NEOP: A Framework for Distributed Mobile Apps on Heterogeneous Devices

Yiwei Zhao
OPPO Research USA
Palo Alto, CA, USA
yiwei@oppo.com

Song Jiang
University of Texas at Arlington
Arlington TX, USA
song.jiang@uta.edu

Weidong Zhong
OPPO Neutron Lab
Chengdu, China
zhongweidon@oppo.com

Lizhong Wang
OPPO Neutron Lab
Chengdu, China
wlz@oppo.com

Xiao-Feng Li
OPPO Research USA
Palo Alto, CA, USA
xiao-feng.li@oppo.com

***Abstract*—Today's apps on a mobile device, such as a smartphone and a tablet, need to access various resources to deliver quality service to users' satisfaction. These resources may include cameras, microphones, screens, processors, various specialized sensors, and data. In today's client-server framework, resources accessible to an app are limited to those available in the device running the app, on the cloud, and likely in a few statically connected devices. However, there can be abundant resources on devices near the app-running one with desirable functionalities that can enable or empower the app's new features and services, but cannot be easily accessed and leveraged.**

The NEOP (Neutron Operation Platform) framework is an app development and execution environment that removes the barrier across the devices. Heterogeneous IoT devices make the capabilities in their hardware and service software available after security and privacy authentication. An app is developed as a composition of capabilities distributed across various end devices and the cloud. Its constituent computing tasks can be dynamically created and scheduled. Different device capabilities can be selectively and dynamically recruited into the app for the optimal user experience.

In this paper we describe example scenarios that motivate the next-generation app framework, the framework's architecture, design principles, technical challenges, and details on its design and implementation. We also compare this work with related efforts on distributed mobile computing to highlight the unique contributions made by the NEOP platform.

***Index Terms*—Android ecosystem, app development**

I. INTRODUCTION

Nowadays, people's lives have been tightly connected to smart mobile devices, such as smart phones and tablets. It would be a huge challenge, if not unlikely, for a person to live conveniently and productively without accesses to apps on the devices. This is especially the case when an app leverages functionalities available on increasingly diverse devices, such as smart watches, smart locks, and smart speakers. Traditionally, an app is designed mostly with capabilities on the hosting device (e.g., display, camera, speaker, microphone, fingerprint reader, and gyroscope sensor on a smartphone) and those in the cloud (e.g., processor and storage on servers at a data center) in mind. When more and more external smart IoT devices with various appealing capabilities (e.g., smart TVs, smart locks, smart speakers, gaming headsets, and surveillance cameras) become available and pervasive in our surroundings, app developers and users alike welcome the opportunity to extend the app's reach beyond its host device.

While people are well aware of the demand on such an extension, and efforts have been continuously made to meet the demand, current solutions are inadequate for a number of reasons. First, connections between devices are manually established and statically maintained. Examples include Bluetooth-connected speakers, Miracast-connected TVs [1], and cloud-assisted connection to surveillance cameras. While these individual technologies can make external devices accessible, they are point solutions that apps have to be customized to at their design phase. They allow connections to only limited compatible devices by design, and do not support access of devices with dynamic selection and connection for optimal user experience. Second, an app (or an Android system service) comprises a number functionalities. The binder IPC mechanism is employed to enable sharing of functionalities within a device [2]. Though cross-device sharing is possible for reaching functionalities on other devices, there is not any infrastructure-level support that can systematically facilitate advertisement, search, and recruitment of functionalities across the end devices. Because there exist rich, diverse, and sometimes unique functionalities across the devices that might be on the move, it is a challenge to dynamically compose a service or an app with distributed functionalities as building blocks. Third, a critical resource for sharing among devices is the data generated on-site at the IoT devices. Current practice usually allows sharing of the data within one device and between client-side and server-resident components. Functionalities on the devices in a collaborative environment often assume accessibility of data in the peer devices. Due to limited upload/download bandwidths and data-privacy concern, it is desired to have an cross-device self-managed data storage and sharing system to support running of distributed apps on the devices.

Instead of adopting point solutions addressing issues raised in individual apps or specific use scenarios, NEOP (Neutron Operation Platform), which was proposed and is being developed by OPPO, provides an app development and execution platform that fundamentally removes the barriers between functionalities on different devices. A NEOP app is a composable entity consisting of functionalities distributed over multiple devices from its creation, instead of being an extension of a one-device-hosted app to reach manually designated external devices. This is made possible in NEOP by

a number of its newly introduced key technologies, including remotization device capabilities[1], intelligent interconnection scheme, across-device distributed file and database systems, and dynamic task scheduling. NEOP is developed as an enhancement of the Android ecosystem and is backward compatible with Android apps.

