new Group Snap function to the existing Snapchat app to allow users in different places to take a group photo or video and add filters with all user cameras instead of needing to crowd around a single camera. We modified Snapchat using Tap to share cameras of systems involved in a Group Snap task such that each user has a split-screen view of their own camera and the cameras of other users. Users can apply their own filters to all cameras in their view, and take a picture with filters or record live video with filters to send to their own list of friends. Since we did not have access to Snapchat source code, we modified the app by decompiling it to smali, adding our Tap changes, then recompiling and installing it. To Group Snap, a user double presses the main Snapchat screen and taps systems with other users to receive invites to the Group Snap. Users can double press the screen again to invite more users or to stop the Group Snap, and a user can be invited without needing to manually add a Snapchat contact.

6.3 Usability Studies

We used the various Tap apps described in Section 6.2 to perform a small IRB-approved user study to evaluate the user experience with Tap. There were eight users, half of them technical users, aged 30-39, and half of them are not, aged 22-41. The users volunteered for up to two hours. A demo of each app was given initially to users to show them what the apps do and how they work. After using each app, users were asked questions specific to the usability experience from strongly agree to strongly disagree. The questions and average scores are listed in Table 6, where higher scores are better. All systems were evaluated using infrastructure WiFi unless otherwise indicated.

Questions (1 = Strongly disagree, 5 = Strongly agree)	Avg
I prefer sharing media through the Tap gallery app	4.88
Sword game with controller is a more intuitive, realistic, & fun gaming experience	4.86
Surround sound music player provided better audio	4.38
Quality of recorded audio using Tap recorder was better	4.25
Snapchat group snap feature was easy to use	3.88
Snapchat group snap was smooth & well synchronized	4.13
I would like to use the Snapchat group snap feature	3.38

Table 6: Tap app-specific survey questions and scores

For the gallery app, users were provided with two P3a and an iPh to share images. The study coordinator discussed the existing ways to share media via AirDrop, messaging, or social media. Users were instructed to share media through the Tap-modified gallery app, first by sharing media between the two P3a, then by sharing from a P3a to the iPh. All users agreed that the Tap-modified gallery app was their preferred method for transferring media compared to existing methods.

For the sword game, users were handed a N9 running the game and a P3a to use as a controller. Users were asked to play the game by swiping on the touchscreen and then by introducing the P3a as a controller via Tap, where the swinging speed correlates with the damage inflicted in the game. All users agreed that playing the sword fighting game using a controller was a much more intuitive, realistic, and fun gaming experience compared to using the touchscreen itself.

For the surround sound music player app, users were given a N7 to listen to multi-channel music. Users were then asked to introduce four other N7s using Tap. Users listened to the same multi-channel music but in a surround sound configuration instead. Comparing the audio from a single system versus using the surround sound music player with multiple systems, most users agreed that the surround sound audio quality produced by the Tap app is much better.

For the multi-channel audio recording app, users were given a P3a to record via the local Android app and listen to the recording through headphones. Users were then asked to introduce a P3a and iPh via Tap to run the Tap multi-channel recording app, record audio, and then listen to the recording using headphones. Users agreed the quality of the recorded audio using the Tap multi-channel recorder is better. Users who chose "Neither" for audio, mentioned they rarely use headphones or do not prefer surround systems in general.

For Snapchat, users were given a N9 with the Tap-modified Snapchat app. They were asked to initiate a group snap with the study coordinator holding a N9, and note the process of starting a group snap and the video quality. Users thought the Group Snap video quality was good, but those who rarely use Snapchat were less enthusiastic about the feature.

After users used the apps, they were asked to fill a System Usability Scale (SUS) [76] ten-item questionnaire for Tap. This is an industry-standard questionnaire used in many similar studies [48, 65–68] as a reliable tool to measure usability. The average SUS score, a percentile, for Tap was 82 out of 100. A score above 68 is considered above average and Tap substantially exceeded that score, indicating a high

degree of usability overall. Our usability studies show that users appreciate an easy way to connect systems for a better experience.

