Mobile Plus: Multi-device Mobile Platform for Cross-device Functionality Sharing

Sangeun Oh, Hyuck Yoo, Dae R. Jeong, Duc Hoang Bui, and Insik Shin
School of Computing, KAIST
Daejeon, Republic of Korea
{ohsang1213, yoohuck12, dragon812, ducbuihoang, insik.shin}@kaist.ac.kr

ABSTRACT

In recent years, the explosion of diverse smart devices such as mobile phones, TVs, watches, and even cars, has completely changed our lives. We communicate with friends through social network services (SNSs) whenever we want, buy stuff without visiting shops, and enjoy multimedia wherever we are, thanks to these devices. However, these smart devices cannot simply interact with each other even though they are right next to each other. For example, when you want to read a PDF stored on a smartphone on a larger TV screen, you need to do complicated work or plug in a bunch of cables. In this paper, we introduce M+, an extension of Android that supports cross-device functionality sharing in a transparent manner. As a platform-level solution, M+ enables unmodified Android applications to utilize not only application functionalities but also system functionalities across devices, as if they were to utilize them inside the same device. In addition to secure connection setup, M+ also allows performing of permission checks for remote applications in the same way as for local. Our experimental results show that M+ enables transparent cross-device sharing for various functionalities and achieves performance close to that of within-device sharing unless a large amount of data is transferred.

CCS Concepts

•**General and reference** → **Design;** •**Human-centered computing** → **Mobile computing; Mobile devices;** *Collaborative interaction;* •**Software and its engineering** → *Operating systems; Communications management;*

Keywords

Multi-device Mobile Platform; Functionality Sharing; Remote Procedure Call; Inter-Process Communication;

MobiSys'17, June 19-23, 2017, Niagara Falls, NY, USA
© 2017 ACM. ISBN 978-1-4503-4928-4/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3081333.3081348

1. INTRODUCTION

The mobile app ecosystem continues to grow and mature rapidly, and offers a wide variety of services, such as SNS, shopping, entertainment, and healthcare. Mobile applications have become complex and diverse, and they have come to use the *functionalities*[1] of other applications or system services. At the same time, users own multiple mobile devices. A recent survey reported an average of more than three devices per person [1], and users are easily tempted to use on each device the functionalities available on other devices.

Such trends present exciting opportunities for multiple smart devices to be used together, including (1) video conferencing on a camera-less smart TV using a smartphone's camera, (2) secure internet shopping on an unsecured public device using payment service from one's own private smartphone, (3) replying to an email on a smartphone while scrolling through its attached documents on a tablet, and doing copy and paste between the two devices, and (4) playing sensor-based games on a tablet while using sensors from a smartphone, which can provide a more convenient method to control them.

Many solutions have been proposed to enable this exciting opportunity. First, many interesting studies have been done to use the resources of other devices through the cooperation of applications. Many apps [26, 32, 6, 7, 15, 31] are available for sharing specific resources, including screen casting, cameras, and sensors. However, their applicability is quite limited, because they work with their own custom applications but do not support unmodified applications. This imposes a great burden on those wanting to develop and deploy such applications for each individual resource or functionality. Second, a great deal of work [39, 14, 27, 37, 22, 23, 13, 21, 4, 42] has been done to develop a cross-device platform that enables unmodified applications to use the resources of other devices. However, these mainly focus on utilizing system resources such as sensors and cameras but do not support application functionalities such as in-app payment and SNS login.

Here, we present a novel platform M+ (Mobile Plus). The main goal of M+ is to allow unmodified applications to share a wide range of functionalities across devices. That is, our system is intended to be insensitive to the type of func-

[1]In this paper, we define functionality as a collection of services, features, content, and resources. We classify functionalities into two categories according to their providers: *application* functionalities offered by applications, and *system* functionalities by Android system services.

tionality, and with no extra burden of application development. To achieve the design goal, M+ extends the existing remote procedure call (RPC) scheme and its underlying binder inter-process communication (IPC) mechanism to multi-device environments. In other words, our key idea is to intercept RPC messages (binder parcels) from a client device and forward them to a server device to execute RPC function logic. This approach suits our design goal for two reasons. First, to the best of our knowledge, all existing applications are using RPCs to utilize the functionalities of other applications or system services. Therefore, extending the RPC scheme will naturally enable support of both application and system functionalities in the multi-device environment. Second, RPC is transparent to the application layer because it is supported at the platform layer. Therefore, the RPC extension does not require modification of applications.

M+ should address several fundamental challenges in extending within-device RPC to cross devices because Android is designed under the assumption that client and server processes run on the same device. (i) Android allows passage of various type of references or handles as RPC parameters, including shared memories, sockets, binders, and files, for better performance. However, passing RPC arguments in a call-by-reference manner does not work properly across different devices. M+ detects reference-type arguments in a functionality-agnostic way and makes their values available transparently between devices. (ii) Android app management logic will not work properly in the multi-device environment: it cannot manage execution context and semantics (e.g., app lifecycles, caller-callee information) of app beyond the device boundary. Hence, M+ addresses this unprecedented issue based on the concept of *virtual activity* to enable unmodified applications to execute on different devices without having their interaction semantics violated, while incurring little change to Android system services. (iii) The Android assumption that there is only one instance of an application will not hold. Because each device can have its own resident app, there could exist multiple instances of the same app when multiple devices are joined for functionality sharing. For this reason, M+ leverages the notion of *remote app registration* to manage multiple instances properly for correct security checks and communication support.

We proved the M+ concept with a prototype using Nexus 6 (smartphone) and Nexus 10 (tablet)[2]. It was demonstrated that M+ allows existing applications on different devices to successfully share functionalities such as Facebook Login, Google Market Payment, Android Contacts, and PDF Viewer; as well as system features (camera, sensor, notification and clipboard). Interestingly, all these sharing processes do not incur significant performance overhead and behave almost like sharing within the same device in most cases, unless they require transfer of a large amount of data. The details about the experiment will be discussed in later sections. In addition, it is worth noting that even if heterogeneous types of devices interact with each other, M+ can easily support the interaction if the devices have just the same RPC interfaces. This is possible because our system is designed to provide functionality sharing via cross-device RPC regardless the type of device.

[2]See http://cps.kaist.ac.kr/mobileplus for our demo video illustrating the interaction between Nexus 6 and Nexus 10.

2. USE CASES

Service sharing: login, payment, and notification. M+ can change the way people utilize services across devices in many places. With M+, one could dispatch certain services to designated devices or users for various reasons. For example, a user might want to log into an SNS app (e.g., Instagram) on a public device such as a free rental tablet in a public library or a smart TV in her hotel room. This comes with a risk to privacy (e.g., social media account hijacking) because public devices could have malware or keystroke-logging software secretly installed. With M+, she can log into the SNS app securely using her Facebook login on her own smartphone. Moreover, a kid might feel an impulse to purchase in-game items after failing to move on to the next round in a mobile game. With M+, her mother can be prompted to deal with the purchase order via PayPal or Google Play on her smartphone. With M+, a user could receive notifications or alarms (e.g., incoming calls, SMS messages, and low battery alarms) forwarded from a smartphone and displayed on a TV.

Contents sharing: file, contact, and calendar. M+ can facilitate the way users access and interact with information between devices. When checking emails on a smartphone, a user might want to view an attached PDF document on a larger-screen device, such as a tablet. With M+, this could be done right away with a single tap on the PDF icon on a smart watch, rather than go through a cloud storage service (e.g., Dropbox) or use an email client app on the tablet. Interestingly, she could work on both devices at the same time: write an email reply on a smartphone and scroll through the attached documents on a tablet simultaneously. In this case, the user could also copy any part of the attached document on the tablet and paste it on the smartphone. With M+, a user could run video conferencing or daily briefing apps on a TV, while accessing contact and calendar data on her smartphone.