There are many use scenarios in our everyday lives where a natively distributed app can provide very different and appealing use experiences. (1) A user opens an email app on his smartphone and then two PDF files attached to an email. The phone's screen appears to be too small when the user needs to simultaneously read and compare contents in the documents and accordingly compose a reply email. Meanwhile, there are a number of devices nearby with large displays, such as a smart TV, a laptop computer, and a tablet. As the external displays have been detected and connected with the host smartphone, the user may choose to move one document viewer to the TV display and another to the PC screen, while dedicating his phone screen to editing reply email. For an app in the distributed environment, the displays, regardless whether they are local or remote, are readily available for the app to selectively show its contents. Similarly, with this framework support it is straightforward to construct a multi-head display for a big screen experience. This experience is in sharp contrast to simply projecting an entire phone screen to an external display. (2) A user has multiple devices, each for one of his required functionalities, such as multiple surveillance cameras for taking photos, and multiple microphones and speakers on the smarter speakers for receiving voice commands and playing sound. Instead of relying on specialized gaming equipment and customer lock-in, his sports or immersive video game apps can coordinate the devices to work cooperatively and achieve a synergistic effect, such as the 3D sound and 3D image effect, taking 360-degree photos and videos, monitoring players' heart rates and gestures from their smart watches, adjusting smart lights for special effects, and leveraging computing power distributed in desktop and laptop computers. Composition of a distributed game app, instead of writing an traditional game app, provides an app developer with almost unbounded access of ever-growing capabilities of stationary and mobile devices for ever-richer play experience. (3) Due to his privacy concern, a user would like to leave his data produced by his in-house devices, such as video cameras, sensors, personal medical devices, and smart home appliances, at home. Some of the data may have been transferred into a home NAS and some still stay in the data-collection devices. The user may use an app to study the correlation of indoor/outdoor temperatures and humidity to his heart rate and blood pressure by synthesizing the data distributed in various devices. Without contacting individual devices and searching for its required data, the app relies on a distributed database, which is part of the NEOP framework's data service, to conveniently obtain the data. This data service will be available regardless of availability of cloud-side service.

[1]The remotization refers to techniques making a device's capabilities or services available to a remote computing device with standardized interfaces.

II. The Design

NEOP aims to provide a development and execution environment for natively distributed apps across end devices. On the one hand, it represents a conceptually disruptive technology to optimize development and user experience of mobile apps. On the other hand, it intends to minimize disruptions to both developers and users for them to easily adapt to this ecosystem.

A. Design Objectives and Principles

There are a number of objectives and principles in the design of the NEOP platform. First, a distributed app is composed of (likely dynamically) recruited services from various end devices and the cloud. Unlike motivation of traditional distributed computing for higher performance via resource sharing, NEOP's distributed apps are to optimize user experience via dynamic selection of services distributed in devices around a user. That is, a NEOP's design objective is to enable people-centered app development ecosystem where all resources should be leveraged for best user experience, including the scenarios where the people are on the move and their surrounding or available resources are changing. Second, the NEOP platform should provide a service market where a device can publish its capabilities as services for any other devices to search for and request on demand. A user's app can be aware of nearby available services, and coordinate their uses with other possible competing apps with the help of the platform. Third, as Android is the dominant open source system, the NEOP platform should be compatible to Android in its app development environment. While Android has extensive mechanisms and protocols for across-process/app communications (e.g., Binder [2]), requesting services (e.g., Intent [3]), sharing data (e.g., Content Provider [4]), and synchronizing events/actions (e.g., Broadcast Receiver [5]), NEOP should retain their API and seamlessly extend them to the cross-device environment with little disruptions to existing Android developers.

B. The Challenges

As a project to revolutionize today's mobile app ecosystem with a platform supporting development and use of naturally distributed apps, NEOP faces a number of challenges in its design and implementation. Being a project directly impacting its potential end users and developers, its ultimate challenge is on seamless and effortless adoption and acceptance into today's dominant mobile environments.

Specifically, most of the challenges are from minimization of disruption to the open-source Android system. (1) Device remotization into standardized services accessible to other remote devices. Devices are highly heterogeneous. They provide a plethora of functions on the hardware from different vendors using non-standard interfaces. An extensive and comprehensive suite of services and their interfaces need to be decided so that apps on a host device can access them

