SAMD: Fine-Grained Application Sharing for Mobile Collaboration

Jaehun Lee*, Hochul Lee*, Byoungjun Seo*, Young Choon Lee†, Hyuck Han‡ and Sooyong Kang*
*Dept. of Computer Science, Hanyang University, Korea, Email: {ljhokgo,lhochul2,sbj8388,sykang}@hanyang.ac.kr
†Dept. of Computing, Macquarie University, Australia, Email: young.lee@mq.edu.au
‡Dept. of Computer Science, Dongduk Women's University, Korea, Email: hhyuck96@dongduk.ac.kr

***Abstract*—The collective use of ever connected and pervasive mobile devices has been increasingly sought for in mobile collaboration, such as multiplayer mobile gaming and distributed processing. The current model of mobile collaboration requires each device to install a particular, 'full' mobile *app* for a respective collaboration. Besides, collaboration functionalities are typically implemented at application level. In this paper, we present *Single Application Multiple Device* (SAMD) as a platform-level mobile collaboration framework. A mobile app developed using SAMD is capable of fine-grained application sharing. In particular, SAMD enables devices, agreed to participate in collaboration, to get portions of the app on-the-fly and run them without the prior installation. To achieve this, we have developed three solutions as core functionalities of SAMD: 1) *Controller* packaging, 2) lookahead transfer and 3) code adaptation. We have implemented SAMD on Android as a proof-of-concept prototype. Our experimental results demonstrate SAMD can provide fine-grained sharing of latency-insensitive applications.**

I. Introduction

In the past decade, we have witnessed the tsunami of so-called "smart" device penetration into every part of our lives from web surfing to online banking and health check. These devices including smart watches, smartphones, and tablets are becoming increasingly resource rich equipping with, for example, several sensors, GPS modules and even GPUs. As a consequence, there emerged a trend to collectively and collaboratively use these devices, i.e., *mobile collaboration*. Example applications are multiplayer mobile games, collaborative document editing and some form of distributed processing.

Despite this growth in quantity, capacity and capability, the platform-level support for mobile collaboration remains limited. The current model of mobile collaboration is largely application-driven. In particular, a mobile collaboration is enabled by an application specifically developed for that. Such an application is often required to be installed in each and every device in the collaboration. A typical approach of communication for collaboration is using a cloud server or wifi direct, as in Clash Royale [1], Google Docs [2], SuperBeam [3] and Send Anywhere [4].

In this paper, we study the platform-level support for mobile collaboration. Motivated by the fact that a mobile app is considered to be a collection of Controllers[1], we propose to dynamically send Controllers/portions of an app to other devices for remote execution (fine-grained application sharing) in a coordinated manner, without the prior installation. This collaboration capability feature is what primarily distinguishes SAMD from Google's Instant Apps [5]. Imagine you are in a metro train station waiting for your train with a few friends of yours. This instant and fine-grained application sharing will let you play some multiplayer mobile game—that only you have on your phone—with your friends.

[1]A Controller as in an Activity of Android and a viewController of iOS is a code-and-resources set of a function/portion of an app.

Our fine-grained application sharing approach has two key advantages: apps can run across multiple devices on-the-fly and collaboration apps are non-intrusive. The main challenge is threefold as it consists in characterizing the composition of an app, in deciding when to send which Controller and in executing Controllers (portions of an app) on remote devices. Above all, these challenges have to be addressed at mobile platform level to ease collaborative application development.

Our solution to these challenges is called Single Application Multiple Device (SAMD) with corresponding functionalities to those three challenges: *Controller packaging*, *lookahead transfer* and *code adapation*. SAMD manages an application as a set of Controllers to dynamically package, send and execute individually in remote devices. In particular, an application developed using SAMD (or simply a SAMD app) is decomposed and *individually* packaged at Controller level. Controllers to be executed on other devices, in which the application is not originally installed, are dynamically transferred prior to their actual execution. The actual remote execution is realized by adapting code and resources (e.g., images) of an individual Controller and plugging the adapted Controller into the dummy *agent app* of SAMD.

We implement SAMD in Android as a proof-of-concept prototype and demonstrate the development of an example SAMD app (http://dcslab.hanyang.ac.kr/samd/). Unlike other multi-user applications, a SAMD app can be easily developed by using APIs SAMD provides. For instance, the discovery of (SAMD-enabled) remote devices is enabled by the device management API of SAMD. Devices approved a collaboration request are only be able to communicate with each other.

The specific contributions of this paper are:

- We model/formalize platform-level mobile collaboration.
- We develop three core functionalities of SAMD.
- We implement a SAMD prototype on Android and demonstrate its capacity with *TexasHoldem* board game.
- We evaluate SAMD with five application scenarios.

Experimental results have demonstrated the feasibility and capacity of SAMD. In particular, we have evaluated SAMD in a testbed with a number of Nexus phones of three models. The evaluation study is conducted with respect to the latency of remote execution, the SAMD overhead on application installation, and the effectiveness of SAMD's distributed processing.

The rest of this paper is organized as follows. In Section II, we discuss related work. In Sections III and IV, we present design and prototype implementation of SAMD. Section V shows an example SAMD application development. In Section VI, we provide our discussion on two most significant aspects of mobile collaboration in the context of SAMD. In Section VII, we show the feasibility and capacity of SAMD followed by our conclusion in Section VIII.

II. Related Work

There is a large body of studies on application sharing for multi-device "collaboration". Two most common approaches are screencasting and code/application offloading. Their main distinction is whether the display information is sent or an (part of) application is migrated/offloaded, to remote devices.

Screencasting is sharing the screen of a device by transferring screen output to different devices [6]–[10]. For example, Apple Airplay [11] can be used to display iOS media contents on Apple TV. However, the application only runs on the local device, and the remote device simply shares the screen. There are some studies on sharing of particular resources like touch screen e.g., [12]. Recently, a more generic kernel-level approach for remote I/O sharing for mobile systems, Rio [13], has been proposed. However, the code still only executes on local devices; this is a clear limitation to providing a proper collaboration functionality.

Application offloading to remote devices or often more powerful servers has been extensively studied [14]–[17]. However, their application to smart devices is limited due to issues with device heterogeneity. As resources of recent smart devices have become increasingly abundant, studies on application offloading to peer smart devices have also started to appear [18]. Flux [19] uses the record and replay approach to migrate running applications at local devices to remote devices. Most, if not all, application offloading approaches require application installation in remote devices. They typically target one-to-one offloading/migration, not multi-user collaboration.