I/O sharing: camera. The digital world is now shifting from 2D to 3D, and one great example is the success of 3D selfie camera applications. However, in order to render 3D images from a single camera, the camera must take multiple images from different angles. Apparently, this condition puts critical restrictions on development of many creative uses. Instead, applications seek to construct 3D images using multiple cameras. In this context, M+ provides a platform-level I/O sharing so that those applications can use remote cameras easily.

3. ANDROID BACKGROUND

Application components. Each functionality in Android is implemented as one of four application components: activity, service, content provider, or broadcast receiver. an activity represents the user interface of an application, allowing interaction between the application and the user. Everything a user sees in an application is provided by activities. For example, Facebook login activity provides a UI to allow users to enter ID and password. a service is designed to perform long-running operations in the background. For example, I/O functionalities, such as camera and sensor, are provided with system services. a content provider gives an interface for storing and sharing data using a relational database (e.g. SQLite database). For instance, contact lists and calendar information are managed by content providers.

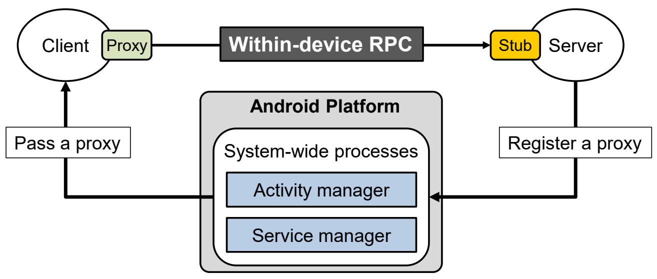


Figure 1: Within-device RPC in Android

a broadcast receiver is for receiving system-wide messages. For example, low battery level warning is broadcast and applications save their states to prevent any data loss.

Functionality sharing in Android. Android is designed to allow applications (processes) to share their functionalities through remote procedure calls (RPCs). Suppose that a client application tries to use a functionality of a server application. Here, function sharing is achieved between them if the client invokes RPC functions for the functionality to the server. The invocation procedure typically consists of two steps. The client first forwards a message (called *parcel*) for calling a RPC function to the server. Upon receiving the parcel, the server executes the RPC function and passes the parcel containing its result to the client.

The Android RPC employs the inter-process communication (IPC) mechanism managed by the binder driver, because it is achieved across the process boundary. To deliver parcels from the client to the server, the binder driver creates a binder connection: a route between the two processes for the parcels. This procedure is called *binding*. Here, the source and target endpoints of a binder connection are a proxy and a stub. RPC invocations are always made in one direction: from proxy to stub.

Figure 1 shows how binding works between client and server processes. A client first requests a proxy to Android system services; then it asks the service manager to share a system functionality or asks the activity manager to share an application functionality. We note that each individual server has registered its proxy to the service/activity manager when it is initiated. When the client obtains the proxy from the server/activity manager, binding is completed in one of two ways depending on the component type of server functionality. The client binds to the server's component directly if the component is a service or content provider, or indirectly via the activity manager if the component is an activity. The latter allows the activity manager to mediate interaction between the client and server activities according to their own interaction semantics. We will describe this in more detail in Section 6.

4. SYSTEM DESIGN OVERVIEW

The main goal of M+ is to allow transparent sharing of both application and system functionalities across multiple mobile devices. To this end, M+ should be oblivious to the type of functionality without incurring any extra burden of application development. M+ extends the within-device RPC scheme across different devices to achieve this goal. Figure 2 shows an overview of M+ consisting of these main

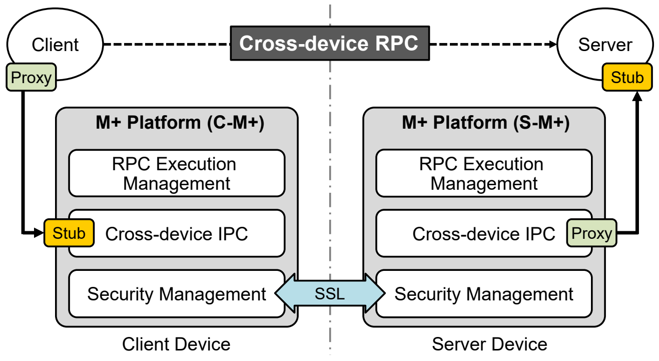


Figure 2: M+ design overview

components: cross-device IPC connection, cross-device RPC execution, and security management.

Cross-device IPC connection. Towards transparent cross-device RPC execution, M+ first seeks to deliver RPC function calls to server processes on different devices without requiring modifications to applications. To this end, M+ extends the existing binder IPC mechanism to create a cross-device binder connection, yet offers the illusion of a within-device IPC. We describe this in detail in Section 5.

Cross-device RPC execution. M+ aims to create an environment in which RPC functions can be executed correctly on remote devices. Many RPC functions may not execute properly across different devices, because Android is designed to support RPC execution within a single device. This gap engenders a different set of challenges which has not previously been addressed to support cross-device RPC. For example, M+ should manage execution context and semantics properly across multiple devices when supporting transparent cross-device RPC execution, because Android does it only within the device boundary. As another example, servers may request information about clients while executing RPC functions. M+ should make such information available on server devices in a transparent manner. In Section 6, we explain which challenges M+ faces and how it addresses them.

Security management. As M+ allows multiple devices to cooperate over the network, various security issues arise. For example, M+ needs to distinguish multiple application instances in multi-device environments in order to perform permission checks accurately. In addition, the existence of various kinds of attacks through the network threaten the security of M+. Thus, M+ establishes a secure connection between the two devices via the SSL protocol. We explain how M+ addresses security concerns in Section 7.

5. CROSS-DEVICE IPC CONNECTION

We first need to extend the binder IPC mechanism to support cross-device RPC, addressing several issues. First, we determine the right place to intercept binder parcels without modifying applications. Second, we need to intercept the smallest set of parcels to minimize an interception overhead. Third, unmodified applications should be able to reach their destinations across devices. This section explains how to address such issues to establish an efficient cross-device binder connection for unmodified applications.

5.1 Binder Parcel Interception

First, we need to intercept parcels from applications to create a cross-device IPC channel transparently. One important decision to make is how to determine the right place to intercept parcels, to support a broad range of functionalities without any modification of existing applications. As shown in Figure 3, a parcel goes through a number of binder layers. The first layer, which is the interface layer, is designed to provide the same function interfaces to both client and server applications. In most cases, the function interfaces are generated in application code through an interface language, called Android interface description language (AIDL). If the parcels were to be intercepted in the interface layer (as done in [25]), function interfaces in existing applications must be modified through recompilation; this is not an adequate solution for our goal.

All the layers below the interface layer belong to the Android platform. These layers are not included in application code, but are dynamically linked during application's runtime. Therefore, in order to support existing applications without modifications, parcels have to be intercepted at one of these layers. M+ determines the parcel intercept point as *BpBinder*, a binder object at the native IPC layer. Because a BpBinder object is placed at the native IPC layer, all Android applications (with either Java or native source codes) should pass through this object for binder IPC. This means that M+ can intercept any parcels at this layer to create cross-device IPC channels for various types of functionalities, no matter whether functionalities are based on either Java or native source codes.

Another important issue is which parcels to intercept. It would impose a considerable overhead if all the parcels should be intercepted. Instead, M+ intercepts the smallest set of parcels, which we call *seed* parcels, that initiate interactions with other applications. Specifically, it intercepts parcels to invoke some specific methods of `ActivityManager` and `ServiceManager`, which fall into two categories: (i) ones to establish a new binder IPC channel such as `getService()` of `ServiceManager`, `bindService()` and `getContentProvider()` of `ActivityManager`, and (ii) others to launch new application components such as `startActivity()` of `ActivityManager`. In this way, M+ narrows down a set of parcels to intercept to the minimum, without having to identify which parcels belong to which functionalities.