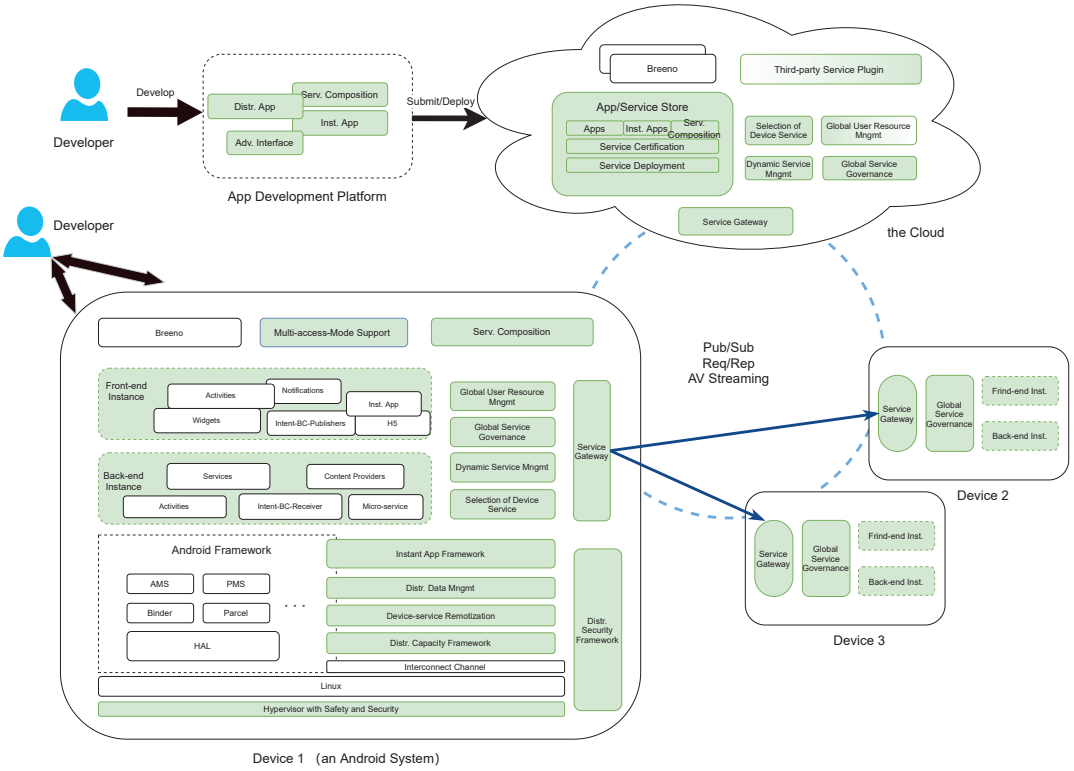

Fig. 1. Architecture of the NEOP app development and execution platform.

locally and remotely in an indistinguishable manner. They also allow hardware vendors to support them without major efforts to be included in the new ecosystem. (2) Self-managed end-device peer-to-peer (P2P) networking for high resilience and robustness. NEOP doesn't require any centralized server(s) to establish its multi-end networked platform, either remote server(s) in the cloud or local one(s) for coordinating operations in the platform. This is especially important when a user moves from one setting (e.g., his home) to another one (e.g., his office or a campsite) where (high-bandwidth) Internet or local servers are not available. When the platform can be P2P connected and its components, such as service publication, location, subscription, and resource allocation, are distributed across the end devices, it is a challenge to manage the service with self coordination in a dynamic and sometimes volatile environment, on aspects including network protocols and data formats, resolution of service/resource conflicts, and selection of the best device for a service request if there are multiple eligible ones. In the meantime, the capability of local decentralized self-management is not supposed to exclude the option of centralized app market and selected service management functionalities in the cloud to facilitate versatility of the ecosystem. (3) Flexible implementation strategy for the market penetration. There are two options to introduce the platform into the mobile system market: either use of a new SDK extended with NEOP capacities in apps or enhancement of the Android framework with NEOP components. For the first option an app needs to explicitly call functions for distributed execution capability enabled by NEOP. This means modification of existing apps to upgrade them to become NEOP-compliant. For the second option, all the new capabilities are added into the Android platform. That is, Android is upgraded to the NEOP platform. An app can run on and benefit from the NEOP platform with little modifications. While this option is more desirable for app developers and will be pursued as a complete solution, an extensive change of the official Android to immediately reach billions of its users is required but less practical. It is a challenge to design a technical path incrementally moving from the first option to the second one for the market acceptance. (4) Intelligent service selection. When increasingly more IoT devices are available for one specific capability, a selection out of them in response to a user's request has to rely on intelligence obtained by profiling user's history behaviors and preferences and a reliable learning model. (5) Strong security and privacy protection. As a NEOP app may request service from many external devices and is therefore exposed to possibilities of being attacked and compromised, it is critical to strengthen the Android's policy and mechanisms on authorization, authentication, and security/privacy protection in a distributed environment.