Executing a part of the code, not the entire application, through cloud servers or nearby smart devices, is where researchers are making vibrant progress. MAUI [20], CloneCloud [21], ThinkAir [22], Cloudlet [23] and femtocloud [24] offload code to cloud server to enable processing that a smart device is practically unable to perform alone. Also, there are some studies on code-level migration with sending the state of code [25]–[27]. The work in [28] proposes a framework to support mobile simulations by distributing the computation between a mobile device and a remote server. As smart devices become increasingly capable with more and new resources, studies on code-level offloading are actively conducted, e.g., Hyrax [29], Serendipity [30] and LWMR [31].

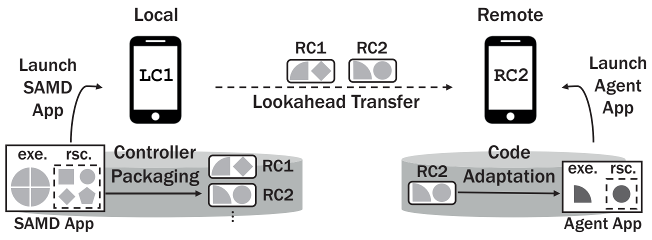


Fig. 1. SAMD design overview: We refer to the device where a SAMD app resides as a 'local' device and others running portions of the app without prior installation as 'remote' devices.

The works in these studies enable simultaneous application code executions across multiple devices. However, they are, unlike SAMD, unable to access smart device features, such as display, sensors and cameras; therefore, those are closer to distributed processing rather than collaboration.

III. SAMD: Single Application Multiple Device

In this section, we present SAMD with its three core functionalities (Figure 1): Controller packaging, lookahead transfer and code adaptation.

A. Controller Packaging

Controllers of a SAMD app are classified into Local Controllers (LCs) and Remoteable Controllers (RCs). While LCs are designed to run on the local device only, RCs are *able* to run on remote devices as well as local devices. RCs are identified during application development, e.g., by setting 'remoteable' flag, in the application configuration file, to true.

Controller packaging takes place while a SAMD app is being installed and consists of three steps: *Decomposition*, *Analysis* and *Packaging* (Figure 2). At the core of Controller packaging is code analysis after decomposing the entire code in a SAMD app into respective code segments for Controllers.

The code level analysis 1) classifies Controllers into LCs and RCs, 2) extracts every RC call from LCs and 3) identifies necessary resources for each RC. Controllers are classified based on the indication of remote execution, e.g., a remote execution API call like *launchActivity()* in our SAMD prototype on Android. In particular, a caller Controller and a callee Controller are classified as a LC and a RC, respectively.

As a result of code level analysis, a Controller call graph (*ControllerMap*, Figure 3a) is constructed. In addition, for RCs, their corresponding resources are identified and tabulated in a "key-value" like lookup table (*ResourceTable*, Figure 3b). ControllerMap is a data structure showing the *caller-to-callee* relationships from LCs to RCs. Vertices represent Controllers. If a Controller is classified as a RC (i.e., made to possibly

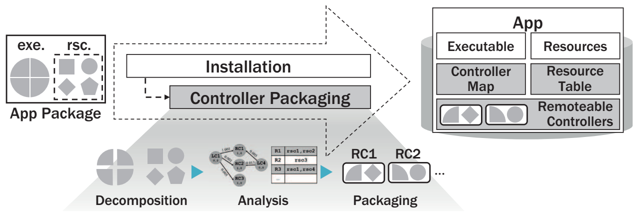
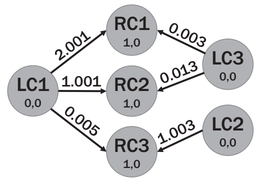


Fig. 2. Controller packaging.



(a) ControllerMap

Controller	Resources
RC1	b.xml, 1.png, c.txt
RC2	d.xml, a.png, c.txt
RC3	e.xml
...	

(b) ResourceTable

Fig. 3. Core data structures in SAMD: two numbers in each vertex of the ControllerMap represent *remoteable* and *cached* status of the Controller.

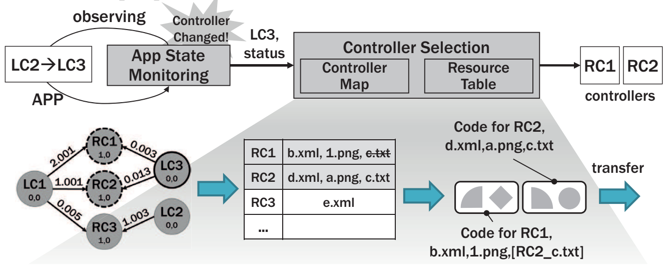


Fig. 4. Lookahead Transfer.

run on the remote device), its *remoteable* status value in the ControllerMap is set to 1. If the Controller has been transferred to remote devices already, its *cached* status is set to 1.

Edges represent call relationships with likelihoods of calls. The calling likelihood score between a LC and a RC is defined as $score(LC, RC) = n + 0.001(r + c)$, where n is the calling precedence value of RC defined as the count (from 1) of all RC calls from the end of the `main` procedure (or method in JAVA) in the LC; for example, the integer part of three RCs in Figure 3a, 2, 1 and 0 for RC1, RC2 and RC3, respectively, indicates that RC1 and RC2 are called in that order but RC3 is not called in the main procedure of the LC1. r is the number of RC calls outside the main procedure and c is the frequency of RC execution in the remote device. While remote execution requests/calls inside the main procedure are made regardless of user's behavior, those outside the main procedure are only made depending on the actions of the user and other factors. Hence, RCs called inside the main procedure should have a higher likelihood score than others. The calculation of a score incorporates the location of such calls making r and c become a fractional part of the score. We also use a scale factor of 0.001 assuming that the RC calls outside the main procedure are not expected to occur a thousand times or more.

ResourceTable is a 2-tuple table with RC IDs as the key and resources as the value. This table is constructed and used only for remoteable Controllers.

Finally, the code and resources of each RC are packaged referencing ControllerMap and ResourceTable. Resource identification and packaging are needed only to RCs for possible remote execution.

B. Lookahead Transfer

As the latency for remote execution of RCs can become a major concern, lookahead transfer in conjunction with application state monitoring determines which RCs are likely to be called upon a state change and sends appropriate RCs in advance. In particular, upon a state change, such as the execution control shift from a LC to another LC (e.g., LC2 to LC3 in Figure 4), ControllerMap is looked up to determine which RCs are likely to be executed in which device after the state change. Lookahead transfer then sends RCs imminent and yet to be called, one by one to remote devices in the order of their calling likelihood scores. The actual transfer of RCs is selective in the sense that only RCs that are likely to be executed and have never been transferred (i.e., not cached) are sent. Besides, when transferring multiple RCs to a particular remote device, same resources are sent only once. More specifically, redundant resources in succeeding RCs are removed before their transfers. In this case, the information of such resource removal (`[RC2_c.txt]` in Figure 4) is also sent for the remote device to properly run those RCs.