5.2 Cross-device Binder IPC

A binder IPC channel is established when a client obtains a (binder) proxy object for a server's (binder) stub, as described in Section 3. One design principle that is important for cross-device binder channels is to provide transparency to both client and server processes. In other words, it should provide the illusion to them that they are interacting within the same device. To achieve this, client and server processes are logically connected via three intermediate connections with two M+ components (see Figure 2). For ease of presentation, we often distinguish the roles of client-side M+ (C-M+) and server-side M+ (S-M+). That is, client and server processes perform binding locally with C-M+ and S-M+, respectively; while C-M+ and S-M+ communicate through a network connection.

Specifically, the binding procedure in the cross-device environment takes place as follows. First, a seed parcel is intercepted and passed to C-M+. It then prompts a window,

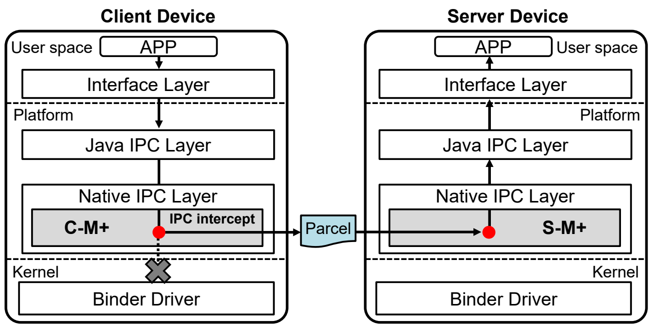


Figure 3: Binder IPC interception point

allowing the client user to decide which server applications to use for the functionality of interest, if there are multiple candidates, including remote ones. When the user decides to use a remote device's functionality, then C-M+ forwards the parcel to S-M+, which passes it to the activity/service manager on the server device. S-M+ obtains a proxy object for the server's stub when the binder driver creates a local binder channel between S-M+ and the server process. The proxy object is then passed back to C-M+. However, this proxy is not valid on the client device, because it is device-specific. So, C-M+ creates a new stub and provides its proxy to the client process, establishing a local binder channel between them. After that, C-M+ and S-M+ maintain the mapping for the two local binder channels by bridging them with a TCP network connection. With the procedure complete, the client recognizes C-M+ as the server and sends all the parcels related to the functionality of interest directly to C-M+. These are then delivered to the server through the cross-device binder connection.

6. CROSS-DEVICE RPC SUPPORT

As explained in Section 5, M+ supports transparent cross-device binder connections, allowing a client to invoke a RPC call on a remote server. Yet, there are many issues to address in supporting cross-device RPC successfully. First, Android applications often pass reference arguments for RPC calls (call-by-reference) to improve efficiency within a single device. Second, client and server applications generally access information about their counterpart in order to perform interaction properly, assuming the information is available on the same device. Third, Android is designed to manage the execution of a client and a server while preserving their interaction semantics mainly for a single device. Each of the above issues makes it impossible for unmodified applications to perform RPC interaction correctly across different devices, if proper support is not provided. In this section, we explain how M+ addresses each individual issue.

6.1 RPC Argument Handling

Android, designed for a single device, allows client and server to pass RPC arguments under call-by-reference semantics to achieve efficiency. For instance, when client or server wants to make an extra binder connection, then they send a proxy as an argument. In addition, when a client wants to share some files (such as photos or PDF documents) with other applications, the client passes either uniform resource identifiers (URIs) or file descriptors. File descriptors for shared memory and Unix domain socket (UDS) also can be used as RPC arguments to share a large amount of data (e.g., camera frames) and to transfer data streams (e.g., sensor readings), respectively.

To support cross-device RPC transparently, M+ needs to identify all reference arguments used in each RPC call and make the value of each reference available on the server device, without incurring any modifications to both client and server applications. In addition, M+ should address such issues in a functionality-oblivious manner, because a great deal of effort is required to examine each individual functionality to determine which reference arguments are involved.

We categorize the reference arguments into four types depending on which IPC channels they are associated with: binder proxy, URI, UDS, and shared memory. For each type, we will explain how M+ detects reference arguments within parcels and deals with them for cross-device RPC. In addition, we will also explain how to resolve synchronization issues for shared memory.

Binder proxy: M+ is able to detect binder proxy objects easily, if included in parcels, since the parcels include metadata in reference to the proxy objects. Upon detecting a proxy object, M+ creates a new binder IPC connection as described in Section 5.

URI: Unlike the proxy object, parcels do not have any information about URI. So, we re-design the parcel structure so that it contains metadata in reference to the URI. We rewrote the `writeToParcel()` function of Android `Uri` class, which is one of the most commonly used URI transmission methods, so that the URI can be easily found in the parcel. If a URI is found, S-M+ stores a copy of the corresponding file on the internal storage of the server device, preventing other processes from accessing it without proper permissions. S-M+ then creates a new URI for the location and replaces the original URI with the new URI before passing the parcel to the server.

UDS and shared memory: UDS and shared memory are stored in parcels in the form of file descriptor, and the parcels contain metadata about file descriptors. Thus, M+ can detect file descriptors easily within parcels. Then, C-M+ refers to the `proc` file system to identify the type of file descriptor exactly. If the file descriptor is found to be UDS, S-M+ creates a new UDS; then passes it to the server. When this comes to shared memory in Android, it is further divided into anonymous shared memory (ASHMEM) and ION memory. S-M+ allocates corresponding shared memory and sends the file descriptor to the server. When allocating memory, it should use the same on both devices. In the case of ASHMEM, C-M+ finds it easily with `ashmem_get_size_region()`, which is provided by the ASHMEM device driver. However, in the case of ION, its device driver does not expose an API function that returns the size of an allocated memory. So, instead of using such a function, M+ obtains it by parsing parameters of the RPC function that is used to allocate an ION memory (i.e., `dequequeBuffer()` of `IGraphicBufferProducer`). It will be easier to solve the issue if the ION device driver can provide proper API functions like ASHMEM.

Unlike other IPC connections, shared memory requires synchronization, because either client or server can make modifications to its own copy. To be synchronized correctly, M+ must know when the memory begins to be modified and when it ends. Android uses some RPC functions related to lock protocols for ION memory (e.g., `requestBuffer()`, `queueBuffer()` of `IGraphicBufferProducer`), and we resolve synchronization issues by using them. Typically, when `requestBuffer()` is called to acquire a lock before modifying the memory, M+ prepares the ION memory where data will be written. Similarly, when `queueBuffer()` is invoked to release the lock after finishing the modification, M+ copies the modified memory back to the remote device. We note that synchronization issues are not considered for ASHMEM, because it tends to be utilized as a one-shot memory to transfer a large amount of data to another process unlike ION. For example, a content provider uses ASHMEM to deliver results for database queries sent by other process.[3] After that, the allocated ASHMEM is not used again by the content providers, and only the receiving process accesses and utilizes the shared memory. This means that M+ does not have to care about its synchronization. We will discuss more about ASHMEM in Section 10.

6.2 Remote App Registration

Client and server processes typically require some information (so-called *app metadata*) about their counterpart application when they begin to interact with one another. For example, upon receiving a RPC request from a client, a server process often requires information about the client application to ensure that the client is eligible for the requested functionality by checking its permissions and/or that the client is authentic by checking its application signature.

