Interconnecting Heterogeneous Devices in the Personal Mobile Cloud

Yong Li and Wei Gao
Department of Electrical Engineering and Computer Science
University of Tennessee at Knoxville
{yli118,weigao}@utk.edu

***Abstract*—Recent diversification of mobile computing devices allows a mobile user to own multiple types of devices for different application scenarios, but also results in various restrictions on the performance and usability of these devices. A viable solution to such restriction is to incorporate and interconnect mobile devices towards a personal mobile cloud where these devices can complement each other via cooperative resource sharing, but is challenging due to the heterogeneity of mobile devices in both hardware and software aspects. In this paper, we propose a novel design of resource sharing framework to address these challenges and generically interconnect heterogeneous mobile devices. Our basic idea is to mask the hardware and software heterogeneity in mobile systems by exploiting the existing mobile OS services as the interface of resource sharing, and further develop the resource sharing framework as a middleware in the mobile OS. We have implemented our design over various mobile platforms with diverse characteristics and resource limits, and demonstrated that our design can efficiently support generic resource sharing among heterogeneous mobile devices without incurring significant system overhead or requiring individual system modification.**

I. Introduction

Nowadays, a mobile user is usually equipped with multiple types of mobile computing devices, ranging from traditional smartphones and tablets to emerging wearables, each of which is designed for a specific application scenario. Such diversified designs satisfy the unique requirements of different application scenarios, but also restrict the performance or usability of these mobile devices in other aspects. For example, wearable devices enable body sensing with a small form factor, at the cost of limited capacities in computation, communication and battery life. A viable solution to eliminate such restriction is to construct a *personal mobile cloud* [6], which incorporates and interconnects all the mobile devices owned by a user via wireless links. These devices are then able to flexibly share system resources with each other, augmenting the mobile computing capability provided to the user. For example, wearables can save their local battery by exploiting the computational power of nearby stronger devices [4], [11], [19], while providing their sensory data to these devices and facilitate their context-aware applications [9], [22].

The major challenge of realizing such a personal mobile cloud is the heterogeneity of mobile computing devices, which resides in both hardware and software aspects and prevents these devices from being interconnected in a generic manner. First, the increasing variety of hardware components being mounted on today's mobile devices results in fundamental difference in the drivers, I/O stacks and data access interfaces being used by these hardware. Even for the same type of hardware, access to the hardware data from a remote system could fail if the hardware drivers are provided by different manufacturers and incompatible with each other. Such incompatibility is usually a result of customized SoC designs adopted by different hardware manufacturers. For example, the accelerometer drivers for the Qualcomm Snapdragon chipsets are definitely incompatible with the Samsung Exynos chipsets. Second, the complexity of today's mobile applications has been dramatically increased, leading to heterogeneity in both their requested types of mobile system resources and their specific ways of accessing these resources. Existing solutions, unfortunately, are limited to interconnecting mobile devices with respect to an individual mobile application [16], [23] or a specific type of shared hardware [3], [17]. Therefore, they will need a large amount of reprogramming efforts to interconnect heterogeneous mobile devices, by rewinding the wheel for each individual hardware or software component of these devices. Such reprogramming efforts do not only impair the usability of mobile computing system in versatile environments, but also incur additional overhead to the operation of mobile OS and hence reduce the mobile system performance.

The key to generic interconnection across heterogeneous mobile devices is to develop an efficient framework for resource sharing between these devices, which appropriately masks the hardware and software heterogeneity in mobile systems from each other. Development of such a resource sharing framework, however, is challenging due to the close interaction between mobile hardware and software. A framework at the lower layer of mobile OS hierarchy unifies the heterogeneous resource requests of mobile applications, but has to tackle with individual hardware drivers which are operated in intrinsically different ways [2] and hence incurs a tremendous amount of re-engineering efforts. Sharing system resources at the application layer, on the other hand, is able to access mobile hardware through a generic OS interface, but has to be associated with specific data transfer protocols and hence has limited generality [5], [8].

In this paper, we present a mobile system framework to address the above challenges and generically interconnect heterogeneous mobile devices towards a personal mobile cloud. Our basic idea is to develop the resource sharing framework as a middleware in the mobile OS, which exploits the existing mobile OS services to share resources between mobile devices. These services hide the low-layer details of device driver operations while providing unified data access APIs to user applications. Interconnection between mobile devices,

This work was supported in part by the National Science Foundation (NSF) under grant number CNS-1456656, CNS-1526769, CNS-1527612 and CNS-1553395.