C. Code Adaptation

Controllers transferred to remote devices are first *depackaged*, and *repackaged* through code adaptation to suit the execution on remote devices. The depackaging process simply extracts the code and resources from a Controller and forwards them for repackaging with the agent app. At the core of repackaging is code adaptation.

In general, resources in mobile applications are installed in a compiled state to minimize the resource access latency by code. Compiled resources are referenced by their static addresses which are bound to all resources in the application at the compilation time. In SAMD, only a subset of application resources are included in a RC and transferred to remote devices for execution. Hence, for a remote device to execute the received RC, it needs not only to generate resource information to be provided to its mobile platform but also to recompile resources to bind new static addresses to them. The resource accessing parts of code in the RC are also needed to be adapted for the changed static addresses.

As RCs are only portions of an app without the prior installation in the remote device, they are not readily executable standalone. The agent app can be seen as a system app, part of SAMD, on all SAMD-enabled devices. It is a dummy app with minimal code and resources to be an application. After code adaptation, the agent app's code and resources are replaced with the adapted code and resources of the RC. Then we can execute the received RC by launching the agent app. Multiple different RCs can run in turn on the remote device by subsequently replacing agent app's code and resources.

IV. Prototype Implementation

In this section, we first give a brief description of Android application installation process and detail the prototype implementation of SAMD on the Android (version 6.0.1r11, Marshmallow) focusing on three key functionalities. In particular, the prototype is implemented primarily in the form of platform-level services (Figure 5). These services are accessed through *SAMDService* using APIs of SAMD. The description of these APIs is also provided in this section.

Android uses Android application package (APK) for application installation. From APK file, it registers application in-

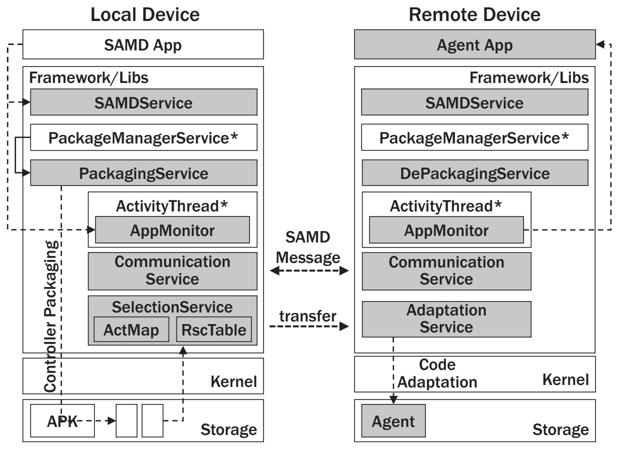


Fig. 5. SAMD components in Android platform. Components added by SAMD are highlighted in gray. Modified components are marked with '*'.

formation and extracts the executable file and resources. These extracted files and resources are allocated an appropriate directory for application installation. The process of APK file generation is as follows: compiling Java source (.class) files and resources generates `classes.dex` and `resources.arsc` files, respectively. These files are then compressed and packaged into an APK file along with `AndroidManifest.xml` which is an application configuration file.

A. Implementation of Controller Packaging

The Controller packaging functionality is implemented through primarily *PackagingService* in cooperation with Android's *PackageManagerService* as shown in Figure 5. We add `SAMDApp` and `Remoteable` attributes to <Application> and <Activity> elements of `AndroidManifest.xml`, respectively. During the installation of an application, if *PackageManagerService* [2] finds "`android: SAMDApp=true`" in <Application>, it considers the application as a SAMD app and identifies remoteable Activities[3] by checking the `Remoteable` attributes in each <Activity> when writing the Activity list. *PackageManagerService* then sends this application information to SAMD's *PackagingService* for the actual Controller packaging.

As `classes.dex` (the Android specific executable file) is a set of Java class files, we port and install `baksmali` [32] to Android to extract human readable 'smali' files from `.dex` file. These smali files are then analyzed to identify different Activities and their necessary resources. Afterwards, ActivityMap (ControllerMap) and ResourceTable are constructed with such pieces of information. In the case that an Activity requires external libraries, they are also listed in ResourceTable along with the Activity's resources. These external libraries are then included in the RC for effective remote execution.

In Android, compiled resources are associated with and referenced by their IDs which are again translated into offsets (static addresses) to `resources.arsc`. As resources in a RC, which are portion of the entire application resources, are compiled separately in the remote device for code adaptation, resource IDs of RCs in the remote device may differ from those in the local device. To resolve this issue, we overload the APIs of resource access in Android for resources to be referenced also by name. For RCs, ID-based API calls are replaced with the name-based API calls in their respective smali files.

[2]*PackageManagerService* is an Android system service that coordinates application installation. We modified it for SAMD app and RC identification.

[3]Activity is an Android specific term for Controller.

B. Implementation of Lookahead Transfer

The lookahead transfer functionality is implemented through *AppMonitor* and *SelectionService*. The former monitors application state while the latter handles the actual transfer of Controllers. As a state change of the current Activity running on the local device (i.e., LC) may trigger remote execution of a RC, *AppMonitor* needs to constantly monitor application state. *AppMonitor* is essentially implemented within the *ActivityThread* daemon in Android. Upon a state change that requires RC transfer, *SelectionService* looks up ActivityMap and selects Activities for transfer. It then in conjunction with *PackagingService* and *CommunicationService* prepares and transfers RCs for selected Activities to remote devices. *CommunicationService* also provides *SAMDMessageListener* as an event handler for communication between Activities of local and remote devices.

C. Implementation of Code Adaptation

The code adaptation functionality is implemented through *DePackagingService* and *AdapationService*. The core functionality of code adaptation is in *AdaptationService* that deals with generating the executable file from the Controller code, forwarded from *DePackagingService*, and plugging it with its resources into the agent app. We have ported `baksmali` and `aapt` (Android Asset Packaging Tool) [33] to Android for converting smali files back to the `classes.dex` file and compiling resources into `resources.arsc`, respectively. *AdaptationService* stores (actually, overwrites) `classes.dex` and `resources.arsc` files in the agent app's installation directory. The agent app executes using them.

D. SAMD APIs

There are three types of API provided in our SAMD implementation on Android: (remote) device management API, Activity management API and communication API.

Device management API, such as *getConncectedDevices()* identifies nearby devices and handles connection/pooling for collaboration. In this work, we adopt techniques developed as part of our previous work in CollaboRoid [34].