C. A Bird's-eye View of NEOP's Architecture

NEOP is a platform supporting development and execution of traditional mobile apps and next-generation distributed apps. Its architecture is illustrated in Figure 1. In the app development platform shown in the left upper of the figure, with the support of NEOP SDKs (e.g., service governance SDK,

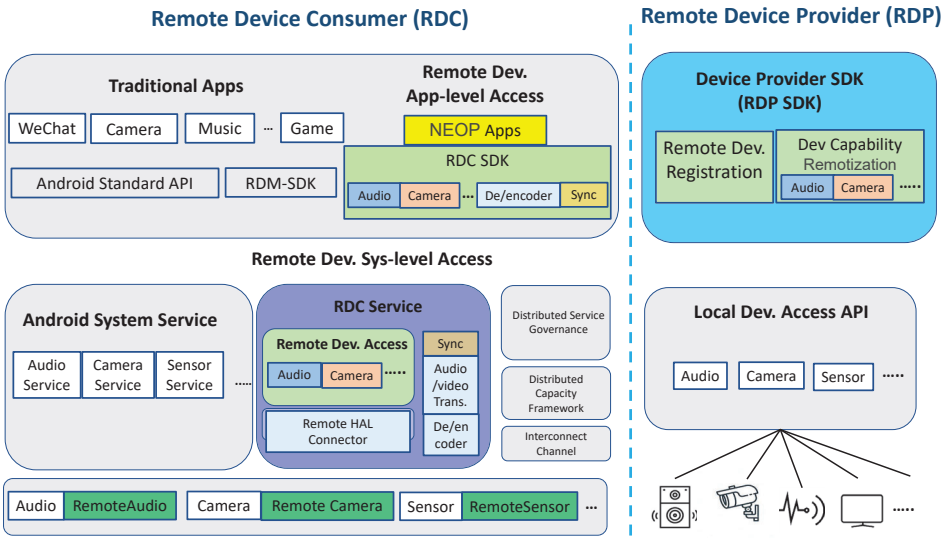

Fig. 2. The architecture of a remoted device and its access

capacity SDK, and data SDK) new apps can be developed as natural-born distributed ones with advanced user interfaces supporting voice and gesture inputs. In the meantime, Apps developed in the Android platform can run in the NEOP execution platform to have access of capabilities in other devices. Without modifying the Android apps the NEOP platform can redirect service requests to remote devices if the requested capabilities are not locally available. An app can also be composed with services that have been registered in the NEOP execution platform. Such an app itself can be registered in NEOP as a service for another app to call or use in its composition. In addition, NEOP supports instant apps. An instant app consists of multiple slices, Yaml configuration, and resource directory. Each of the slices can be independently and on-demand deployed on different devices. In a more dynamic environment where a user requests a service for only a few times (e.g., a traveler uses an airport app running on a distributed platform including his phone and airport kiosk.), such installation-free, light-weight, and customized instant apps are appealing and well supported in the NEOP platform.

The NEOP platform is distributed over the cloud and participants' end devices. The app/service store is in the cloud, where apps are registered for sharing. In addition, the store conducts service certification and deployment. The cloud server and each of the end devices have their respective gateway modules, which serve as connectors to make remote services available at local devices. Furthermore, they also have a set of system service modules (Selection of Device Service, Global User Resource Management, Dynamic Service Management, and Distributed Service Governance) that allow services at individual devices to be published, discovered, selected, and accessed. On each of the end devices, there is a front-end instance for interacting with users via either traditional Android's APIs, such as Activities, Notifications, and Widget, or human-language-based instructions, such as Google's Built-in Intents [6]. The back-end instance provides services running in the background. As an extension of Android's framework, NEOP's supporting infrastructure includes instant app framework, distributed data management, device-service remotization, distributed capacity framework, and distributed security framework. These modules together provide the upper-level service modules in the end devices with data access, remote service access, and security supports.

D. Device Remotization

A critical foundation of NEOP's distributed platform is remotization of devices and making remote devices accessible in a way similar to accessing local devices. To this end, NEOP provides modules at the remote device side to enable Remote Device Provider (RDP), and at the local app side to make an app accessing the remote device be the Remote Device Consumer (RDC). RDP and RDC are depicted in Figure 2, which shows the architecture of device remotization. NEOP has APIs specified in its SDK for each type of remoted devices. Device vendors use the SDK to remotize their devices and register their implemented services to the Distributed Service Governance module at each RDC, so that they can be discovered and reached.

There is a remote device access (RDA) module at the app level or/and another one at the system level in a RDC. To minimize changes in the apps or the difference between codes for accessing local and remote devices, NEOP prefers to place RDA in the system level. As shown in Figure 2, a regular Android app accesses the Android System Service as usual for a service provided by a local component, such as audio, camera, or sensor, which then connects to the corresponding remote HAL, such as remote audio HAL, in the Android HAL module, if a remote device is selected. This remote HAL is then connected to the RDA via the RDA's remote HAL Connector module. The RDA relies on the Distributed Service Governance module to find and establish its connection with the RDP and to receive its service over the network. The RDA