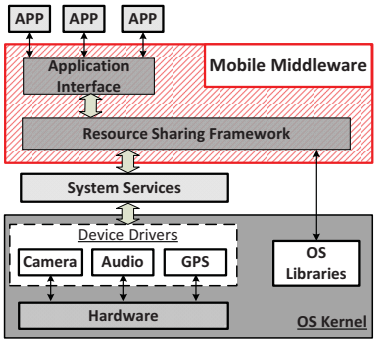

Fig. 1. The big picture of interconnecting heterogeneous mobile devices

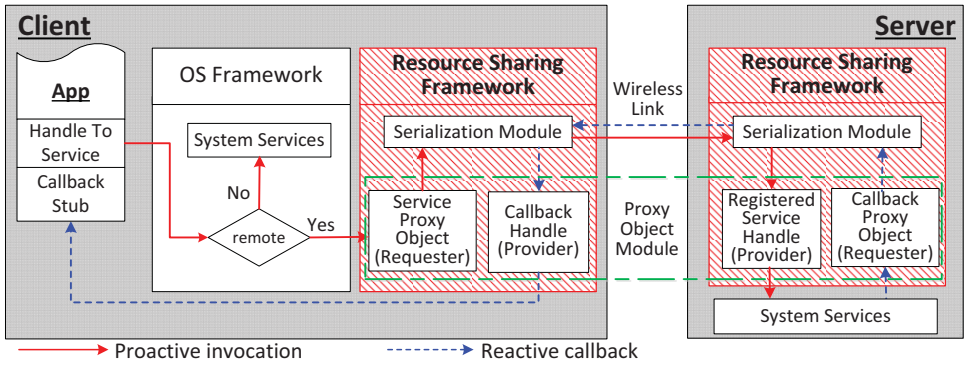

Fig. 2. Design of resource sharing framework

then, could be realized via remote access and invocation of these OS services. Since these services are executed as a standalone system process by the OS kernel and are separated from application processes, remote service invocation can be done via inter-process communication (IPC) between mobile systems without involving complicated issues such as memory referencing and synchronization. As a result, any new device can be incorporated into the mobile cloud by inserting our framework into its OS, without modifying the OS kernel, our framework itself, or the source code of any mobile application.

We have implemented our design on Android OS with less than 5,000 Lines of Codes (LoC) over various mobile platforms including smartphones, tablets and smartwatches, and demonstrated the efficiency of sharing various types of hardware (GPS, accelerometer, audio speaker, camera) between remote mobile devices. The evaluation results show that our design can efficiently support ubiquitous resource access between remote systems with arbitrary mobile applications accessing these resources, without incurring any significant system overhead. Our proposed framework is also fully compatible with existing application-level remote messaging protocols (e.g., MQTT and XMPP), and hence can also be efficiently exploited for mobile application development.

The rest of this paper is organized as follows. In Section II we provide a high-level overview about our motivation and designs. Section III and Section IV present the details of our resource sharing framework and application interface. Section V describes how we support sharing multimedia resources between mobile devices. Section VI presents our implementations over various mobile platforms, and Section VII presents our evaluation results based on these implementations. Section VIII discusses the related work. Finally, Section IX discusses and Section X concludes the paper.

II. Overview

In this section, we start with a brief description about the layered architecture of mobile OSes, which motivates our proposed design. Based on this motivation, we further provide a high-level overview of our proposed framework.

A. Motivation

Our design is motivated by the layered architecture of mobile OSes, which isolates user applications from low-layer system implementations. Such isolation allows more efficient management of the limited mobile system resources, and also protects the mobile system from resource depletion due to poorly designed applications or malicious attacks from mobile malware. Mobile applications in such a hierarchical OS architecture do not access system resources directly. Instead, resource access is provided by system services via a suite of generic and pre-defined APIs, which are invoked by user applications via IPC with binder mechanism in Android and message passing in iOS, respectively. For example, instead of directly accessing GPS or WiFi network interface, an application in both Android and iOS retrieves the location information of the device via a location service provided by the OS. Then, these system services interact with hardware device drivers through a hardware abstraction layer (HAL), whose interfaces are pre-defined by the OSes and implemented as libraries by the manufacturers.

We support generic resource sharing between remote mobile devices based on the such layered architecture of mobile OSes. First, since system services are the only interface for user applications to access system resources, access to any type of resource on a remote device could be provided by the same generic framework, as long as this framework can intercept the requests of resource access from user applications and redirect these requests to the remote system. Furthermore, different types of system services are invoked following the same mechanism (e.g., binder in Android and message passing in iOS), and hence we will not confront with the heterogeneity of service operations. Second, these system services hide the details of hardware operations from user applications. Hence, resource sharing based on these services addresses the heterogeneity of hardware driver implementations, and allows devices with different hardware models and drivers to access each other. More importantly, since system services are invoked through the pre-defined set of APIs, interception and redirection of these invocations are transparent to user applications, which will access the remote system resources in the same way as they access the local counterparts, without any modification to their source codes.