Second, *launchActivity()* API is provided for sending a message to execute a specified Activity from a specified remote device. It includes typically two parameters: *device number* and *Activity name*. On receiving the message, if the RC for the specified Activity has already arrived to the remote device via lookahead transfer, the remote device plugs the specified Activity into the agent app and then launches the agent app. If the RC has not arrived yet, the remote

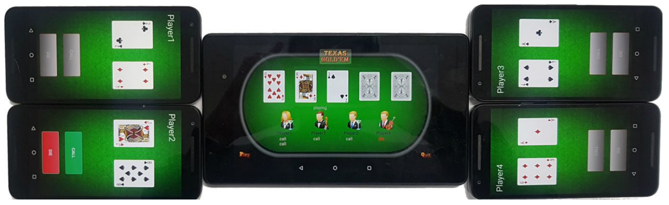


Fig. 6. TexasHoldem: App is installed and launched *only* in the center (local) device where BoardActivity is currently being executed. Four remote devices in both sides are executing PlayerActivity with different cards each other.

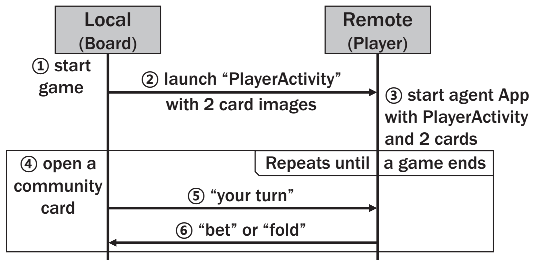

Local (Board)
Remote (Player)
① start game
② launch "PlayerActivity" with 2 card images
③ start agent App with PlayerActivity and 2 cards
④ open a community card
Repeats until a game ends
⑤ "your turn"
⑥ "bet" or "fold"

Fig. 7. TexasHoldem application processes.

device waits for the RC and executes the Activity after code adaptation. *launchActivity(DEVICE_NUM, ACTIVITY_NAME, BUNDLE)* API is also provided for additional resources and parameters to be dynamically sent. *BUNDLE* is a String type parameter. If it starts with 'rsc_', the specified resource is transferred to the specified device.

The third and last type of API is communication API, *sendMessageToRemote(DEVICE_NUM, SAMD_MESSAGE)* and *sendMessageToLocal(SAMD_MESSAGE)*, for sending messages between local and remote devices. *SAMD_MESSAGE* can have various types of object including string and file. Also, SAMD provides *MessageListener* for processing messages.

V. Example SAMD Application: *TexasHoldem*

In this section, we illustrate the development of the *TexasHoldem* (Figure 6) as an example SAMD application. The game plays by interacting between BoardActivity (LC) in the local device and PlayerActivity (RC) in remote devices. These Activities are analogous to the dealer and players of the real-life game. The sequence of *TexasHoldem* app is as shown in Figure 7. In the following, we show the core parts of Texas Holdem application code (Programs 1 and 2).

Program 1 shows the partial, core code of BoardActivity in the *TexasHoldem* app. The first step is to get *SAMDService* (service manager, line 7 in Program 1) that plays as a gateway to SAMD services. A Listener is then registered for message passing (line 8). These two steps are typically the very first two tasks any SAMD app has to do. As BoardActivity is playing as a coordinator (i.e., a dealer), it identifies participating devices (line 9); this is another typical step of the main Activity in the local device. A message is sent to each remote device to launch the PlayerActivity with private card images in the *Bundle* object (lines 12–19, ② in Figure 7). The Controller for PlayerActivity may have already arrived to the remote devices at the time since it has been sent, via lookahead transfer, as soon as the BoardActivity started. Since the private cards for each player are determined dynamically, it cannot be sent together with the PlayerActivity, a priori. The actual game starts by calling the *playNextTurn* method (line 21). This method opens community cards, records the move the current player has made and sends a message ("your turn") to the next player (lines 30–40, ④ and ⑤). A player who receives message "your turn" selects either 'bet' or 'fold' as her move. Then, a message with the selected move gets sent to the local device and the game continues on (lines 24–28).

Program 1. BoardActivity of TexasHoldem

```
 1 public class BoardActivity extends Activity {
 2     SAMDService SAMD;
 3     SAMDMessage msgToRemote;
 4     ...
 5     public void onCreate(Bundle savedInstanceState) {
 6         ...  // set layout and initialize SAMDMessage and components
 7         SAMD = (SAMDService) getSystemService(Context.SAMDService);
 8         SAMD.setListener(mSAMDListener);
 9         int [] devices = SAMD.getConnectedDevices();
10         ArrayList<Player> players = initPlayer(devices);
11         // launch PlayerActivity with Bundle
12         for ( int deviceNum : devices) {
13             int privateCard1 = deck.getCard();
14             int privateCard2 = deck.getCard();
15             Bundle b = new Bundle();
16             b.putString("rsc_card1", "Card_"+privateCard1);
17             b.putString("rsc_card2", "Card_"+privateCard2);
18             SAMD.launchActivity(deviceNum, "PlayerActivity", b);
19         }
20         ...
21         playNextTurn(0, null);
22     }
23     // handle SAMDMessage
24     private SAMDMessageListener mSAMDListener = new
               SAMDMessageListener() {
25         public void onSAMDMsg(int deviceNum, SAMDMessage msg) {
26             playNextTurn(deviceNum, msg.getString());
27         }
28     }
29     ...
30     public void playNextTurn(int deviceNum, String move) {
31         openCommunityCards();
32         msgToRemote.putString("your turn");
33         // start game
34         if (deviceNum == 0)
35             SAMD.sendMessageToRemote(players.getFirstPlayer(),
                       msgToRemote);
36         else {
37             players.getPlayer(deviceNum).setMove(move);
38             SAMD.sendMessageToRemote(players.getNextPlayer(deviceNum),
                       msgToRemote);
39         }
40     }
41 }
```

The first two steps of PlayerActivity implementation (Program 2) are the same as those of BoardActivity (lines 9 and 10). Two card images in the *Bundle* object received from the local device are displayed on the screen (lines 12–14). Up to line 14 is the process from the start of the game to the point the game shows players their cards. After that, when "your turn" message is received from the local device, bet and fold buttons get activated for the player to choose from (lines 32–37). If either button is clicked, PlayerActivity sends a message of the selected button to the local device and the buttons are deactivated until next "your turn" message arrives (lines 16–29, ⑥).