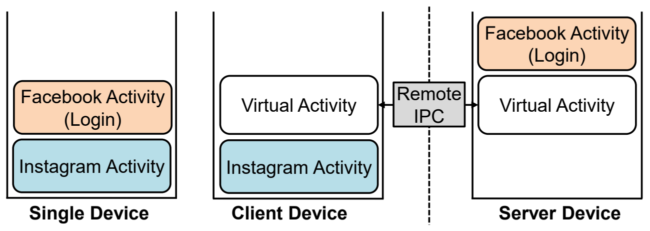
On Android, each application comes with a configuration file, called a *manifest*, that describes its metadata, such as its components, package name, permissions, and application signature. Upon the installation of an application, the package manager, which is one of the system-wide processes in Android, stores and maintains its metadata. Then, applications and system services access the metadata of their counterpart applications via the package manager.

Problems arise when a server process asks the package manager to provide metadata of a client application running on a different device. A simple solution is to intercept such a request, forward it to the client device, and bring the metadata over the network. However, this solution will be inefficient when such requests are made several times.

For this reason, M+ performs what we call *(remote) app registration* to make app metadata available between different devices. During app registration, the package manager brings the metadata of an application from a different device, where the application was installed. The package manager is then able to store and manage the metadata of the remote application, providing it to local applications on the same device. We note that an application can be installed once on a home device and may be registered remotely to any different device.

Client and server applications are remotely registered at different times. Upon device discovery and secure channel setup, two devices exchange a list of their own server applications and server applications are registered to the other device. On the other hand, a client application is registered to a different server device when it makes the first RPC request to a server device. App registration requires user agreement to proceed, the same as app installations do. This is explained further in Section 7.

[3]As another example, ASHMEM is also used to manage Dalvik machines, but it is beyond the scope of this paper.



(a) Single-device (b) Multi-device

Figure 4: Android back stack example: Instagram launches Facebook login activity.

6.3 Cross-device Execution Management

In many cases, client applications launch server activities when executing server functionalities. Here, a new execution context is generated, and it should be managed properly according to the execution semantics.

One of the most important kinds of context information is the caller-callee relationship. This information is particularly useful for many reasons. The callee typically wants to identify and verify who is making a request in order to ensure it is legitimate and to pass the RPC results to the caller correctly. In order to maintain such a caller-callee relationship, the activity manager in Android uses a stack (the back stack) to arrange a sequence of activities in the order in which each activity is launched. Figure 4(a) shows an example, where an activity of Instagram launches the Login activity of Facebook on the same device. In addition to the caller-callee relationship, the stack is also useful for indicating which activities are currently active or paused; the activity at the top of the stack is active, and all the others below it are paused.

Execution semantics can broadly fall into two categories: sequential and concurrent executions. While the former indicates a situation where caller and callee activities should be executed one by one synchronously to guarantee correct behavior, the latter allows a situation where caller and callee can execute concurrently. It is important to preserve such execution semantics when supporting cross-device RPC execution.

In the case of sequential execution, when a new activity is launched on the same device, it belongs to the same back stack, as shown in Figure 4(a). When it comes to multiple devices, however, caller and callee activities are placed in separate stacks across different devices. For example, the Instagram activity remains in the stack on a client device, and the Facebook Login activity is placed onto a new stack on a server device. This causes a couple of problems. First, the two back stacks of the client and server devices are independent, and they do not capture the caller-callee relationship. Second, the Instagram activity remains at the top of the stack, and the activity manager considers it to be active. That is, the Instagram activity will not be paused but will continue to execute even before receiving a login result from the Facebook activity. This may yield incorrect execution behavior. One solution is to re-design the activity managers in a way that they collaborate each other by exchanging execution contexts across devices. However, this requires a significant amount of modifications to existing activity managers, which is not good for ease of deployment.

For these reasons, M+ employs the concept of *virtual activity* to address the above problems, while incurring little change to activity managers. A virtual activity is an artificial activity to maintain the caller-callee relationship under the sequential execution semantics across different devices. As shown in Figure 4(b), virtual activities are added to each back stack of the client and server devices. Here, virtual activities play two major roles. First, a virtual activity is pushed to the top of the Instagram activity to keep the Instagram activity in the paused state. This allows the Instagram activity to wait to receive a login result from the Facebook activity. Second, it builds a chain of caller-callee relations across the two devices. In this chain, the virtual activity on the server side has identifier information about the client activity (Instagram). Thus, the Facebook activity can identify what has been launched. In addition, through this chain, the Facebook activity can send login results to the Instagram activity. That is, the Facebook activity sends a return value to a virtual activity on the server side, which forwards the value to S-M+ such that it is finally forwarded to the client device.

On the other hand, in the concurrent execution case, users can execute caller and callee activities independently. The activity manager puts the callee activity into a new separate back stack, which its focus is set to. This causes the callee activity to be active, while all activities in the other stacks remain paused. The user can switch the focus to the caller activity manually by pushing the menu button. This way, the user can alternate the execution between caller and callee activities on a single device. However, in multi-device environments, M+ is able to run caller and callee activities truly concurrently across different devices. As an example, consider a user reading an email on a smartphone. The user can launch the activity of a PDF Viewer on a tablet to open a PDF file attached to the email. She can then continue scrolling through the email for further reading and/or start writing an email reply on the smartphone while reading through the PDF file on the tablet at the same time. In this way, M+ offers a new user experience through an unprecedented way of using unmodified applications over multiple devices.

7. ENHANCING CROSS-DEVICE SECURITY

In this section, we discuss a fundamental difference between M+ vs. Android from the permission system point of view and describe how to extend it to close the difference. In addition, because M+ uses the network as the medium for data transmission, we describe what kind of new threats are introduced and how M+ mitigates the vulnerability derived from the network.

7.1 Cross-device Permission Check

Android uses a permission-based security mechanism to restrict the types of operations (or functionalities) that each application is allowed to perform, and so do many other popular mobile operating systems, including iOS and Windows Phone. Because each Android application operates in a process sandbox, applications must acquire the permissions for additional capabilities needed to share resources and data beyond the sandbox boundary. Because the Android permission system is designed for a single device environment, M+ extends it to a multi-device environment.

In Android, a client process is normally trusted when accessing the resource provided by another process. This is because the same stakeholder is responsible for the client process imposing a risk of misusing or abusing the resource, and is exposed to the risk. In M+, however, the client process is no longer be fully trusted when accessing resources across devices. The user of a server device can be vulnerable to the risk while little information is available about the client process.

During app registration, M+ determines whether or not to allow the client's permission request at the discretion of server device users. When a remote client appears to utilize server functionality for the first time, M+ prompts the server device user for consent. M+ informs the user of the remote server process, including what kind of application it is and which device it comes from; then asks the user to grant the permissions that the client application requires. For backward compatibility, M+ considers the client application to require the same set of permissions as the one it required when installed on its local device; such permissions are declared in a configuration file, called a manifest file.

Once a client application is registered to a server device, M+ allows server processes to carry out permission checks for the client application as if it were a local application installed on the server device. In the Android permission system, upon receiving a request from a client application for a particular functionality, a server process may ask the package manager to get further information about the client application. For example, Android system services (e.g., camera service) will check the package manager the relevant permissions of the client applications before allowing them to use the functionalities. Server applications (e.g., Facebook for Login) may also ask the package manager to get the relevant information about client applications, such as application signatures. In M+, both existing Android applications and system services can ask the package manager the relevant permissions or information about remote client applications in the same way as the above, while no modification is required to both client and local processes.

One of the key differences between Android and M+ is that M+ allows applications to register to remote server devices. This makes it possible for a single server process to interact with multiple instances of the same application, while each instance is either installed or registered to the server device. Because each instance can be individually granted or not granted during its installation or registration, it is important to distinguish each instance properly to do the permission check correctly. Because M+ assigns a unique UID to each individual application instance at its installation or registration, server processes can successfully identify individual application instances by calling `getCallingUID()`, which returns the UID of the caller process. Because the binder driver was in charge of returning the UID, we slightly modified Android `Parcel` class and `getCallingUID()` to deliver the remote client's UID instead of M+'s UID.