B. The Big Picture

As shown in Figure 1, our proposed middleware resides between user applications and OS services, and consists of two major components: *a)* Application Interface and *b)* Resource Sharing Framework.

The *Application Interface* regulates how a user application accesses the shared resource at the remote system through an application-specific metadata file that configures the usage of remote resource. When a user application requests to access a system resource, the Application Interface parses

the configurations to return the appropriate handle to the corresponding system service. Hence, no matter what system service is invoked and whether the service is invoked at the local device or the remote device, the service is operated through the same way.

The *Resource Sharing Framework* interacts with the local OS and communicates with the remote OS services to provide remote resource access to user applications. As shown in Figure 1, the details of low-layer device driver implementations and hardware operations in the OS kernel are completely separated from the resource sharing framework, and different OS services are invoked in a universal manner through the same set of pre-defined APIs.

III. Resource Sharing Framework

Our design of the resource sharing framework is shown in Figure 2. In general, our framework intercepts the requests of resource access generated from local mobile applications, and forwards these requests to another remote mobile device which acts as the server and provides the shared resource. Every time when a resource access request is received, the server will invoke its local OS service corresponding to the requested resource, and reply with the resource data.

This framework consists of two major components: *Proxy Object* module and *Serialization* module. The responsibility of the Proxy Object module is to serve as a portal of remote service invocation, and manage the proxy objects for resource sharing at both endpoints of the resource sharing system. The Serialization module is responsible for data serialization, which is the process of converting memory objects into a binary format that can be transmitted through the network link. These binaries can be reconstructed back to memory objects by deserialization at the other endpoint.

Our framework supports resource access between mobile devices in two ways: *proactive invocation* and *reactive callback*. First, when a remote system service is available, a service proxy object will be created by our framework to initiate the IPC between the client and the server. In proactive invocation, every time when an application requests to remote service access, the proxy object at the client triggers a remote invocation event, which is captured at the server to invoke the corresponding service method. Second, applications can also access system resources reactively by receiving data in system events, e.g., location update. Resource access in this case is handled by reactive callbacks, which allow applications to register and listen to a system event with a callback handle. This handle is called via a callback proxy by the service at the server when the system event occurs, and then used to transmit resource data back to the client. This invocation procedure is similar to proactive invocation, but in a reverse direction.

A. Invocation of Remote OS Services

Our framework supports remote invocation of both Java-based and native OS services. As we mentioned above, both types of services can be remotely invoked via both proactive invocation and reactive callback.

1) Java-based OS Services: In Android, part of system services are implemented in Java and running in a standalone system process. On one hand, to devise a generic solution to the serialization module for sharing these services, we utilize the *Java reflection mechanism* which enables developers to inspect classes, interfaces, fields and methods at run-time without knowing the names of classes and methods in advance. Specifically, we access all the field values of any memory object by reflection and convert them into binaries. Similarly, this feature can be applied for deserialization by creating the object instance and setting all the field values at run-time, so as to reconstruct memory objects from the received binaries.

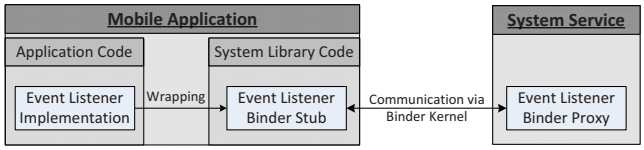

Fig. 3. Resource sharing through callbacks

On the other hand, we develop the proxy object from the existing system service class, by appending the bytecodes of remote invocation operations to the service class via *dynamic weaving*. The dynamic weaving technique in Java allows us to instrument the bytecodes of an existing Java class at run-time, generating a new class instance as the subclass of the original class type. Since a system service class in Android is specified as a subclass of the Binder class by itself, we dynamically weave a system service class at run-time whenever it will be remotely accessed, with the extra bytecodes of remote invocation operations added to service methods. Hence, this newly weaved class will be a subclass of the Android Binder class and can be registered to binder kernel driver to receive and intercept the applications' invocation to a service method.

Our framework also supports callback to user applications which register a system event with any class object implementing the event listener interface. Then, as shown in Figure 3, user applications exploit the Android system library to wrap this listener into a binder stub which communicates with the binder proxy for event listening in the corresponding system service. To realize such callbacks across two mobile devices, as shown in Figure 2, we allow an application to register and listen to an event in the remote system via a *callback proxy*, and utilize the event listener binder proxy at the client as the *callback handle*. Afterwards, when the system event happens at the server, the callback proxy will initiate a remote invocation to the callback handle at the client.

2) Native OS Services: Another large body of system services in existing mobile OSes is implemented in native C/C++ languages, e.g., sensor and graphic services in Android and all system services in iOS. Since these services are executed as compiled binaries at run-time, proxy objects for these services cannot be developed through run-time manipulation due to the following reasons. First, it is hard to locate the entry and exit points of a native method at run-time, and hence difficult to dynamically attach the weaving instructions to the service class. Second, the machine instructions in these native service classes depend on the hardware architecture and hence can only be operated and compiled statically.