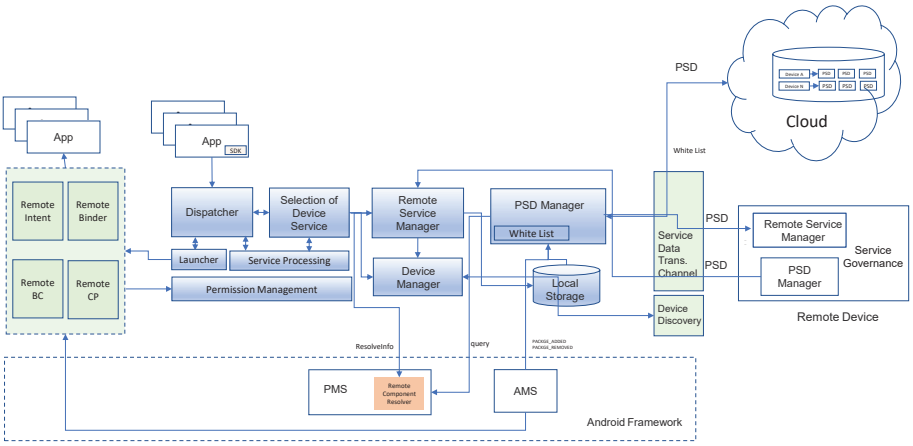

Fig. 3. The architecture of distributed service governance module

includes capabilities of accessing various types of RDP, such as audio and camera. It also provides basic functionalities, such as AV encoding/decoding and synchronization. If a remote RDP is not automatically selected by an intelligent strategy, it is manually selected by the user. In this case, the app can be upgraded with the RDM (Remote Device Management) SDK to provide an interface for the user to make the selection. Otherwise, the user can make his selection via a UI of RDC for the entire system.

NEOP also provides a RDC SDK to allow the app-level access of remote devices. In this case, an app can benefit from NEOP's distributed execution platform without having to upgrade his Android OS to a NEOP-compatible one. The app needs to explicitly access functionalities available in the SDK, such as the AV encoding/decoding, transmission, and connection to the remote RDP. The option of app-level remote device access is important for NEOP to quickly reach app developers and users and to motivate the community to adopt NEOP . An example is the demand on introduction of a new feature to some popular video-sharing mobile apps. A webcast anchor on the apps often desires to have more scenes than that the frond/back-end cameras built in his phone can capture. He then can conveniently live-stream selected scenes, some of which may be feeds from remote sites. The app on an OPPO mobile device with a built-in system-level RDA can quickly make the new feature available by accessing remote cameras, while the one on a third-party device without a system-level RDA in its system service can still enable the feature by upgrading the apps with the RDC SDK.

Note that RDP and RDC can simultaneously run on one device, which can then provide service to remote apps as a RDP and access remote devices as a RDC. One example device that plays the dual role is a smart watch. A watch usually does not have a camera. If an app on the watch needs to take a photo or create a video stream, it can assume the role of RDC and requests service from a remote smartphone with camera(s). In the meantime, for an app on the phone the watch is a more accessible device to take user inputs and display its quick response. In this case, the watch acts as a RDP to provide service for the RDC on the phone.

To implement the remoted device service as an extension of Android, NEOP uses Android's existing code components and system infrastructure as much as possible to take advantage of well-tested Android code base and to help with NEOP's market acceptance. As an example, in Android's audio HAL of the *r_submix* type, an audio stream is sent to AudioRecord/AudioTrack, an Android component for data buffering with well-defined interfaces for accessing its data stream. For a local speaker, the data stream from the AudioTrack is converted to the PCM stream for playing out at the speaker. For easy access of the audio stream, NEOP's remote audio HAL is also of the *r_submix* type. RDC can then use AudioTrack's output interface to conveniently receive the audio stream and route it to a remote speaker's RDP. This avoids the chores of creating another interface to access the audio output stream, such as the PCM stream from Audio HAL, and makes the implementation more Android-compatible.

E. Distributed Service Governance

In NEOP's distributed service platform, a service becomes the main entity for management. It can be registered, discovered, activated, or removed across multiple devices and the cloud. This management module is named Service Governance, whose architecture is depicted in Figure 3. A cross-device service's implementation at a device is encapsulated in a PSD (Public/private Service Descriptor). A service can be delivered only when it is associated with a PSD. A service can be a public one, which is either built in the NEOP system (e.g., MediaStore for providing media of common types, such as audio, video, and image) or provided by a third-party developer (e.g., GPS navigation service from a map provider). A public service needs to be certified by the platform before being published. A private service is available only to apps developed by the service provider. For example, a sports equipment supplier may install a private service at its equipment, such as the treadmill and bike, for