It is worthwhile to mention that some existing Android applications recognize applications by their package names. This works successfully in the Android single-device environment, because two or more instances of the same application cannot be installed on the same device. However, package names are no longer distinct identifiers in the M+ multi-device environment. Thus, some applications may have to change their logic to use UID for identifying application instances in the M+ multi-device environment. To this end, M+ provides an additional API, `getCallingActivityUID()`, that returns the UID of a client (caller) activity.

7.2 Network Security

Because M+ uses the network as the medium for communication and data transmission, the network can be a source of vulnerability. Therefore, various network-based attacks have presented as challenges. Packet sniffing attacks capture the wireless traffic and export sensitive data (i.e., phone number, ID and password). Man-in-the-middle attacks (MitM) [17] can eavesdrop or possibly change the communication between a client and server believed to be communicating with each other. Also, adversary can operate the server device maliciously by a packet replay attack [16] that repeatedly transmit valid data. M+ has to be safe against attacks such as these.

M+ leverages the popular Secure Sockets Layer (SSL) protocol [11] to mitigate the vulnerability derived from the network. It is a challenge to establish an SSL channel without requiring public key infrastructures (PKI) and public key servers. Therefore, we use a pre-shared password for device authentication and secure public key exchange. Although we can simply transmit data encrypted by the shared password, SSL provides advanced security features such as Perfect Forward Secrecy [5, 30]. With the pre-shared password, M+ provides security channels similar to WiFi Protected Access with advanced encryption standard (AES) encryption [41].

To establish an SSL channel, a client device needs to know the network address of its server device. Here, we assume that the client has already discovered the server and knows its address. As described in Section 10, this assumption makes sense, because device discovery can be easily done with existing technologies.

8. EVALUATION

We implemented an M+ prototype in Android and demonstrated its complete operation of functionality sharing for unmodified applications across multiple devices. M+ seamlessly enables Android applications to share a wide range of functionalities. For compatibility, the prototype was tested to see whether it works between different versions of Android.

We measured the performance and power consumption of M+. The M+ implementation used for our measurements was based on Google Nexus 6 with the Android open source project (AOSP) version 5.0.1, unless otherwise stated. During the measurements, we enabled all cores to run at the maximum CPU frequency (2.65 GHz), and all devices were connected to the same Wi-Fi access point. This connection had a throughput of 24.9 Mbps and a round-trip time (RTT) with the median, average, and standard deviation of (61.55, 62.80, and 37.69) ms, respectively.

8.1 Functionality Coverage

We first explore how broad a range of functionalities M+ can support unmodified applications. We installed and ran twenty-seven different applications from the listing of top free Android applications from Google Play, including Facebook, the top social media application.

Type		Client ↔ Server	Functionality	Coverage	Max parcel size (bytes)
Activity		Instagram ↔ Facebook	Login	○	2,180
		Candy crush ↔ Facebook	Login	○	2,196
		Line Ranger ↔ Line	Login	△	604
		Gmail ↔ Adobe Acrobat Reader	PDF file viewing	○	680
		SMS ↔ Browser	Web page browsing	○	588
		Browser ↔ Android System UI	Web page sharing	○	190,372
		Gallery ↔ Facebook	Photo sharing	○	1,160
Service	Application service	Real racing 3 ↔ Google play store	Payment	○	39,316
		Candy crush ↔ Google play store	Payment	○	10,352
	I/O system service	Real racing 3 ↔ Sensor service	Game control	○	5,228
		Ladybug ↔ Sensor service	Game control	○	5,228
		Camera2Basic ↔ Camera service	Camera preview	○	15,512
		Body messager ↔ Vibrator service	Vibration	○	204
	Non-I/O system service	Chrome ↔ Clipboard service	Copy & Paste	○	1,540
		SMS ↔ Notification service	Message notification	○	1,500
		Clock ↔ Notification service	Alarm notification	○	4,324
Content Provider		Google drive ↔ Downloads	File download	○	1,956
		Dropbox ↔ Downloads	File download	△	2,528
		True phone ↔ Contacts	Contacts management	○	1,936
		Simple contacts ↔ Contacts	Contacts management	○	1,936
		Google messenger ↔ Telephony	SMS management	○	1,500
		ZeroSMS ↔ Telephony	SMS management	○	1,500

Table 1: Use case list for the functionality coverage test. Coverage is marked with ○ if M+ supports the use case correctly or △ if it does so with user's proper choice.

Table 1 shows a list of use cases, where client applications utilize various functionalities from server applications or Android services on different devices. The fourth column (labeled "Coverage") shows whether M+ supports each use case. As shown in the table, M+ successfully supports cross-device functionality sharing for a wide range of functionalities, except the two use cases marked with triangles (△).

Those two use cases employ multiple independent binder connections while sharing a single functionality. Line, one of Japan's popular messengers, returns an access token as a login result through a separate binder connection that it initiates with a new seed parcel. Dropbox also uses two binder connections initiated by two seed parcels, one for downloading a file and the other for naming the file. Those two binder connections should be made between the same client and server pair for correct operation. When a new binder connection is requested with a seed parcel, a user is prompted to select the right destination if there are multiple candidates. Confusion may arise when there are multiple instances of the same app across several devices. For example, when Line initiates a new binder connection to pass an access token in response to Line Ranger's login request, the user of the Line device may be asked to select the right instance of Line Ranger among multiple across different devices. If the user makes the right choice, M+ can work correctly. However, if the user makes a wrong decision, such a use case would not work properly. We will discuss possible solutions in Section 10.

8.2 Performance and Power Consumption

We quantitatively evaluate performance and power consumption of M+ in supporting cross-device functionality sharing. We generated four types of micro-benchmarks depending on which IPC connections are used: binder, URI, Unix domain socket (UDS), and ION memory. In each micro-benchmark, a client utilizes the functionality of a server through one of the IPC connections. We ran each micro-benchmark within a single device (labeled "LOCAL") and across two devices with M+ (labeled "M+ "). Unless otherwise stated, we repeated each experiment ten times and herein report the average and standard deviation of these measurements. We also report the overhead from using the SSL connection.

8.2.1 *Performance*

Interception overhead: The time to intercept a parcel for decision making can be considered as *interception overhead*, if the destination of the parcel is a server process on the same device. In order to measure such overhead, we wrote a micro-benchmark program that repeated launch of a simple activity via the `startActivity` method 100 times. Our experiment results show that it takes 17.79 ms on average (standard deviation of 8.99 ms) to run the simple activity on stock Android, and 20.23 ms on average (with a standard deviation of 19.51 ms) with M+. Thus, the interception overhead of M+ is 2.44 ms on average. Though such an overhead is not negligible, its impact on the overall sys-

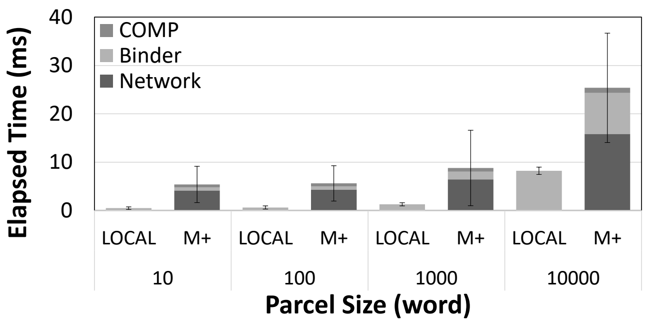


Figure 5: Performance of binder communication. The X axis is size of a parcel in words; each is four bytes in Android.