To address this challenge, in our current design we modify the source codes of each system service class to realize the functionality of serialization and deserialization, as well as the remote invocation of service methods. Being different from existing schemes which share system resources over the device drivers and have to reprogram the driver implementations for

each individual system, our modifications are applied to OS services and applicable to all mobile systems with heterogeneous hardware components. This advantage enables our work to be applied to a variety of different mobile platforms, and we will describe such implementation details in Section VI.

B. Unix Domain Socket

Another IPC scheme supported in our framework is *Unix domain socket*. For example in Android, sensor data is not delivered from the sensor service to applications as method arguments of the binder method invocation. Instead, it is delivered by a Unix domain socket between the sensor service and the application, in order to reduce the system overhead of highly frequent data transmission.

In our design, we build the reliable data exchange channel between mobile devices with a network socket instead of the Unix domain socket used in existing IPC schemes. More specifically, the system service process in the server will listen to the connection request for building a distributed data channel. When an application requests a remote system service, the proxy object in the client will establish a connection to the server and pass the file descriptor of this connection back to application. Thereafter, the system service in the server can exchange data reliably with the client application through TCP. Since the mobile OSes operate these two types of sockets with the same APIs, the actual type of the data channel socket can be hidden from the system which uses the data channel and the existing IPC code for data exchange can be kept unchanged.

IV. Application Interface

Our design of the Application Interface is shown in Figure 4. Whenever a user application requests to access an OS service, the Application Interface intercepts this request and returns a handle to the appropriate service, which could be located in either the local system or the remote system. Since both handles provide the same programming interface to applications, the process of resource sharing between mobile devices is completely transparent to user applications.

Decisions on the service handle being returned are made based on the configurations stored in an application-specific *Metadata File* in the application directory. More specifically, when a user application requests to access a system service, the framework loads the configurations from the metadata file. If the configurations indicate that a local system resource will be accessed, the local service handle is returned to the user. Otherwise, our framework will create a remote service proxy and build a TCP connection to the remote system for resource access. In our design, this metadata file is operated by a special *Proxy Application* which is embedded as part of the Application Interface. This proxy application manages and overrides the resource access configurations for all user applications, and also receives inputs from the mobile OS settings about the list of available system resources. All these information will be written by the proxy application into the metadata file, which are then checked by our framework at run-time to ensure correct service invocations.

Through the development of this proxy application, our design allows a user application to configure its access of system resources in three ways. First, we allow developers to distribute their configurations along with the application

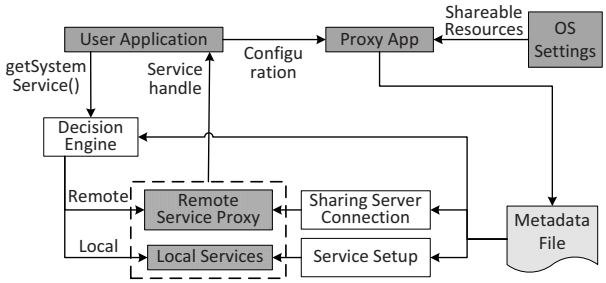

Fig. 4. Design of application interface

binaries during installation, and explicitly specify how the application will access system resources. For example, the developers can decide the destination of the remote resource and the data rate at which they want the remote resource to be accessed. Second, if the target for resource sharing is unknown, developers can opt to adopt existing service discovery protocols (e.g., [24], [18]) and explore for the available shared resources nearby, by specifying the service discovery protocol being used in configurations. Third, we also allow mobile users to manually modify the configurations of resource sharing via the proxy application. For example, a mobile user can explicitly configure the application to project the screen content to a nearby large LCD.

V. Supporting Multimedia Operations

Operations over multimedia resources usually involve large sizes of bulk data and need to exploit shared memory for data exchange between applications, resulting in new challenges when sharing these resources between remote mobile devices. In this section, we present our design to support the access to such shared memory of multimedia services between remote mobile devices. The major challenge, however, lies in how to ensure the remote IPC reliability with the minimum intrusion to the original mobile OS structure and interface.

According to the way shared memory is operated, we categorize the shared memory into two cases and handle them separately: the general buffer and the graphic buffer. The general buffer is defined as shared memory that is portable and vendor-independent. It is directly allocated and operated by system services. The graphic buffer, on the other hand, stores image data such as camera preview and video frames, and is usually operated by vendor-specific HAL.