Program 2. PlayerActivity of TexasHoldem

```
public class PlayerActivity extends Activity {
    SAMDService SAMD;
    SAMDMessage msgToLocal;
    ImageView ivCard1, ivCard2;
    Button betBtn, foldBtn;
    ...
    public void onCreate(Bundle savedInstanceState) {
        ... // set layout and initialize SAMDMessage and components
        SAMD = (SAMDService) getSystemService(Context.SAMDService);
        SAMD.setListener(mSAMDListener);

        Bundle b = getIntent().getExtras();
        ivCard1.setImageResource(b.getString("rsc_card1"));
        ivCard2.setImageResource(b.getString("rsc_card2"));

        betBtn.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v){
                msgToLocal.putString("BET");
                SAMD.sendMessageToLocal( msgToLocal);
                endTurn();   // make bet and fold buttons unclickable
            }
        }
        foldBtn.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v){
                msgToLocal.putString("FOLD");
                SAMD.sendMessageToLocal(msgToLocal);
                endTurn();
            }
        }
    }
    // handle SAMDMessage
    private SAMDMessageListener mSAMDListener = new
          SAMDMessageListener() {
        public void onSAMDMsg(int deviceNum, SAMDMessage msg) {
            if ( msg.getString().equalsIgnoreCase("your turn"))
                playTurn(); // make bet and fold buttons clickable
        }
    }
    ...
}
```

VI. Discussion

In this section, we discuss three particular aspects of mobile collaboration in the context of SAMD.

Security. The security and privacy concern becomes far greater when multiple (possibly arbitrary) devices are performing some form of mobile 'collaboration' that SAMD is designed for. In essence, SAMD enables "arbitrary" code to dynamically run on multiple devices owned by different users. This may become a serious security concern if that arbitrary code is malicious. In SAMD, we address the security issue in three ways: connection authorization, Controller management and application permission inheritance. The users of devices willing to participate in a particular collaboration shall manually select the collaboration initiating device (i.e., local device) to get connected and authorized for the collaboration.

To make SAMD less intrusive, Controllers in remote devices may be removed at the time of termination of a collaboration or cached for later use if and only if remote device users allow.

In the Android platform, the privilege of an application (e.g., in accessing resources) is dictated by its permissions. However, in the case of the agent application in SAMD, it is difficult to anticipate what code will execute; and thus, setting particular permissions a priori is not quite possible. Such a permission concern may be addressed by adopting a permission inheritance technique as in [35] that enables agent app plugged with a received Activity to inherit permissions of the SAMD application in the local device.

Portability. SAMD can be implemented in any mobile platform if the platform features the following two characteristics: 1) decomposing and analysis of application executable file is possible and 2) replacing the executable file and resources of the application is possible. There are a number of tools, that can be ported to mobile platforms, for decomposing and analyzing executable files. The main challenge is with the second characteristic as mobile platforms differ in structure. Typically, at the time of application installation, mobile platform creates a sub-directory, in its installation directory, to store all necessary data for execution. For instance, Android creates the application's directory in /data/app/ or /system/app/ and then stores APK. Similarly, iOS creates a directory in /Apps/ and stores Documents, Library, and application binary. Due to these similarities in application runtime environments across mobile platforms, the challenge in the third application repackaging characteristic can be overcome.

Error handling. 'Mobile' devices are transient in nature. They may disappear (without a notice) during collaboration due to, for example, out of batteries or network instability. SAMD handles these errors by regularly checking heartbeats of mobile devices participating in the collaboration. This is for simply notifying the local device (more precisely, the app in the local device) and excluding disconnected devices from the list of participating devices. The actual error handling is often specific to an app. For a multi-player board game (e.g., TexasHoldem), the disconnection of any participating device makes the entire game invalid. In the meantime, for a distributed processing app that delegates tasks of a particular job to multiple devices, the app may need to have some form of fault tolerance feature specific to the nature of job the app performs, e.g., re-sending failed tasks to other participating devices and restarting them there or check-pointing tasks and resuming failed tasks in other devices.

VII. Evaluation

We have evaluated the performance of SAMD in terms of feasibility and capacity. The evaluation has been conducted with five applications in SAMD-enabled Android phones of three models, Nexus 6P, Nexus 5X and Nexus 5. The five experimental SAMD apps and their details are shown in Table I. In particular, *TexasHoldem* represents typical multi-user apps. *Code-Intensive* and *Resource-Intensive* represent apps with two extreme characteristics, 1) long remoteable code with small amount of resources and 2) short remoteable code with large amount of resources, respectively. *TestApp(n/m)* represents apps with multiple LCs and/or RCs. It is a transformable app that consists of n LCs and m RCs. *Mosaicing* is a distributed image processing app.

All mobile devices used in our experiments are located in the same room and connected to each other via a dedicated WiFi AP that supports 802.11 a/b/g/n/ac. We repeated the same experiments five times and used their average to evaluate the performance of SAMD.

TABLE I
CHARACTERISTICS OF EXPERIMENTAL SAMD APPS.

Application name	Description	Controller size (code/resource)	Number of LCs/RCs	APK size
TexasHoldem	Texas Holdem board game app	62 KB (8 KB/54 KB)	1/1	1.5 MB
Code-Intensive	Includes a RC with $\geq$2,000 lines of code	72 KB (68 KB/4 KB)	1/1	60 KB
Resource-Intensive	Includes a RC which uses large sized resources	20 MB (3 KB/19.8 MB)	1/1	20 MB
TestApp(n/m)	Includes n and m LCs and RCs, respectively	variable (c MB)	n/m	$(n+m) \times c$ MB
Mosaicing	Distributed image mosaicing	13 KB (9 KB/4 KB)	1/1	39 KB

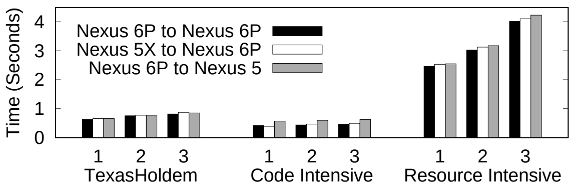


Fig. 8. Remote execution latency: x-axis indicates #remote devices.

A. Remote Execution Latency

We first evaluate the remote execution latency which is the time difference between the launch time of an app in the local device and the launch time of a transferred RC in the remote device. Up to three remote devices participate in this experiment. We used three combinations of local and remote devices to represent the device heterogeneity in the real world: (1) Nexus 6P (Local) and Nexus 6P (Remote) to represent the same devices on both sides, (2) Nexus 5X (Local) and Nexus 6P (Remote) to represent different devices with similar performance, (3) Nexus 6P (Local) and Nexus 5 (Remote) to represent different devices with different performance.