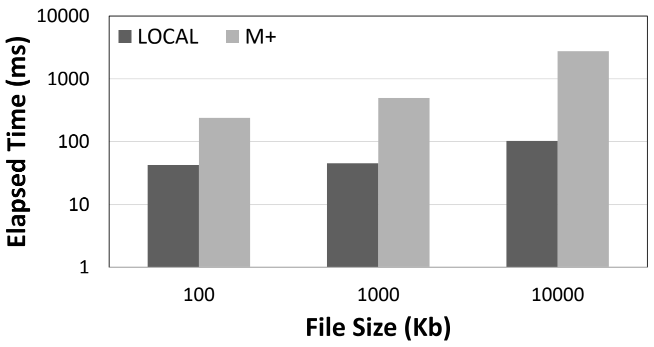


Figure 6: Performance of communication using URI for file. Note that the Y axis is log-scaled.

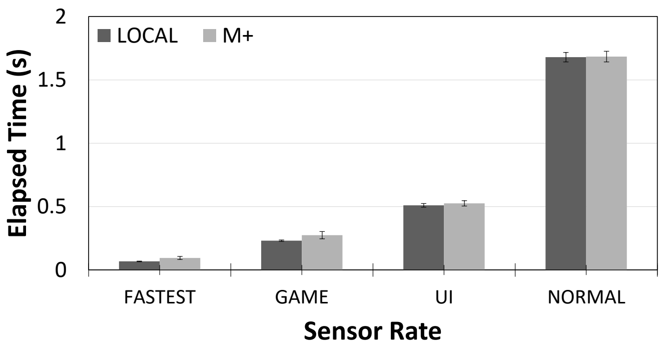


Figure 7: Performance of streaming accelerometer sensor data. X axis shows the registration type declared in Android Sensor Manager. M+ has sufficient throughput (maximum of 90 Hz).

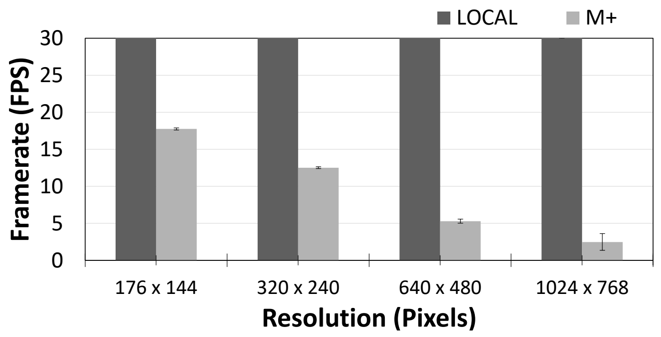


Figure 8: Performance of a real-time streaming camera preview with a 24.9 Mbps wireless LAN connection between the client and server. As the wireless network speeds up, performance will improve without requiring changes to M+.

tem performance is quite limited since M+ intercepts only seed parcels, which occupy a very small portion of the whole parcel communication.

Binder: Figure 5 shows the time between when a client sends a binder parcel to a server and when the client receives a return signal from the server, as a function of the amount of data included in the parcel. It also shows the percentage breakdown of average elapsed times. We broke down the time into three stages: (1) "COMP " involves spending time in C-M+ and S-M+ to intercept and determine its destination, (2) "binder" involves the time spent in the binder communication between Java applications and the binder driver, including the time to copy data to the server's memory space, and (3) "network" involves the time spent on cross-device communication via M+. The figure shows that it takes longer to send a larger amount of data through binder and network, while network delays are the most significant factor. It is also shown in the figure that M+ does not impose much computation overhead. We note Table 1 shows the maximum parcel size of each use case.

File: Figure 6 shows the average time between when a client sends a uniform resource identifier (URI) to share a file and when the client receives a return signal from the server, including the time for the server to read the file. As the file size increases, the performance gap between LOCAL and M+ increases due to file transfer costs. Given that the performance of file sharing is bound by the network bandwidth, it will continually improve as wireless technologies evolve. Similar performance characteristics are observed for anonymous shared memory (ASHMEM).

UDS: To evaluate the performance of M+ when using UDS, we selected a sensor service that uses UDS internally to transfer sensor data streaming. We measured the average time it took for a client to obtain the first 10 sensor readings since sensor callback is registered. We registered accelerometer sensor callbacks at four different sensor rates, which are pre-defined in Android Sensor Manager: FASTEST, GAME, UI, and NORMAL. Each of them has a target sensor delay of (0, 20, 60, and 200) ms, respectively. Figure 7 shows the elapsed time according to the sensor rate. M+ was shown to have comparable performance with the single-device case, except when the sensor rate is FASTEST.

ION memory: We evaluated the performance of M+ when using ION memory. We wrote a camera microbenchmark that uses a real-time streaming camera preview. In each experiment, we measured the average frame rate (frames per second) that could be achieved during the first 1000 frames, excluding the first 50 frames to avoid the effect of camera initialization. Figure 8 shows M+ can achieve acceptable performance (i.e., > 13 FPS) at low resolutions (i.e., 176×144 and 320×240). At higher resolutions, however, M+ exhibited performance degradation. To maintain 15 FPS for the NTSC (i.e., 720×480) or higher resolution, it requires at least 89 Mbps of network throughput to transmit more than 760 KB of data per frame. We expect that M+ will support 15 FPS for HD resolutions (i.e., 1280×720 and 1920×1080) by taking advantage of future wireless net-

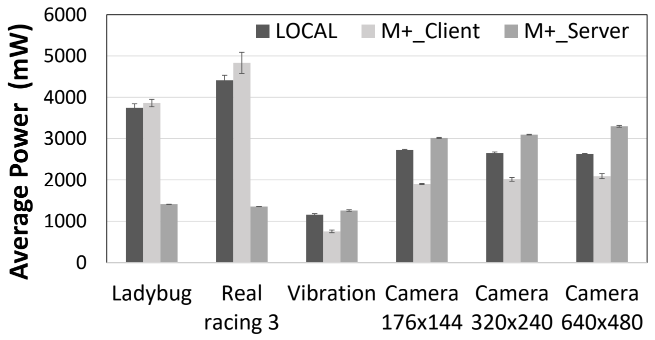


Figure 9: Average system power consumption of when sensor, vibrator, and camera (for preview streaming) are used remotely with M+ and LOCAL. Note that the numbers below `Camera` represent resolution.

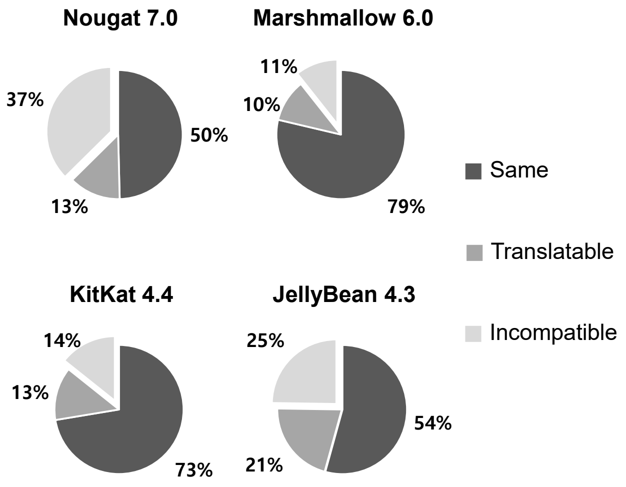


Figure 10: Compatibility study between Android Lollipop 5.0.1 and other latest Android versions. We compared the RPC APIs used by the functionalities shown in Table 1.

working standards. For example, 802.11ad can reach around 7 Gbps of throughput. In addition, M+ can be further extended to maintain acceptable FPS on high resolutions by incorporating video encoding/compression techniques.