A. The General Buffer

The general buffer can be divided into two parts. First, the *content part* stores the resource data and is operated following the producer-consumer pattern, i.e., one operator always writes data into the buffer and the other always reads the data. In this way, the two operating parties are always synchronized without contentions on writing. Second, the *control part* stores the control information that is necessary to operate the content part, such as the reading and writing pointers. The control part, therefore, can be written by both applications and system services which may conflict with each other when writing. In our design, we develop different techniques for synchronizing the shared memory of these two parts.

1) Content Part: To efficiently operate the content part, being different from traditional approaches of Distributed Shared Memory (DSM) [10] which always shares memory in

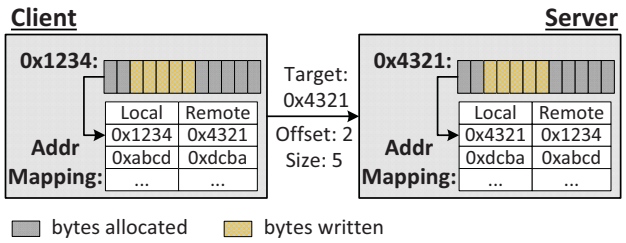

Fig. 5. Memory synchronization for content part

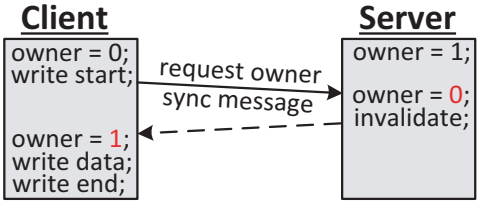

Fig. 6. Memory synchronization for control part

fixed-size units and may result in large amounts of redundant data synchronization between mobile devices, we flexibly synchronize the shared memory at arbitrary sizes based on the actual application patterns of memory access. This flexibility is mainly due to the fact that there is no write contention between the sharing client and server when they synchronize the content part, ensuring any size of synchronized memory to be always coherent. As a result, our basic idea is to establish a memory mapping between the client and the server, and synchronize the memory contents based on the mapping. Whenever one endpoint allocates a block of shared memory, a corresponding memory block with the same size is allocated correspondingly at the other endpoint, and a mapping entry between their addresses is added into the mapping table maintained at both endpoints. Whenever one endpoint finishes its write operation, we use the mapping entry and the writing offset to calculate the destination writing address and synchronize the data being written. For example in Figure 5, when the client allocates 12 bytes of memory, the server also allocates the same amount of memory accordingly, and the addresses of both the client and server memory are stored in the mapping table in both endpoints. Later, when the client writes 5 bytes into the buffer, the resource sharing framework synchronizes and updates memory with the appropriate size, according to the received target address and offset.

2) Control Part: We also develop a flexible synchronization scheme for the control part, which can synchronize either the whole control part or the individual fields of it. To ensure data consistency with write contention, we establish a happened-before relation between two mobile devices through an ownership flag, and only allow the owner of a memory field to write to the field. The ownership transfers when the other endpoint tries to write the field. In this way, we ensure that writes happen in sequential turns between the client and the server and hence the memory at two endpoints will always be consistent. For example in Figure 6, the client intends to write a control field but does not have the ownership to the field. Therefore, the client sends a message to the server and requests for the ownership of the field. Having received this message, the server transfers its ownership to the client.

B. Graphic Buffer

Graphic services in the mobile OS, which operate multimedia devices such as the camera or LCD screen, usually utilize the GPU to accelerate the speed of image processing and rendering. Hence, being different to the general buffer which is allocated and written by the applications or system services themselves, the graphic buffer are allocated and operated by the vendor-specific HAL. As a result, we cannot directly intercept the buffer operations from our resource sharing framework and further establish memory mapping for synchronization between remote systems. Instead, our approach is to integrate our resource sharing framework with the APIs provided by the OS for user applications to manage and operate the graphic buffer.

We use Android OS as a nominal example to present the details of our design. In Android, the core of its graphic services is the *BufferQueue* class, which manages different graphic buffers allocated by the vendor-specific *gralloc* module. The operation of graphic buffers also follows the producer-consumer pattern: the producer (usually the hardware device driver) dequeues an empty buffer handle from BufferQueue and queues the filled buffer back to BufferQueue; the consumer (e.g., Surface Flinger) acquires a handle of filled buffer from BufferQueue and releases the consumed buffer back.

As a result, our resource sharing framework acts as a consumer of BufferQueue to extract the graphic buffer contents, and then pushes these contents to the remote device. For example in Figure 7, after the camera driver in the server posts a preview image buffer into the BufferQueue, our framework collects the graphic buffer contents as a consumer and sends them to the client. Then, the framework in the client retrieves an empty buffer, fills the buffer with the received image content and posts the buffer back into BufferQueue. Afterwards, the Surface Flinger in the client renders the preview image.