Figure 8 shows the results. Since the remote execution latency in each remote device can be different from each other, we chose the largest (i.e., worst) value among them. *TexasHoldem* and *Code-Intensive* apps show less than one second of latencies in all cases, while *Resource-Intensive* app shows up to about four seconds. The large latency of the *Resource-Intensive* app is due to the huge sized (about 20 MB) resources that is transferred to all remote devices. Hence, in the current network environment, latency-sensitive applications having large sized resources are not adequate to be SAMD apps. However, since SAMD shows reasonable latencies in *TexasHoldem* and *Code-Intensive* apps, latency-insensitive applications such as board games and instant messaging apps can be good candidates for SAMD apps.

While the Controller sizes of *TexasHoldem* and *Code-Intensive* apps are similar to each other, their latencies are noticeably different due to the interactive resource transfer latency in *TexasHoldem*. Dynamically determined resources (i.e., private card images for each user) can not be transferred in advance via lookahead transfer. They are transferred at the time of the agent app launch, which increases the overall launching time of the agent app. We can confirm it from Figure 9, which shows the breakdown of the entire remote execution latency.

We can see that the major source of remote execution latency in each app differs from each other. The interactive resource transfer latency makes the Launching phase the main source of latency in *TexasHoldem*. The Delivery phase

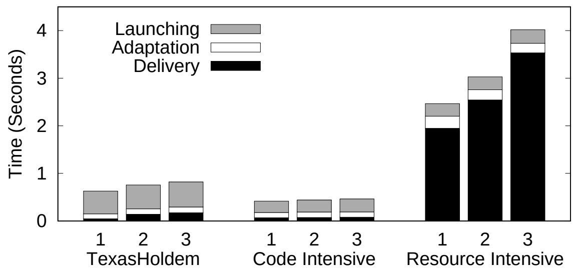


Fig. 9. Remote execution latency breakdown (Nexus 6P to Nexus 6P): Delivery, Adaptation and Launching represent Controller transfer, code adaptation and agent app launching phases, respectively.

dominantly contributes to the latency in *Resource-Intensive* app due to its large sized resources.

The number of remote devices affect the latency in the Delivery phase since it determines the total amount of data to be transferred via network. However, it does not affect latencies in other phases. In Launching phase, only small-sized messages are exchanged between local and remote devices if interactive resource transfer does not occur, hence its latency is not largely affected by the number of devices. Other latency included in the Launching phase is the agent app launching time, which takes about 0.2 seconds in Nexus 6P. The Adaptation phase is performed in the individual remote devices, which shows constant latency regardless of the number of remote devices. Controller size is the dominant factor determining the latency in Adaptation phase, which makes *Resource-Intensive* app show the largest latency in that phase.

Through the lookahead transfer in SAMD, RCs are transferred and adapted before they are actually called by a LC. Hence, if there is sufficient interval from the beginning of a LC to the call of a RC, the only latency users actually experience is that of the Launching phase, which makes SAMD feasible even for large-sized applications having large amount of resources.

To evaluate the effect of the amount of data, transferred to each remote device, on the feasibility of SAMD, we measured the remote execution latency varying the Controller size of *TestApp(1,1)*. Figure 10 shows the results in the Nexus 6P to Nexus 6P case. Due to the increasing latency of the Delivery

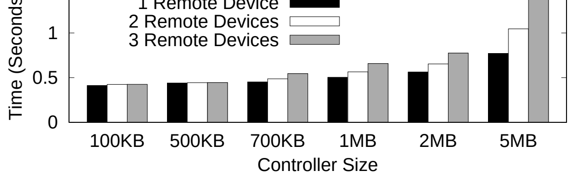


Fig. 10. Remote execution latency with varying Controller size.

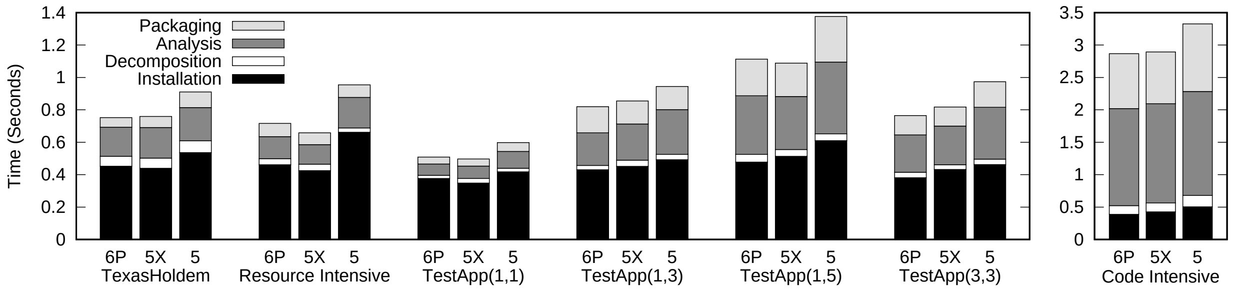


Fig. 11. SAMD app installation time breakdown: Controller size of *TestApp* is fixed to 2 MB (code: 5 KB, resource 2 MB).

phase, the remote execution latency increases with the increasing Controller size. Therefore, SAMD app developers need to consider not only the latency-sensitivity of the application but also the sizes of RCs and the number of simultaneous users.

B. Installation time for SAMD apps

As the installation of a SAMD app requires additional steps (decomposition, analysis and packaging) for Controller packaging, we measured the installation time of SAMD apps. Figure 11 shows the results. The 'Installation' is the time for installing APK file to the device. After installing the APK file, the Controller packaging is performed to make the app 'SAMD-ready'. Since the time for decomposition step depends on the code size of the app, *Code-Intensive* and *TestApp(1,1)* show the largest and least decomposition times, respectively. The time for analysis step also depends on the code size because it not only reads smali files but also modifies them for ID-based API call replacement as described in Section IV-A. One more notable fact is that, since the modification of smali files incurs costly write operations to the storage device, the analysis step takes longer time as the number of RCs increases. The time for packaging step depends on both the number and code size of RCs. While *Code-Intensive* has only one RC, it shows large packaging time due to the large code size of the RC. *TestApp(1,5)* also shows large packaging time due to the large number of RCs.

C. Effectiveness of the Lookahead Transfer

SAMD uses lookahead transfer to send RCs, a priori, to remote devices in order to reduce the remote execution latencies of RCs. To evaluate the effectiveness of the lookahead transfer in SAMD, we measured the remote execution latency of each RC. The latency is defined as the time between the next RC is determined in the local device and the RC is actually launched in the remote device. If we do not use lookahead transfer, a RC is transferred to the remote device only after it is determined as the next Controller to be executed in the remote device, which may cause a large latency.