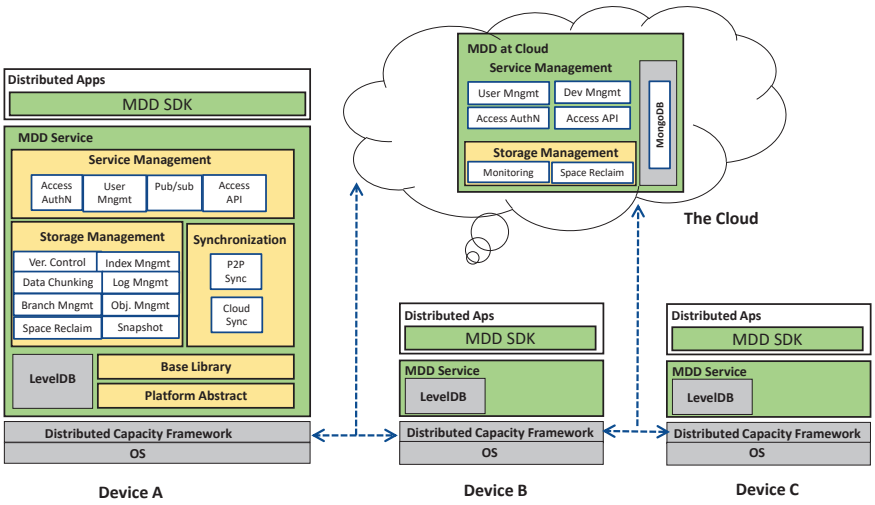

Fig. 4. The architecture of the Distributed Data Management module

its mobile app to remotely monitor and control the equipment. A service's PSD includes a list of TypeIDs, each representing an implementation-independent service (such as Google's BII (Built-in Intents)) that this PSD has implemented if the service is a public one or service URIs (Uniform Resource Identifier) if it is a private one. It also contains DistPackgeInfo, indicating package name, version code, app icon, and permission, as well as service status, such as INSTALLED, AUTHORIZED, or DEVICE_ONLINE/OFFLINE.

The service governance module has two major subsystems to make cross-device services available and accessible. One is service discovery. And the other is service invocation. For service discovery, a device has a PSD Manager module for locally provided services. New services are first registered in the cloud. The module periodically retrieves a list of public services (PSDs) matching the device type from the cloud, where all registered public services are recorded. For each new or updated PSD it checks its corresponding package in local Android's PMS and places matched PSDs into a local white list. The module then scans the local packages to see if some of their services are declared as distributed ones, which are added to a list of private services. The updating of the white lists for public and private services in the PSD Manager may take place with each installation/removal of packages to keep them up to date. The cloud and the Remote Service Manager module will be notified of the updating. Each Remote Service Manager dynamically tracks available services published by other devices. It answers queries about availability of given services as well as their hosting devices. With constant communications between one device's PSD Manager and another device's Remote Service Manager, information on up-to-date available PSDs on the remote devices can be well maintained to allow apps at a device to have a global view of the entire system's service availability.

To invoke a service, including data service, within a device in Android, one has a number of methods, such as Intent, Binder, Content Provider, and Broadcast. As services on remote devices are made visible locally, NEOP extends the methods to invoke remote services and can enable app-level access of the distributed services with a NEOP SDK. When it turns out that the services to be invoked are in the local devices, the extended methods are functionally equivalent to their respective original ones. It is optional to include info such as device ID or device type in the methods' APIs, depending on if the apps explicitly designate certain device(s).

A service request is first sent to the Dispatcher module. The module then passes the request to the Selection of Device Service module, which matches the service requirement to the PSDs in the Remote Service Manager. If there are multiple PSDs meeting the request requirement, the module may either select one according to factors such as user's use history and location, or ask for a selection from the user. The selection module returns the selected PSD and device ID to the Dispatcher module, which then contacts the Launcher module. The Launcher invokes one of the extended methods with the PSD and Device ID as additional parameters. Note that before the cross-device invocation is made, a permission has to be granted at both device ends.

If the Service Governance is implemented at the system level, the apps do not need to be modified. Instead, the Dispatcher and Launcher modules become part of the Android's AMS (Activity Manager Service). Remote Service Manager and PSD Manager will supply their PSDs in response to queries from Remote Component Resolver of Android's PMS (Package Management Service [7]), which makes the selection of PSD for a requested service. While the PMS is capable of carrying out service discovery, AMS sends its query to PMS for a PSD for a service request from a user. It then uses one of the methods for service invocation at a remote device.

F. Distributed Data Management

It is desirable for a distributed mobile computing platform, such as NEOP, to have the support of a fully distributed data service, instead of relying on a centralized database, such as one at the cloud, for data access and sharing. A fully distributed data management system allows user data

to stay at the end devices with the cloud storage as an optional component in the system. In such a peer-to-peer (P2P) structure, user data can be accessed and shared among local devices often leading to lower latency and higher bandwidth by removing round-trip communications with the cloud off the critical path. This is especially beneficial in scenarios like sharing of a large volume of streaming data or networking on a bandwidth-limited mobile data service. With the potentially high-performance data transfer directly between end devices, a user can perform a copy&paste operation between two devices, or switch video playing at a smartphone instantly to a playing device in a smart car once he enters the car.