VI. Implementation

We implemented our design in Android v5.1.1 with CyanogenMod 12.1, and build the resource sharing framework on Linux distribution Ubuntu 12.04. The dynamic weaving technique is implemented with dexmaker[1]. Our implementation consists approximately 1,500 lines of Java code and 1,850 lines of C++ code to support Java-based and native system services, respectively. It is deployed on multiple types of mobile devices, including smartphones, tablets and smartwatches.

Based on this implementation, we are able to further implement the functionality of sharing different types of resources between remote mobile devices, and the implementation details are listed in Table I. First, our framework supports remote access of the location service, which involves operations of both GPS and WiFi, with less than 10 Lines of Java code. Comparatively, remote access of other system services involves operations over native classes and requires more LoC on data serialization and deserialization.

A. Service-Specific Optimization

Sensor Service: We optimized the sharing of sensor service and allow mobile applications to receive sensor data at their specified rates, no matter how fast such data is generated by the hardware or OS, so as to minimize the data transmission cost

[1]https://github.com/crittercism/dexmaker/

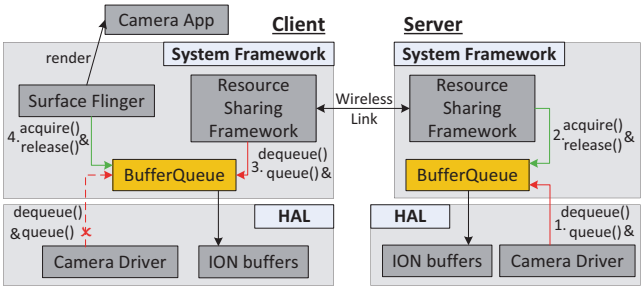

Fig. 7. Operating the graphic buffer for remote camera access

of ubiquitous sensor access. More specifically, we attached a specialized control module to the socket channel for sensor data exchange between mobile devices, and customized the data transmission rate between mobile systems without modifying the sensor service methods or the sharing framework themselves. In practice, the sensor service in the server will receive the sensor data rate from the client, and send sensor data to the client only if necessary.

Audio Service: In Android, the audio playback will not start until the audio buffer is fully filled. However, the amount of audio data that an application writes in one operation may not be enough to fill up the buffer. As a result, mobile users may experience a long latency of initializing remote audio playback if the framework synchronizes as soon as a write operation happens. In order to reduce such latency, our framework accumulates the buffer contents and holds from synchronization until the local buffer is fully filled. Consequently, we eliminate the multiple round trips for the initial buffer synchronization of audio playback.

TABLE I
LIST OF SUPPORTED SERVICES

Service	Type of code	LoC	Hardware
Location	Java	<10	GPS, WiFi network
Sensor	C++	283	All onboard sensors
Audio	C++	647	Speaker
Camera	C++	594	Camera

Notification Service: Besides the system services operating the hardware resources, another collection of system services also exists to let mobile applications utilize the system-wide software resources. Since software system services interact with mobile applications in the same way as hardware system services, our framework also supports sharing of software system services between mobile systems. We have implemented the sharing of Android notification service between mobile systems with less than 15 lines of Java code, and allow the user to show notifications on a remote mobile device. When a mobile user clicks the notification icon, the action associated with this notification will be performed back to the local mobile device through remote callback.

B. Deployment over Different Mobile Platforms

Our proposed framework allows generic resource sharing among heterogeneous types of mobile devices, which are equipped with hardware from different manufacturers or running different versions of device drivers. Being different from traditional DSM-based schemes which have to manually port and reprogram the driver implementations from one device to another, our framework utilizes the OS service interfaces to hide the hardware heterogeneity from user applications and realizes automated migration between mobile platforms

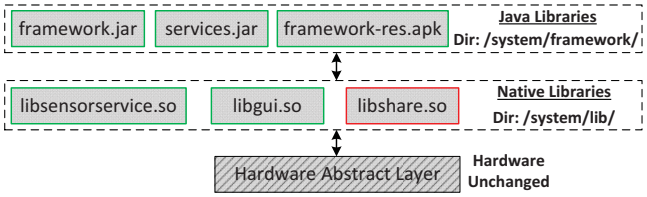

Fig. 8. Porting sensor service libraries in Android wear

without manual modification. In our implementation, we share system resources among smartphones, tablets and smartwatches, even if their implementations of hardware drivers are not open-sourced and not accessible in CyanogenMod. Instead, we exploit the source code of the Android framework service layer, which is publicly available in the Android Open Source Project (AOSP) and remains the same for different platforms.

Deployment of our framework on smartphones and tablets is trivial since they are directly supported by Android CyanogenMod OS. Their deployments can be simply done by building and flashing their full OS installation packages. Deployment over smartwatches, however, is more complicated because the open-sourced OS codes for Android wear is incomplete. We deploy our framework over smartwatches by porting and replacing the existing library files in the rooted smartwatch. Since these libraries are dynamically loaded and linked with interface symbol names in Android, such library replacement will not affect OS execution as long as the new libraries keep the same programming interfaces as the old ones.