To mimic the user interaction with the app, we modified *TestApp(1,3)* so that a remote device execute multiple RCs one by one. Every RC is programmed to send a message to the LC one second after its execution. On receiving the message, the LC is programmed to determine the next RC to be remotely executed among unexecuted RCs. In this way, all three RCs are executed once in some order, and we measure their respective remote execution latencies.

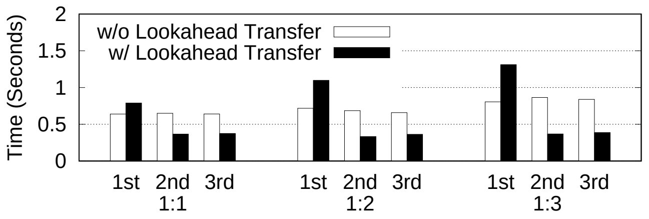


Fig. 12. Remote execution latency of the *TestApp(1,3)* w/ or w/o lookahead transfer: n:m in the x-axis represents the number of local:remote devices. Controller size: 2 MB.

Figure 12 shows the results. When using lookahead transfer, SAMD starts to transfer three RCs as soon as the LC is executed. The first RC determined by the LC is executed when it becomes ready in the remote device, i.e., delivered and adapted. Using the lookahead transfer, the remote execution latency of the first RC can be larger than that without the scheme, since the RC can be the second or third one in the transfer order from the local device and so has not been delivered or adapted yet. However, for the second and third RCs, their latency are very likely to be smaller than those without the scheme, since they have high probability of being already ready to be executed in the remote device.

It is notable that the latency of the first RC when using lookahead transfer can be similar to those of the second and third RCs when the LC issues execution of the first RC after sufficient time elapsed from the start of the LC. Hence, if the time interval between user interactions is not extremely small, the remote execution latency can be reasonably small. The experimental results in Figure 12 confirms it.

When lookahead transfer is not used, the latency is not affected by the execution order. However, it increases as the number of remote devices increases due to the increasing Controller delivery time.

D. APK Installation scheme versus SAMD

The easiest way to execute a local app on the remote device is to transfer the whole package file (i.e., APK file in Android) of the app and then install and launch the app in the remote device. We call this simple approach as 'APK installation scheme' in this paper. In this section, we compare the remote execution latencies between the APK installation scheme and SAMD to show the relative performance of SAMD. We

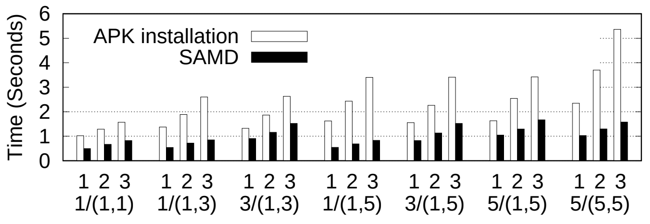


Fig. 13. Remote execution delay of *TestApp(n,m)* between APK installation scheme and SAMD: 1, 2 and 3 in the x-axis represent the number of remote devices, and $k/(n, m)$ represents the case where there are k *current* RCs in current state of the *TestApp(n,m)*. Controller size: 2 MB.

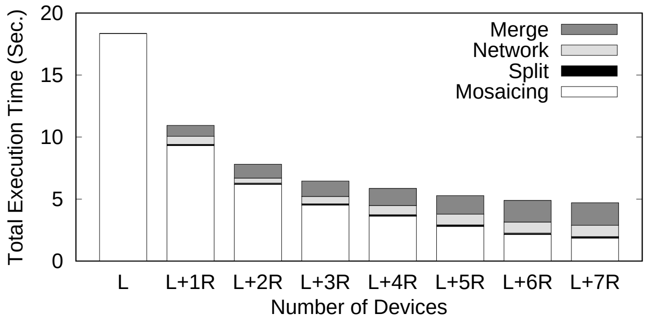


Fig. 14. Total execution time breakdown: Split, Network, Mosaicing and Merge represent image splitting, transfer, processing and merging time, respectively. L and R in x-axis denote local and remote devices, respectively.

implemented dynamic application package extraction, transfer and installation functionalities to the Android platform. Using these functionalities, when a local app is launched, its APK file is extracted and transferred to remote devices and then installed (as a foreground job) and launched in those devices.

Figure 13 compares the remote execution latencies between APK installation scheme and SAMD. Since the APK installation scheme transfers and installs the whole app to remote devices, its remote execution latency is inevitably larger than that of SAMD, as shown in the figure. In this experiment, we varied not only the total number of RCs but also the number of '*current*' RCs. The *current* RCs are remoteable Controllers that are called by the *currently executing* LC for remote execution. Only *current* RCs are transferred to the remote devices, a priori, by the lookahead transfer in SAMD. The RC that will be executed in the remote device is determined, among the *current* RCs, by the user action or the current context of the app. Hence, as the number of *current* RCs increases, the average remote execution latency becomes larger since the lastly transferred RC can be determined to be executed. In the experiment, we randomly selected the RC to be executed. All values in Figure 13 are the average of measured times from 10 repeated experiments.

In APK installation scheme, the latency depends only on the number of total Controllers in the app, since the number of Controllers determines the size of the app to be transferred and installed before launch. On the other hand, the latency of SAMD depends on the number of *current* RCs not on the number of total Controllers.

E. Distributed processing

SAMD can also be effectively used for distributed processing of computation-intensive applications. To evaluate the feasibility of distributed processing using SAMD, we have implemented an image mosaicing app, *Mosaicing*. *Mosaicing* consists of one LC and one RC. The LC equally divides an input image into multiple subimages of which the number is equal to the number of participating devices, and distributes them along with RC to all devices. The RC, after mosaicing subimage, returns the result (mosaic subimage) to the LC. Then the LC merges mosaic subimages to produce the final mosaic image. The RC is executed in both local and remote devices.

Figure 14 shows the breakdown of the total execution time for *Mosaicing* to process an image file. We repeated experiment three times using three different image files with the same resolution (5,500×4,000), and then averaged their execution times. We used Nexus 6P for local device and three Nexus 6P and four Nexus 5X devices for remote devices. From L+1R to L+3R, only Nexus 6P were used for remote devices. When no remote device is used (L case), the standalone version of the *Mosaicing* is used to exclude any overhead induced by distributed processing.

Figure 14 shows that SAMD can be effectively used for distributed processing of computation-intensive applications among mobile devices. The increasing overhead along the number of devices is largely due to the design nature of the *Mosaic* rather than due to the native overhead of SAMD. For example, since the LC performs merge operation as many times as the number of subimages, the merge overhead linearly increases to the number of devices.