7 RELATED WORK

Various approaches have explored ways to compose mobile systems [16]. Early work explored dynamic composition of mobile systems to share discoverable resources for nearby systems to rely on other nearby components, such as using a larger TV display nearby rather than combining displays of multiple systems [80]. System-level sharing approaches such as Rio [2] and Mobile Plus [60] provide one-to-one transparent sharing, while M2 [1] provides optimized performance for transparent sharing of multiple devices. Flux [38] migrates apps across Android systems. FLUID [59] introduces a system to migrate or replicate user interface (UI) elements in apps to nearby systems. DynaMix [20] provides a restrictive framework for efficiently sharing resources such as CPU, memory, and storage over different IoT devices. These platform approaches require system changes that make them difficult to deploy in practice, and cannot effectively support app-specific behaviors. In contrast, Tap takes an app-level sharing approach which requires modifications of apps, but does not require any system changes and supports a broader range of app functionality by allowing apps to express app-specific behavior regarding how remote resources are used. For example, Tap enables the Epic Swords 2 app described in Section 6.2 to base the damage inflicted by a sword on the swing speed of a remote device, an app-specific behavior not possible with the original app as it was limited to local touchscreen input. Various industry solutions [6, 9, 10, 15, 29, 55, 58] also provide one-to-one sharing of display content, messages, and other features. Unlike Tap, none of these approaches provide easy ways of connecting systems together to create the actual dynamic composition of mobile systems, and none of them provide support for developers to create apps that can leverage multiple mobile systems.

Tap introduces data source virtualization to enable composition by virtualizing the data source namespace. Namespace virtualization is the key idea used by containers to isolate processes [4, 11, 18, 24, 39, 49, 62]. Data virtualization [77] has also been used to provide a shared view for databases. While these concepts are somewhat related, Tap uses virtualization in a new way for a new purpose. While M2 relies on existing APIs and data sources to share features, Tap uses a fundamentally different approach. M2 is designed to support unmodified apps, which are not designed to interact with both local and remote data sources, by using a local data source as a proxy for a remote data source so apps only see one data source, not multiple data sources. In contrast, Tap is designed to allow apps to explicitly manage and interact with multiple data sources. Data source virtualization in Tap provides a way for all multiple local and remote data sources to be used with existing APIs.

To make it easier to connect mobile systems, proximity discovery approaches have used Bluetooth signal strength [31, 43, 44, 52], acoustics [31], gesture-based pairing [37, 57], and QR codes [27], but these do not work as well as NFC. AirDrop [8] facilitates connectivity between Apple systems and uses Bluetooth for discovery, but is also not as easy to use as NFC [23, 46, 63, 75]. Android Beam [33] and Android S Beam [69] both use NFC for discovery, but Android Beam uses Bluetooth while Android S Beam [69] uses WiFi Direct for data communication. A precursor to Android Beam, Bump [17] could exchange contacts and files between systems, but lacked a peer-to-peer connectivity mechanism, instead requiring connectivity to cloud infrastructure to allow systems to bump against each other to connect through the cloud. All of these approaches only offer connectivity through a single pre-selected medium and only to share files or contacts. Unlike Tap, they do not provide support for heterogeneous mobile systems and networks, do not provide a way to adaptively leverage infrastructure networking for data communication, and do not provide a programmable interface to allow apps to leverage these mechanisms for general feature sharing. Other work for convenient system pairing relied heavily on the use of additional hardware as covered in spontaneous device association [22] and electromagnetic sensing based research [21, 35, 50, 53, 79, 82, 83]. Tap provides a cross-platform solution to easily identify and authenticate nearby systems without requiring additional hardware or network infrastructure.

Various frameworks have been developed to use nearby systems. Beam [71], different from Android Beam, provides a developer friendly framework in an Internet-of-Things (IoT) environment to discover and use sensing devices, mostly for data gathering. Proxemic systems define the interactions and types of interactions between nearby systems, without considering cross-platform support or proximity detection without additional hardware [34, 54]. Conductor [36] shares tasks and data across systems. SPF [12] utilizes AllJoyn [61] to support proximity connectivity. Junction [25] provides a limited connectivity API via NFC. These frameworks primarily focus on connectivity, not feature sharing. Liquid software [13, 28] refers to software that can operate seamlessly across multiple systems, but only offers basic app collaboration. Unlike Tap, none of these approaches provide a multi-network connectivity mechanism or make it possible to reuse existing APIs to build apps across dynamic compositions of mobile systems.