8.2.2 *Power Consumption*

We evaluated the power consumption of M+ compared to a single-device case. For proper evaluation, we selected only the I/O system services shown in Table 1. This is because applications typically utilize such I/O system services continuously consuming a significant amount of energy while the other use cases utilize functionalities only once in a while, and incurring negligible additional power consumption. We ran each experiment for one minute and measured the average power consumption of a mobile device using the Monsoon Power Monitor [8][4]. During each experiment, we turned on the display of all the devices and set the brightness to 50% which consumed 819 mW on average.

Figure 9 compares the power consumption of different applications on three different devices: a single device with stock Android, a client device with M+, and a server device with M+. It is shown that M+ consumes 1.5× to 2× the power consumption by the local device, depending on which functionality to use. The first two applications, Ladybug and Real racing 3, are 3D game applications that make use of continuous sensor readings. Those clients perform heavy computation (e.g., 3D image rendering) and consume a little more energy with M+, mostly for additional cross-device communication, while incurring much less power consumption on servers. On the other hand, other client applications, such as Vibration and Camera, are shown to impose even slightly more power consumption on servers than on the local device. This is because some computation, such as camera preview encoding, takes place on servers.

8.2.3 *SSL Overheads*

As explained in Section 7.2, M+ is able to establish a secure SSL connection between client and server devices. Here, we evaluate the overheads of using the SSL connection in terms of performance and power consumption. Using the SSL connections incurs overheads in encrypting and decrypting data to transfer. During our experiments, using SSL connections degraded performance by 3.1%, 6.8%, 4.5%, and 51.7%, when using binder, file, sensor, and camera, respectively. On average, the client uses 7.3% more energy and the server consumes 5.9% more energy when using SSL connections, compared to the case of using M+ without SSL.

8.3 Compatibility Study

In this subsection, we discuss the compatibility of M+ with different versions of Android. Because M+ allows client applications to utilize the functionalities of server processes through cross-device RPC, the compatibility of M+ depends solely upon that of RPC application programming interfaces (APIs) between different Android versions. To explore such compatibility, we conducted a comparison of the five latest Android versions for the RPC APIs used by all the functionalities shown in Table 1. Specifically, we investigated 173 RPC functions in 16 RPC interfaces, including `IServiceManager` and `IActivityManager`, to study the compatibility of Android Lollipop 5.0.1 with the four other latest versions.

Figure 10 shows the compatibility of the RPC APIs between different Android versions, reporting 50% to 79% of identical RPC APIs. We used the `diff` tool to examine how many RPC functions have exactly the same APIs between different Android versions. The figure shows that more than 70% of the RPC functions examined have identical function names and arguments between Lollipop and its closest versions (KitKat and Marshmallow), and the numbers drop to 50% when they are two versions away.

Figure 10 also shows that the compatibility increases by 10% to 21% with a trivial effort. We analyzed the RPC functions of non-identical APIs to see whether they can be translated in a trivial way. For example, we can use zero or null for any additional arguments or combine multiple arguments into a single argument without violating the semantics. We translated such RPC arguments and verified

[4]We note that since Nexus 6 and its battery are integrated, it is not trivial to use the Monsoon tool for measurement. We thus measured the power as follows. We first disconnected the battery by breaking the two pins corresponding to (+) and (-) terminals on the circuit. We then soldered wires to pads on the circuit that are connected to the (+) and (-) terminals and connected the wires to the Monsoon.

that they worked correctly between different Android versions; and then classified them as *translatable*. The figure shows that Lollipop is easily compatible with other Android versions with an average of 78.25% of the RPC APIs, when including translatable ones. Some RPC APIs are inherently incompatible, because some functionalities are available only on newer or older Android versions or do not support backward compatibility with older versions.

In addition, we ran each individual use case shown in Table 1 while client and server devices ran on Lollipop and Nougat, respectively. All use cases but three successfully worked between different Android versions. The three did not work when they utilized camera and notification services through the RPC interfaces of `ICameraServer, ICameraDeviceUser`, and `INotificationManager`. Nougat came with a number of new camera RPC interfaces and significant changes to notification RPC interfaces with new user interface.

It is worth noting that the compatibility of application functionality does not depend on the Android version. Applications are compatible without problems if they are all up-to-date versions via the Google Store. Note that even a client cannot use a server's functionality on a single device if the RPC interface between client and server is different.

9. RELATED WORK

Multi-device mobile platform: Support of mobile applications in multi-device environments has been receiving increasing attention. One theme in the literature is seamless migration of execution of a single app between devices. There have been several studies such as MAUI [19], CloneCloud [18], Odessa [35], COMET [20], ColPhone [38] and Lee et al. [28]. The focus of these papers is on automatic execution partitioning and migration of applications onto nearby smartphones and cloud servers to save energy and improve performance. Recently, Flux [42] leverages app migration to switch from a mobile device to another. Sapphire [43] provide a framework that supports distributed execution of the same app on different mobile devices. However, all the aforementioned studies do not consider supporting an app running on one device to utilize functionalities on another device.

Another recent theme is cross-device functionality sharing. There are many pieces of work that support remote I/O resource access for a specific class of I/O, such as remote file systems [14, 27, 37], network USB [22, 23, 13, 21], sensors [24, 40], and remote printers [4]. Rio [39] is the closest work to ours, and allows sharing of various I/O resources across devices by virtualization at device file layer with distributed shared memory. Compared to Rio, M+ provides a platform-layer solution for cross-device functionality sharing, with the following differences: (i) it supports a wider range of functionalities for cross-device sharing, including application functionality and Non-I/O system services, (ii) it extends single device execution management logic to cross-device, and (iii) it provides a cross-device permission check.

Android RPC: AndroidRMI [25] considers extending Android binder IPC to support remote method invocation. Yet, its solution requires modifying existing applications by extending their Android interface definition language (AIDL) interface description. Compared to AndroidRMI, M+ handles reference types for RPC arguments and extends execution management logic so that unmodified applications can still use RPC interactions across different devices. Thus, M+ supports a much wider range of applications. Nakao et al. [33] proposes an extension to the intent class for remote service invocation at the application layer in Android, while supporting screen sharing only by using VNC remote screen sharing middleware [3].

App-level I/O resource sharing: Many apps are available for sharing specific I/O resources, including IP Webcam [26] for camera, WiFi Speaker [32] for audio, and MightyText [6] for SMS and MMS messages. Several apps support screen sharing, such as Miracast [7], by allowing streaming a screen from one device to another. VNC [15] and Microsoft Remote Desktop Services [31] employ thin client solutions instead of resource sharing. The above approaches develop cross-device sharing atop low-level APIs, coming with high development cost. On the other hand, M+ provides a platform that reduces the burden of developing applications for I/O resource sharing across multiple devices. This extends app capabilities to provide novel user experience.

10. DISCUSSION

Device discovery: In this paper, devices are assumed to know the addresses of others. We can leverage existing techniques to relax this assumption. The DNS-SD [2] protocol uses multicast DNS [9, 10] that performs DNS queries over IP multicast on a local area network with no conventional DNS server installed. When there is no common local WiFi network around, WiFi Direct [12] can be used to facilitate device discovery and data transmission. Using these techniques, M+ can easily find out the address of the device to which it wants to connect.

Mitigating compromised M+: In this work, we assume that M+ is not compromised. If so, privacy-sensitive data can be leaked to unauthorized parties. One may check the integrity of M+ through remote attestation [34, 36] to avoid interaction with compromised devices. This will be a topic of future work.

Supporting ASHMEM reuse: We could not find any case such that ASHMEM is reused while experimenting for twenty-seven different applications (in Section 8.1). If there are some applications that reuse ASHMEM under their custom synchronization protocols, the current prototype of M+ cannot handle them. However, if M+ uses a distributed shared memory (DSM) technique as Rio [39] does, it is possible to support ASHMEM reuse in the future.