The goal of this Distributed Data Management is to enable a Multi-End Distributed Database (MDD) that provides apps in the platform with access of a light-weight data service via a high-level API. The database is self-synchronized across end devices and the cloud. The apps have a consistent global data image without being involved in the management tasks such as data placement, migration, and synchronization operations.

As shown in Figure 4, the Data Management module has a device-end MDD component in each end device (specifically, the MDD SDK to support apps' service access and the MDD Service module for data management) implemented at the app level. It also has a cloud-side component as an option to assist data synchronization.

The MDD Service module includes Service Management component, Storage Management component, Synchronization component, Base Library, and Platform Abstract. The Service Management component is in charge of data security, user accounts, data publication and subscription, and implementation of API in the MDD SDK. In particular, to access the data service, both user and app need to be authenticated. It relies on HeyTap Account SDK for user authentication. It obtains User-Token for account login, uid generation, and token refresh. It uses the OCS SDK for app authentication. Furthermore, data is encrypted before its storage and communications. It leverages the Storage Management module for data access.

The Storage Management component includes core database service functionalities, such as Index management, MVCC Versioning Control, Data Chunking, Log Management, Space Reclamation, and Snapshot. It receives remote data from the synchronization component for local data persistency and sends local new or updated data to remote devices or the cloud for synchronization. The actual local data storage is implemented in the LevelDB subsystem, a key-value store optimized for high-performance writes. The Platform Abstract module provides a set of functions commonly needed at devices, such as system time and socket.

To make data well organized and to conveniently enforce access control, the platform places all data belonging to the same user into a container named repository. All data from the same app of a user is in a ledger. A ledger consists of a number of pages, each of which stores a number of key-value pairs. Therefore, a repository is owned by a user and can be accessed by authorized users. Similarly, only authorized apps can access data in a ledger. A ledger may have multiple presences on devices, which are synchronized at the page unit.

This design accommodates a number of much desired properties of the system. First, the design is efficient and scalable. While both the number of devices and the number of apps in a device in a NEOP platform may keep increasing, the amount of the data and number of data objects will correspondingly increase. Because the system needs to track and automatically synchronize data distributed on the platform, the management cost is likely to become excessive and non-scalable. This issue is addressed by limiting management operations within individual users/apps. In the system's design, data from different users and apps are placed into different ledgers. The management system only provides data sharing with a ledger. In particular, a ledger consists of multiple pages. And the synchronization is provided only among a page and its replicas on other devices. That is, tracking data change and performing data synchronizations are at the unit of page, rather than for individual data entries or an entire ledger, for low management cost and high data transfer efficiency. Second, the format selected for data storage and sharing is general enough to conveniently support differently structured user data. User data can be a short text message, a large video file, and metadata of a distributed file system. All of these data can be represented as key-value pairs (entries). In particular, a large file can be split into multiple entries using Content-defined chunking (CDC) for data de-duplication. Metadata about a file-system directory can be represented as a collection of (inode, directory content) entries. And a database table can be conveniently broken down into (row, column) cells as entries in a KV store. Third, the design leverages existing mature software codes in its implementation. In particular, it organizes all KV pairs in a page into a KV store and uses LevelDB for its management. It uses Fuchsia's distributed storage system [8], Ledger [9], for its data synchronization, including conflict resolution strategy.

III. NEOP in the Context of its Related Work

NEOP has been motivated by availability of diverse mobile devices, demands from their consumers, and development trend of mobile technologies (in particular, Android-based mobile service platform). Its design represents a convergence of years of efforts in the research community and vision in the IT industry on next-generation distributed mobile platforms.

In the research community, people have long seen the opportunities of leveraging diverse devices, such as TV displays, cameras, microphones, and speakers, in a mobile device environment to expand capability of apps. There are many proposed point solutions on specific devices, types of apps, and use scenarios. However, compared to NEOP's framework-level system solution, these proposals have one or more of the limitations. (1) Some assume a shared service (or a functionality in an app) that has its unique set of APIs. And an app has to be modified accordingly to access such services, such as Miracast [1], MightyText [10], IP Webcam [11], WiFi Speaker [12], and Remote Desktop Services [13]. The specialized remote service sharing is highly limited and is thus