VIII. Conclusion

In this paper, we present Single Application Multiple Device (SAMD) as a novel mobile collaboration framework. We have demonstrated fine-grained application sharing enabled by the platform-level support of SAMD significantly facilities the development of mobile collaboration applications. In particular, platform-level solutions of SAMD to the three main challenges in mobile collaboration liberate application developers from custom-implementation of collaboration functionalities. Our thorough evaluation study and experimental results confirm our claims.

For our future work, we plan to study resource scheduling for mobile collaboration, distributed processing applications in particular, taking into account resource capacity and status of remote devices. Error handling and fault tolerance are also part of our future work.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A2A01004187). Sooyong Kang is the corresponding author of this paper.

References

[1] Supercell, "Clash Royale." [Online]. Available: https://play.google.com/store/apps/details?id=com.supercell.clashroyale

[2] Google Inc., "Google docs." [Online]. Available: https://docs.google.com/

[3] LiveQoS, "SuperBeam — WiFi Direct Share," 2015. [Online]. Available: https://play.google.com/store/apps/details?id=com.majedev.superbeam

[4] Estmob Inc., "Send Anywhere," 2017. [Online]. Available: https://play.google.com/store/apps/details?id=com.estmob.android.sendanywhere

[5] Google Inc., "Google Instant Apps." [Online]. Available: http://developer.android.com/topic/instant-apps/index.html

[6] R. A. Baratto, L. N. Kim, and J. Nieh, "THINC: A Virtual Display Architecture for Thin-Client Computing," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 277–290.

[7] R. A. Baratto, S. Potter, G. Su, and J. Nieh, "Mobidesk: Mobile Virtual Desktop Computing," in *Proceedings of the 10th ACM Annual International Conference on Mobile Computing and Networking*, 2004, pp. 1–15.

[8] J. Kim, R. A. Baratto, and J. Nieh, "pTHINC: A Thin-Client Architecture for Mobile Wireless Web," in *Proceedings of the 15th ACM International Conference on World Wide Web*, 2006, pp. 143–152.

[9] K.-W. Lim, J. Ha, P. Bae, J. Ko, and Y.-B. Ko, "Adaptive Frame Skipping with Screen Dynamics for Mobile Screen Sharing Applications," *IEEE Systems Journal*, 2017.

[10] S. Clinch, J. Harkes, A. Friday, N. Davies, and M. Satyanarayanan, "How Close is Close Enough? Understanding the Role of Cloudlets in Supporting Display Appropriation by Mobile Users," in *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2012, pp. 122–127.

[11] Apple, "AirPlay," 2015. [Online]. Available: https://developer.apple.com/airplay

[12] A. Lucero, J. Holopainen, and T. Jokela, "Pass-Them-Around: Collaborative Use of Mobile Phones for Photo Sharing," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2011, pp. 1787–1796.

[13] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A System Solution for Sharing I/O Between Mobile Systems," in *Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys '14)*, 2014.

[14] B. C. Tak and C. Tang, "Appcloak: Rapid Migration of Legacy Applications into Cloud," in *7th IEEE International Conference on Cloud Computing (CLOUD)*, 2014, pp. 810–817.

[15] V. Andrikopoulos, A. Darsow, D. Karastoyanova, and F. Leymann, "CloudDSF–The Cloud Decision Support Framework for Application Migration," in *European Conference on Service-Oriented and Cloud Computing*, 2014, pp. 1–16.

[16] Z. Cai, L. Zhao, X. Wang, X. Yang, J. Qin, and K. Yin, "A Pattern-Based Code Transformation Approach for Cloud Application Migration," in *8th IEEE International Conference on Cloud Computing (CLOUD)*, 2015, pp. 33–40.

[17] J. Ejarque, A. Micsik, and R. M. Badia, "Towards Automatic Application Migration to Clouds," in *8th IEEE International Conference on Cloud Computing (CLOUD)*, 2015, pp. 25–32.

[18] Y. Hu, T. Azim, and I. Neamtiu, "Versatile Yet Lightweight Record-and-Replay for Android," in *ACM SIGPLAN Notices*, vol. 50, no. 10, 2015, pp. 349–366.

[19] 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 (Eurosys)*, 2015.

[20] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (Mobisys)*, 2010.

[21] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic Execution Between Mobile Device and Cloud," in *Proceedings of the 6th European Conference on Computer Systems (Eurosys)*, 2011.

[22] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings of the 31st IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

[23] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Adaptive Deployment and Configuration for Mobile Augmented Reality in the Cloudlet," *Journal of Network and Computer Applications*, vol. 41, pp. 206–216, 2014.

[24] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge," in *8th IEEE International Conference on Cloud Computing (CLOUD)*, 2015, pp. 9–16.

[25] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code Offload by Migrating Execution Transparently," in *OSDI*, vol. 12, 2012, pp. 93–106.

[26] Y. Li and W. Gao, "Minimizing Context Migration in Mobile Code Offload," *IEEE Transactions on Mobile Computing*, vol. 16, no. 4, pp. 1005–1018, 2017.

[27] S. Yang, Y. Kwon, Y. Cho, H. Yi, D. Kwon, J. Youn, and Y. Paek, "Fast Dynamic Execution Offloading for Efficient Mobile Cloud Computing," in *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2013, pp. 20–28.

[28] C. Dibak, A. Schmidt, F. Dürr, B. Haasdonk, and K. Rothermel, "Server-assisted interactive mobile simulations for pervasive applications," in *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2017, pp. 111–120.

[29] E. E. Marinelli, *Hyrax: Cloud Computing on Mobile Devices using MapReduce*. Masters Thesis, Carnegie Mellon University, 2009.

[30] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling Remote Computing Among Intermittently Connected Mobile Devices," in *Proceedings of the 13th ACM international symposium on Mobile Ad Hoc Networking and Computing*, 2012, pp. 145–154.

[31] D. Díaz-Sánchez, A. M. López, F. Almenares, R. Sánchez, and P. Arias, "Flexible Computing for Personal Electronic Devices," in *IEEE International Conference on Consumer Electronics*, 2013, pp. 212–213.

[32] JesusFreke, "Baskmali tool." [Online]. Available: http://baksmali.com

[33] G. Inc., "Android Asset Packaging Tool." [Online]. Available: http://elinux.org/Android_aapt

[34] J. Lee, H. Lee, Y. C. Lee, H. Han, and S. Kang, "Platform Support For Mobile Edge Computing," in *10th IEEE International Conference on Cloud Computing (CLOUD)*, 2017.

[35] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "FLEXDROID: Enforcing In-App Privilege Separation in Android," in *NDSS*, 2016.