8 CONCLUSIONS

Tap is an app framework to make it easy for users to dynamically compose collections of heterogeneous mobile systems and developers to write apps that make use of those impromptu collections. Tap enables users to tap systems together for ephemeral interactions. Tapping triggers a NFC-based mechanism that exchanges connectivity information and security credentials, allowing systems to securely connect through heterogeneous networks without user accounts, email addresses, phone numbers, or cloud infrastructure. Tap makes it possible and easy for apps to use existing mobile platform APIs across mobile systems by virtualizing data sources, such as media, cameras, and speakers, so that local and remote hardware and software features can be combined together upon tapping. We have built a Tap prototype and modified and implemented various apps using Tap, demonstrating that it is easy to use and enables apps to provide powerful new functionality across multiple mobile systems. Our results show that Tap works across unmodified Android and iOS systems, has good performance, and is user and developer friendly.

9 ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-1717801, CNS-1563555, CCF-1918400, and CNS-2052947.

REFERENCES

[1] Naser AlDuaij, Alexander Van't Hof, and Jason Nieh. 2019. Heterogeneous Multi-Mobile Computing. In *Proceedings of the 17th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2019)*. Seoul, South Korea, 494–507.

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

[3] 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.

[4] Jeremy Andrus, Christoffer Dall, Alex Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*. Cascais, Portugal, 173–187.

[5] 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.

[6] Apple Inc. iCloud. https://www.icloud.com.

[7] Apple Inc. NFCISO7816Tag. https://developer.apple.com/documentation/corenfc/nfciso7816tag.

[8] Apple Inc. 2018. Share Content with AirDrop on Your iPhone, iPad, or iPod touch. https://support.apple.com/en-us/HT204144.

[9] Apple Inc. 2019. How to AirPlay Content from Your iPhone, iPad, or iPod touch. https://support.apple.com/en-gb/HT204289.

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

[11] Ricardo Baratto, Shaya Potter, Gong Su, and Jason Nieh. 2004. MobiDesk: Mobile Virtual Desktop Computing. In *Proceedings of the 10th Annual ACM International Conference on Mobile Computing and Networking (MobiCom 2004)*. Philadelphia, PA, 1–15.

[12] Luciano Baresi, Laurent-Walter Goix, Sam Guinea, Valerio Panzica La Manna, Jacopo Aliprandi, and Dario Archetti. 2015. SPF: A Middleware for Social Interaction in Mobile Proximity Environments. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. Florence, Italy, 79–88.

[13] Luciano Baresi, Anita Imani, Cristina Fra, and Massimo Valla. 2018. LIQDROID: Towards Seamlessly Distributed Android Applications. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft 2018)*. Gothenburg, Sweden, 597–601.

[14] Ben Gruver. 2017. smali/baksmali. http://www.baksmali.com.

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

[16] Frederik Brudy, Christian Holz, Roman Radle, Chi-Jui Wu, Steven Houben, Clemens Nylandsted Klokmose, and Nicolai Marquardt. 2019. Cross-Device Taxonomy: Survey, Opportunities and Challenges of Interactions Spanning Across Multiple Devices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI 2019)*. Glasgow, United Kingdom, 1–28.

[17] Bump Technologies. Bump - Easily Transfer Photos, Files and Contacts between Your Phone and Computer. http://www.bu.mp.

[18] Canonical Ltd. Linux Containers. https://linuxcontainers.org/.

[19] Centers for Disease Control and Prevention. Coronavirus (COVID-19). https://www.cdc.gov/coronavirus/2019-ncov/index.html.