unlikely to become a general solution for any applications to access shared resources. NEOP remotizes shared devices with interfaces the same as their local counterparts, such as local camera, screen, and speaker. This makes it possible for an app to access remote services simply via transparent service re-direction in the platform. (2) There have been efforts on a cross-device platform for unmodified apps to access resources shared by remote devices. These designs are often concerned only with sharing a particular resource. For example, AnywhereUSB [14] and Swiss Army Smartphone [15] make the USB device remotely available. And Flux [16] allows multiple remote screens to be used simultaneously with the display of a local app. Some designs rely on changes in low-level implementations for re-directing I/O data stream for I/O device sharing. One example is Rio [17], which enables access of remote devices via redirecting I/O to the local device file. While this approach makes transparent device sharing possible, its low-level sharing strategy limits its ability for service discovery and selection. In contrast, NEOP extends Android's facilities exposed to app developers to reach remote services, instead of remote devices' low-level functions. Specifically, Binder and Intent are upgraded to remote Binder and remote Intent, respectively. If a remote service, including a remotized device that is presented as a remote service, is discovered and selected for the sharing, the Binder or Intent is transparently delivered to the remote service. All of the shared resources are managed in one common service framework. This approach enables service-level sharing and access of remote services, such as in-app payment and SNS login. (3) Some share services at the device level, rather than at the app level. For example, Vysor [18] and Chromecast [19] project a screen from one device to another device's screen. However, it does not support well-controlled sharing of a display, such as only migrating a movie app's output but keeping a messaging app's output at the local device. NEOP can conveniently support differentiated access of remote resources for different apps and even different windows of an app,

While NEOP is a platform supporting development and running distributed apps, HarmonyOS [20] is developed as a distributed OS for mobile and IoT devices. It enables seamless interconnection and coordination between smart devices over DSoftBus, a unified communication infrastructure. HarmonyOS uses a component-based design approach to tailor its features to better fit into devices with relatively small amount of DRAM (at 128 KiB to GiB-level). It intends to provide an ecosystem as an Android alternative. As a comparison, NEOP is developed to be a distributed app execution platform as an extension of Android system, and maintains its compatibility with existing Android. Its potential acceptance and impact can be larger because (1) Android has penetrated and dominated the mobile device and smart device markets; and (2) NEOP is designed for the Android ecosystem from its beginning.

IV. Conclusions

In this paper, we propose NEOP , a distributed Android app development and execution platform where various mobile and IoT devices can dynamically participate and make their capabilities available as remote services to apps running at different devices. While it supports apps that are developed in place to naturally use remote services, existing Android apps can also be readily ported to the platform to become distributed. This platform is developed in response to explosive emergence of smart devices with rich and diverse capabilities in various places and great demand on easily accessing of them. Instead of being simply a collection of point solutions for individual devices/services, NEOP provides a coherent framework with architectural and protocol supports for devices to publish their services and for apps to reach them. Furthermore, it is highly extensible to include new features and protocols allowing new capabilities to be deployed and used.

As the NEOP platform is designed to introduce new features and capabilities in the Android ecosystem, this paper focuses on the architectural design and technical details concerning integration of NEOP's design seamlessly into existing Android's system or service modules. Its performance, such as device connection/response times and data synchronization/access times, are comparable to their counterparts in the the existing Android ecosystem. Actually as NEOP is developed for and runs on the existing infrastructure, including networking and the Android framework, its performance is mostly determined by the existing Android ecosystem and will benefit from continuous improvements on its efficiency. Currently, NEOP has been in the phase of internal testing and vendor trials, and its release is in the plan.

References

[1] "Miracast," http://www.wi-fi.org/wi-fi-certified-miracast.
[2] "Using binder ipc," https://source.android.com/devices/architecture/hidl/binder-ipc.
[3] "Intent," https://developer.android.com/reference/android/content/Intent.
[4] "Content providers," https://developer.android.com/guide/topics/providers/content-providers .
[5] "Broadcasts overview," https://developer.android.com/guide/components/broadcasts .
[6] "App actions built-in intents," https://developers.google.com/assistant/app/reference/built-in-intents .
[7] "Packagemanager," https://developer.android.com/reference/android/content/pm/PackageManager .
[8] "About fuchsia," https://fuchsia.dev/ .
[9] "Ledger," https://fuchsia.googlesource.com/fuchsia//+/020330bdeddad1b77e8a866ff6202106ba8e200f/src/ledger/docs/README.md .
[10] "Sms text messaging," https://goo.gl/oLXH0T.
[11] P. Khlebovich, "Ip webcam," https://goo.gl/FQgQst.
[12] W. Morrison, "Wifi speaker," https://goo.gl/N128Ar.
[13] "Microsoft remote desktop services." https://technet. microsoft.com/en-us/windowsserver/ee236407.aspx.
[14] D. International, "Anywhereusb," http://www.digi.com/products/usb/anywhereusb.jsp.
[15] A. Hari, M. Jaitly, Y.-J. Chang, and A. Francini, "The swiss army smartphone: Cloud-based delivery of usb services." New York, NY, USA: Association for Computing Machinery, 2011.
[16] 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*, ser. EuroSys '15, 2015.
[17] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A system solution for sharing i/o between mobile systems," ser. MobiSys'14.
[18] ClockworkMod, "Vysor," https://www.vysor.io/, 2019.
[19] Google, "Chromecast," https://store.google.com/product/ chromecast, 2019.
[20] Huawei, "Harmonyos," https://www.harmonyos.com/.