For example, for the Android sensor service shown in Figure 8, we build individual modified modules as library files. More specifically, the Java libraries serve as the entry for user applications to interact with the native sensor service, and hence are modified to return the remote service handle to applications. The native libraries receive the service invocation from Java libraries and request the local or shared resources. These individual library files are then used to replace the files with the same names in the directories on a rooted device. Note that, this porting technique is generic and applicable to all other system services such as location service and multimedia services, because the source code of these system services are also available in AOSP and the service libraries are dynamically linked and loaded.

VII. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our proposed designs on sharing resources between remote mobile devices. More specifically, we first evaluate the general performance of resource sharing between remote systems by adopting resource-specific performance metrics, and show that our design reaches satisfiable sharing performance with little computational overhead. Afterwards, we evaluate the power consumption of resource sharing between remote mobile devices, and demonstrate that resources at remote mobile devices can be accessed without consuming significant amounts of energy. Last, we measure the network throughput of sharing different types of resources, and report the amount of wireless network bandwidth required to support resource sharing. Our experiments are performed by sharing GPS, accelerometer sensor, speaker and camera between mobile devices. Note that our evaluations are directly performed over individual hardware components, which are accessed by the mobile OS

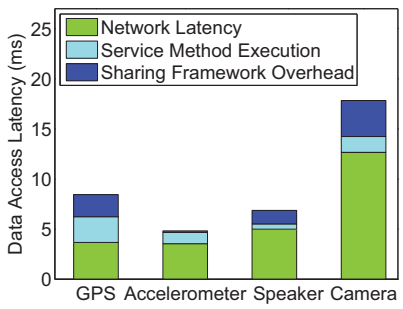

Fig. 9. Latency of accessing shared resources

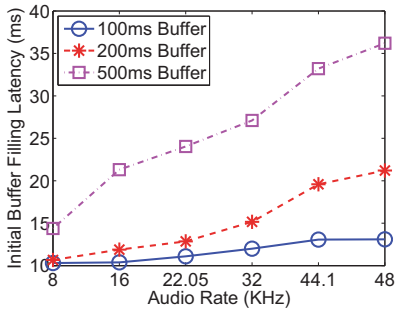

Fig. 10. Latency of initial audio buffer filling

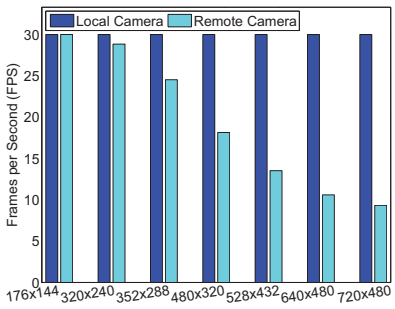

Fig. 11. FPS for real-time camera preview

itself. On the other hand, since our proposed resource sharing framework keeps the method of user applications' resource access as intact, it is able to seamlessly support any off-the-shelf mobile application with the corresponding modification of the resource metadata file.

A. Experiment Setup

We perform our experiments over different types of mobile platforms including Samsung Galaxy S4 smartphone, LG Nexus 4 smartphone, Samsung Nexus 10 tablet and LG Watch Urbane, all of which are running Android v5.1.1. The generality of our resource sharing framework then ensures its reliable execution over these mobile platforms and seamless interaction between different mobile devices. These devices are interconnected via 40Mbps campus WiFi unless explicitly stated in the paper. Our devices are placed close to each other and the network latency is about 3.5 ms. We use a Monsoon power monitor[2] to gather the real-time information about the devices' power consumption. Note that, although in our experiments only one client device is connected to the sharing server, our framework allows multiple clients to be connected to a server simultaneously.

In each experiment, we adjust the parameters of system resources to evaluate the resource sharing performance in different application scenarios. More specifically, we vary the data rates from GPS and accelerometer to emulate the requirements from different mobile applications. We play music with different audio rates which determine the amount of data being transmitted. We also share camera previews with different image resolutions, which significantly impact the data size of the preview image.

B. Performance of Resource Sharing

We first evaluate the access latency of sharing different types of system resources, which is measured by the average elapsed time from the time when the framework starts to process data to the time when the resource data is returned back to the local device. Such latency, hence, consists of the network transmission latency, the execution time of system service methods, and the overhead incurred by our sharing framework. Our experiments use GPS data report interval as 1 second, accelerometer data report interval as 20 ms, audio rate of 44.1 KHz and camera preview resolution of 176×144. Each experiment runs 3000 times, based on which the average data access latency is measured.