[20] Dongju Chae, Joonsung Kim, Gwangmu Lee, Hanjun Kim, Kyung-Ah Chang, Hyogun Lee, and Jangwoo Kim. 2018. DynaMix: Dynamic Mobile Device Integration for Efficient Cross-device Resource Sharing. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 2018)*. Boston, MA, 71–83.

[21] Ke-Yu Chen, Gabe A. Cohn, Sidhant Gupta, and Shwetak N. Patel. 2013. uTouch: Sensing Touch Gestures on Unmodified LCDs. In *Proceedings of the 2013 CHI Conference on Human Factors in Computing Systems (CHI 2013)*. Paris, France, 2581–2584.

[22] Ming Ki Chong, Rene Mayrhofer, and Hans Gellersen. 2014. A Survey of User Interaction for Spontaneous Device Association. *ACM Computing Surveys* 47, 1 (May 2014), 8:1–8:40.

[23] Christian Zibreg. 2016. AirDrop not working? Try these troubleshooting tips. https://www.idownloadblog.com/2016/02/20/airdrop-troubleshooting-tips-2/.

[24] Docker Inc. Get Started with Docker. https://www.docker.com/.

[25] Ben Dodson, Aemon Cannon, Te-Yuan Huang, and Monica S. Lam. 2011. *The Junction Protocol for Ad Hoc Peer-to-Peer Mobile Applications.* Technical Report. Computer Science Department, Stanford University.

[26] Elizabeth Schulze. 2019. Contactless cards are just catching on in the US - years after the rest of the world. https://www.cnbc.com/2019/04/12/contactless-cards-and-apple-pay-are-just-catching-on-in-the-us.html.

[27] Florian Draschbacher. Fast File Transfer - Apps on Google Play. https://play.google.com/store/apps/details?id=com.floriandraschbacher.fastfiletransfer.

[28] Andrea Gallidabino, Cesare Pautasso, Tommi Mikkonen, Kari Systa, Jari-Pekka Voutilainen, and Antero Taivalsaari. 2017. Architecting Liquid Software. *Journal of Web Engineering* 16, 5-6 (Sept. 2017), 433–470.

[29] Google Inc. Chromecast. https://www.google.com/chromecast/.

[30] Google Inc. Host-Based Card Emulation Overview | Android Developers. https://developer.android.com/guide/topics/connectivity/nfc/hce.

[31] Google Inc. Nearby | Google Developers. https://developers.google.com/nearby.

[32] Google Inc. NfcAdapter | Android Developers. https://developer.android.com/reference/android/nfc/NfcAdapter.

[33] Google Inc. Sharing files with NFC | Android Developers. https://developer.android.com/training/beam-files.

[34] Saul Greenberg, Nicolai Marquardt, Till Ballendat, Rob Diaz-Marino, and Miaosen Wang. 2011. Proxemic Interactions: The New Ubicomp? *Interactions* 18, 1 (Jan. 2011), 42–50.

[35] Sidhant Gupta, Matthew S. Reynolds, and Shwetak N. Patel. 2010. ElectriSense: Single-point Sensing Using EMI for Electrical Event Detection and Classification in the Home. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing (UbiComp 2010)*. Copenhagen, Denmark, 139–148.

[36] Peter Hamilton and Daniel J. Wigdor. 2014. Conductor: Enabling and Understanding Cross-device Interaction. In *Proceedings of the 2014 CHI Conference on Human Factors in Computing Systems (CHI 2014)*. Toronto, Canada, 2773–2782.

[37] Ken Hinckley, Gonzalo Ramos, Francois Guimbretiere, Patrick Baudisch, and Marc Smith. 2004. Stitching: Pen Gestures That Span Multiple Displays. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2004)*. Gallipoli, Italy, 23–31.

[38] 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.

[39] 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.

[40] Internet Society. 2018. State of IPv6 Deployment 2018. https://www.internetsociety.org/resources/2018/state-of-ipv6-deployment-2018/.

[41] Ivan Jenic. 2019. Fix: Dropbox for Android not showing files. https://mobileinternist.com/dropbox-android-not-showing-files.