Non-standard way of cross-device interaction: The design of M+ cannot support apps across devices if they interact with each other through private channels rather than using the standard binder IPC. For example, apps may communicate over a shared file (such as a common data file in *tmp* directory). However, in our experience, such communication is not popular because it requires tight coupling between the apps which are typically developed independently.

Multiple seed parcels: As discussed in Section 8.1, applications can share a single functionality with multiple binder connections each of which is initiated by an individual seed parcel. In this paper, M+ assumes that users, when prompted, will determine the right destination in the presence of multiple instances of the same application across several devices. There are two possible solutions. M+ will be able to address such a case correctly without relying on users, if (i) application developers provide information about the dependency between seed parcels, or (ii) applications distinguish different app instances individually by device IDs and package names.

Applying M+ to other mobile platforms: Although M+ is specialized for Android, the general design of M+ is applicable to other mobile platforms. M+ was designed under the key assumption that applications utilize the functionalities of other processes through RPC over well-defined IPC mechanisms. This assumption is a common mobile platform design paradigm (e.g., XPC and Mach messages in iOS [29]). Supporting transparent cross-device functionality sharing over such mobile platforms will face the following three challenges; it must i) extend execution management logic, ii) handle multiple instances of the same application, iii) extend the single-device IPC mechanism to cross-device. For the first challenge, the design of M+ will be typically applicable in managing execution context and semantics across multiple devices. For the second, the concept of remote app registration will be helpful to manage multiple app instances. For the third, M+ employs a functionality-agnostic approach leveraging the following three binder characteristics. Binding is made through a proxy and a stub, new binder connections are generated through seed parcels, and there are some standard reference types for RPC arguments. If such characteristics are not provided, engineering effort can be made to address the challenge in a functionality-aware manner.

11. CONCLUSION

We presented M+, an extension of Android that supports a range of cross-device functionality sharing. With M+, unmodified applications can utilize application and system functionalities across devices, when they use RPC based on the binder IPC mechanism. M+ also provides secure connection and performs cross-device permission check for both local and remote applications in the same way. In addition, M+ distinguishes multiple instances of applications, which has not been considered in existing mobile platforms. Our experiment with the M+ prototype demonstrated that it achieves performance close to local unless data transfer is large. We expect the cross-device sharing (M+) platform to accelerate development of creative and useful applications to provide novel user experience.

Acknowledgements

This work was supported in part by BSRP (NRF-2015R1D1A1A01058713), IITP (B0101-15-0557), and NRF(2015M3A9A7067220) funded by the Korea Government (MEST/MSIP/MOTIE). We also thank our anonymous reviewers for their insightful and constructive comments that helped us improve this paper.

12. REFERENCES

[1] A number of connected devices in 2016. https://www.globalwebindex.net/blog/digital-consumers-own-3.64-connected-devices.

[2] DNS-Based Service Discovery. https://tools.ietf.org/html/rfc6763.

[3] Droid VNC Server. https://github.com/oNaiPs/droidVncServer.

[4] Epson Remote Print. http://goo.gl/f7EdUh.

[5] Forward Secrecy. https://en.wikipedia.org/wiki/Forward_secrecy.

[6] MightyText: SMS Text Messaging. https://goo.gl/oLXH0T.

[7] Miracast. http://www.wi-fi.org/wi-fi-certified-miracast.

[8] Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[9] Multicast DNS. http://www.ietf.org/rfc/rfc6762.txt.

[10] NsdManager. http://developer.android.com/reference/android/net/nsd/NsdManager.html.

[11] The transport layer security (tls) protocol version 1.2. https://tools.ietf.org/html/rfc5246.

[12] Wi-Fi Direct. http://www.wi-fi.org/discover-wi-fi/wi-fi-direct.

[13] Wireless USB. http://www.usb.org/developers/wusb/.

[14] Network file system (nfs) version 4 protocol. *IFTF Network Working Group RFC Draft*, 2003.

[15] T. R. andQ. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 1998.

[16] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Replay attack in tcg specification and solution. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 11–pp. IEEE, 2005.

[17] F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security and Privacy*, 7(1):78–81, 2009.

[18] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proc. ACM EuroSys*, 2011.

[19] E. Cuervo and A. Balasubramanian. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. ACM MobiSys*, 2010.

[20] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *Proc. USENIX OSDI*, 2012.

[21] A. Hari, M. Jaitly, Y. Chang, and A. Francini. The Swiss Army Smartphone: Cloud-Based Delivery of USB Services. In *Proc. ACM MobiHeld*, 2011.

[22] D. International. AnywhereUSB. http://www.digi.com/products/usb/anywhereusb.jsp.

[23] D. International. USB Over IP. http://usbip.sourceforge.net/.

[24] Y.-W. Jong, P.-C. Hsiu, S.-W. Cheng, and T.-W. Kuo. A semantics-aware design for mounting remote sensors on mobile systems. In *Proceedings of the 53rd Annual Design Automation Conference*, page 140. ACM, 2016.

[25] H. Kang, K. Jeong, K. Lee, S. Park, and Y. Kim. Android RMI: A User-Level Remote Method Invocation Mechanism between Android Devices. *The Journal of Supercomputing*, 2015.

[26] P. Khlebovich. IP Webcam. https://goo.gl/FQgQst.

[27] P. J. Leach and D. Naik. A common internet file system (cifs/1.0) protocol. *IFTF Network Working Group RFC Draft*, 1997.

[28] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim. Architecture-aware automatic computation offload for native applications. In *Proceedings of the*

48th International Symposium on Microarchitecture, pages 521–532. ACM, 2015.

[29] J. Levin. *Mac OS X and iOS Internals: To the Apple's Core*. Wrox, 2012.

[30] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[31] Microsoft. Remote Desktop Services. https://technet.microsoft.com/en-us/windowsserver/ee236407.aspx.

[32] W. Morrison. WiFi Speaker. https://goo.gl/N128Ar.

[33] K. Nakao and Y. Nakamoto. Toward Remote Service Invocation in Android. In *Proc. Ubiquitous Intelligence & Computing/Autonomic & Trusted Computing*, 2012.

[34] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert. Beyond kernel-level integrity measurement: enabling remote attestation for the android platform. In *Proceedings of International Conference on Trust and Trustworthy Computing*, 2010.

[35] M. Ra, A. Sheth, L. Mummert, P. Pillai, d. Wetherall, and R. Govidan. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *Proc. ACM MobiSys*, 2011.

[36] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, et al. fTPM: A firmware-based tpm 2.0 implementation. Microsoft Research Technical Report (MSR-TR-2015-84), 2015.

[37] A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh. RFS Architectural Overview. In *Proc. USENIX Conference*, 1986.

[38] A. Salem and T. Nadeem. ColPhone: a smartphone is just a piece of the puzzle. In *Proc. ACM MobiCom*, 2014.

[39] A. A. Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A System Solution for Sharing I/O between Mobile Systems. In *Proc. ACM MobiSys*, 2014.

[40] C. Shen, R. P. Singh, A. Phanishayee, A. Kansal, and R. Mahajan. Beam: Ending monolithic applications for connected devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016.

[41] I. Standards. *IEEE 802.11i-2004: Amendment 6: Medium Access Control (MAC) Security Enhancements*. 2003.

[42] A. Van't Hof, H. Jamjoom, J. Nieh, and D. Williams. Flux: Multi-surface computing in android. In *Proceedings of the Tenth European Conference on Computer Systems*, page 24. ACM, 2015.

[43] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and Extensible Deployment for Mobile/Cloud Applications. In *Proc. USENIX OSDI*, 2014.