[42] Jessica Dickler. 2019. As of today: an NYC commute without cash. https://www.cnbc.com/2019/05/31/the-new-york-city-subway-systems-cashless-payments-start-friday.html.

[43] Tero Jokela, Ming Ki Chong, Andres Lucero, and Hans Gellersen. 2015. Connecting Devices for Collaborative Interactions. *ACM Interactions* 22, 4 (June 2015), 39–43.

[44] Tero Jokela and Andres Lucero. 2014. FlexiGroups: Binding Mobile Devices for Collaborative Interactions in Medium-sized Groups with Device Touch. In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices and Services (MobileHCI 2014)*. Toronto, Canada, 369–378.

[45] Jonny Evans. 2018. How to fix iCloud when it stops working. https://www.computerworld.com/article/3322896/how-to-fix-icloud-when-it-stops-working.html.

[46] Karen Haslam. 2019. How to fix AirDrop problems. https://www.macworld.co.uk/how-to/mac/fix-airdrop-problems-3693158/.

[47] Kevin Peachey. 2019. Half of all debit card payments now contactless. https://www.bbc.com/news/business-50015312.

[48] John S. Koh, Steven M. Bellovin, and Jason Nieh. 2019. Why Joanie Can Encrypt: Easy Email Encryption with Easy Key Management. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 2019)*. Dresden, Germany, 2:1–16.

[49] Oren Laadan and Jason Nieh. 2010. Operating System Virtualization: Practice and Experience. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR 2010)*. Haifa, Israel.

[50] Gierad Laput, Chouchang Yang, Robert Xiao, Alanson Sample, and Chris Harrison. 2015. EM-Sense: Touch Recognition of Uninstrumented, Electrical and Electromechanical Objects. In *Proceedings of the 28th Annual Symposium on User Interface Software and Technology (UIST 2015)*. Charlotte, NC, 157–166.

[51] Alexander Loffler, 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.

[52] Andres Lucero, Tero Jokela, Arto Palin, Viljakaisa Aaltonen, and Jari Nikara. 2012. EasyGroups: Binding Mobile Devices for Collaborative Interactions. In *CHI 2012 Extended Abstracts on Human Factors in Computing Systems (CHI EA 2012)*. Austin, TX, 2189–2194.

[53] Takuya Maekawa, Yasue Kishino, Yasushi Sakurai, and Takayuki Suyama. 2011. Recognizing the Use of Portable Electrical Devices with Hand-worn Magnetic Sensors. In *Proceedings of the 9th International Conference on Pervasive Computing (Pervasive 2011)*. San Francisco, CA, 276–293.

[54] Nicolai Marquardt, Robert Diaz-Marino, Sebastian Boring, and Saul Greenberg. 2011. The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. In *Proceedings of the 24th Annual Symposium on User Interface Software and Technology (UIST 2011)*. Santa Barbara, CA, 315–326.

[55] Microsoft Corporation. Windows Continuum for Windows 10 Phones and Mobile. https://www.microsoft.com/en-us/windows/continuum.

[56] 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.

[57] Heidi Selmer Nielsen, Marius Pallisgaard Olsen, Mikael B. Skov, and Jesper Kjeldskov. 2014. JuxtaPinch: An Application for Collocated Multi-device Photo Sharing. In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices and Services (MobileHCI 2014)*. Toronto, Canada, 417–420.

[58] Nintendo Co., Ltd. Nintendo Switch. http://www.nintendo.com/switch.

[59] Sangeun Oh, Ahyeon Kim, Sunjae Lee, Kilho Lee, Dae R. Jeong, Steven Y. Ko, and Insik Shin. 2019. FLUID: Flexible User Interface Distribution for Ubiquitous Multi-Device Interaction. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking (MobiCom 2019)*. Los Cabo, Mexico, 1–16.

[60] 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.

[61] Open Connectivity Foundation. AllJoyn Open Source Project. https://openconnectivity.org/developer/reference-implementation/alljoyn.

[62] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. 2002. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*. Boston, MA, 361–376.

[63] Rene Ritchie. 2020. AirDrop not working? Here's the fix! https://www.imore.com/how-to-fix-airdrop-iphone-ipad.

[64] Kay Romer. 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.

[65] Scott Ruoti, Jeff Andersen, Scott Heidbrink, Mark O'Neill, Elham Vaziripour, Justin Wu, Daniel Zappala, and Kent Seamons. 2016. "We're on the Same Page": A Usability Study of Secure Email Using Pairs of Novice Users. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI 2016)*. San Jose, CA, 4298–4308.

[66] Scott Ruoti, Jeff Andersen, Travis Hendershot, Daniel Zappala, and Kent Seamons. 2016. Private Webmail 2.0: Simple and Easy-to-Use Secure Email. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST 2016)*. Tokyo, Japan, 461–472.

[67] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent Seamons. 2015. Why Johnny Still, Still Can't Encrypt: Evaluating the Usability of a Modern PGP Client. *arXiv e-prints* (Oct. 2015), 5 pages. arXiv:1510.08555 [cs.CR]

[68] Scott Ruoti, Nathan Kim, Ben Burgon, Timothy van der Horst, and Kent Seamons. 2013. Confused Johnny: When Automatic Encryption Leads to Confusion and Mistakes. In *Proceedings of the 9th Symposium on Usable Privacy and Security (SOUPS 2013)*. Newcastle, United Kingdom, 5:1–5:12.

[69] Samsung. What is S Beam in Samsung Smartphones? https://www.samsung.com/in/support/mobile-devices/what-is-s-beam-in-samsung-smartphones/.

[70] Arne Schmitz, Ming Li, Volker Schonefeld, 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)*. Norrkoping, Sweden, 45–52.

[71] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. 2016. Beam: Ending Monolithic Applications for Connected Devices. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 2016)*. Denver, CO, 143–157.

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

[73] Stephanie Walden. Banking After COVID-19: The Rise of Contactless Payments in the U.S. https://www.forbes.com/advisor/banking/banking-after-covid-19-the-rise-of-contactless-payments-in-the-u-s/.

[74] The Khronos Group, Inc. OpenGL - The Industry Standard for High Performance Graphics. https://www.opengl.org.

[75] Tim Brookes. 2019. AirDrop Not Working? Here's How to Fix It. https://www.howtogeek.com/442534/airdrop-not-working-heres-how-to-fix-it/.

[76] U.S. Department of Health & Human Services. System Usability Scale (SUS). https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html.

[77] Rick Van Der Lans. 2012. *Data Virtualization for Business Intelligence Systems: Revolutionizing Data Integration for Data Warehouses* (first ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA.

[78] Vikas Saraogi. 2020. Contactless payments will be the new normal for shoppers in the post Covid-19 world. https://newsroom.mastercard.com/asia-pacific/2020/05/20/contactless-payments-will-be-the-new-normal-for-shoppers-in-the-post-covid-19-world/.

[79] Edward J. Wang, Tien-Jui Lee, Alex Mariakakis, Mayank Goel, Sidhant Gupta, and Shwetak N. Patel. 2015. MagnifiSense: Inferring Device Interaction Using Wrist-worn Passive Magneto-inductive Sensors. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2015)*. Osaka, Japan, 15–26.

[80] 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.

[81] Wi-Fi Alliance. 2018. Wi-Fi Peer-to-Peer (P2P) Technical Specification v1.7. https://www.wi-fi.org/file/wi-fi-peer-to-peer-p2p-technical-specification-v17.

[82] Robert Xiao, Gierad Laput, Yang Zhang, and Chris Harrison. 2017. Deus EM Machina: On-Touch Contextual Functionality for Smart IoT Appliances. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI 2017)*. Denver, CO, 4000–4008.

[83] Chouchang Yang and Alanson P. Sample. 2016. EM-ID: Tag-less Identification of Electrical Devices via Electromagnetic Emissions. In *Proceedings of the 10th Annual IEEE International Conference on RFID (RFID 2016)*. Orlando, FL, 1–8.

[84] 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/.

[85] Zoom. In this together. Keeping you securely connected wherever you are. https://www.zoom.us/.