The experimental results when sharing resources between two Nexus 4 phones are shown in Figure 9. We can see that remote resource access only incurs negligible latency, which is mainly dominated by the network latency in most cases. Specifically, the network latency for GPS and accelerometer is small because of the small size of resource data being transmitted, and sharing the camera between mobile devices experiences larger network latency due to the large data size of multimedia content. On the other hand, execution of our resource sharing framework only incurs negligible computational overhead to mobile systems at both endpoints.

Furthermore, as we mentioned in Section VI-A, when we share the audio between mobile systems, the audio playback will not start until its buffer is all filled with audio data. Therefore, the latency of initial buffer filling is a key factor of users' conceived delay of using the remote speaker. Our experiments evaluate such latency by measuring the average time it takes to fill the audio buffer since the client starts to write audio data. The experiments set the buffer size as the length of audio segments with respect to the audio rate. Each experiment plays 20 audio tracks. The experimental results are shown in Figure 10. We can see that the initial latency of buffer filling increases along with the buffer size, but is efficiently controlled within 40ms in all cases.

We evaluate the performance of sharing the camera between mobile devices using the average frames per second (FPS) for the camera preview. Our experiments are performed with Galaxy S4 smartphones with different resolutions of camera preview over 2000 frames. From the experimental results in Figure 11, we can see that our framework can reach the same FPS of remote camera preview as that of the local camera, when a low resolution of 176×144 is used. Even if we increase the resolution to 480×320, our framework can still provide a FPS of 18, which is more than sufficient to support smooth camera preview (minimum FPS of 15). When the resolution further increases, the FPS will drop due to the increasing amount of data being transmitted. For example, one 720×480 preview frame has the size of 518 KB with the NV21 pixel format, hence requiring 62 Mbps of wireless network bandwidth to reach 15 FPS at the remote system.

C. Power Consumption

In this section, we evaluate the energy efficiency of our work, by measuring the average power consumption at both the sharing server and the sharing client, and compare such power consumption to the power consumption of using local resources. To remove the dynamic power consumed by the smartphone screen, we disable the functionality of automatic brightness adjustment during our experiments and keep the display dimmest. In each experiment, we use the device for three minutes and measure its average power consumption.

[2]https://www.msoon.com/LabEquipment/PowerMonitor/

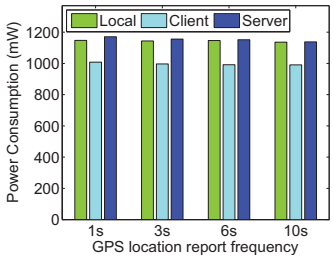

Fig. 12. Power consumption for GPS sharing

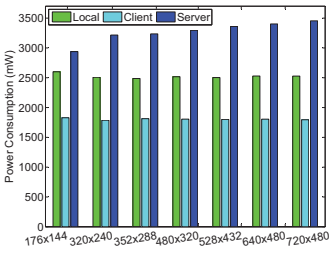

Fig. 13. Power consumption for camera sharing

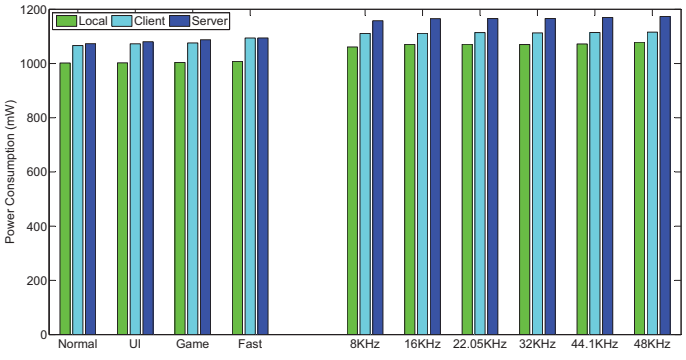

Fig. 14. Power consumption for sharing the accelerometer and speaker

Each experiment runs three times over Galaxy S4 phones that are interconnected via Bluetooth links.

The experimental results for sharing the GPS and camera are shown in Figure 12 and Figure 13 respectively. In contrast to Rio [2] which consumes a tremendous amount of additional energy for resource sharing between mobile system, our framework reduces the power consumption by 13% and 29% respectively, when accessing the remote resources instead of the local counterparts. On the other hand, the server consumes extra energy to provide the shared resources, and the majority of such extra energy is consumed by sending the resource data to the client. Considering that the server is usually the device with stronger capabilities, such additional cost could be acceptable in most cases.

The experimental results for sharing the accelerometer and audio speaker are shown in Figure 14. From the figure we can see that the client consumes a small amount of extra power (7% and 4% for accelerometer and speaker, respectively) to access the remote resource. The basic reason for such additional power consumption is that both of these two resource modules are power efficient but require highly frequent synchronization of resource data, leading to additional energy consumed by wireless data transmission. In particular, adopting a higher audio rate does not noticeably increase the power consumption, because it only leads to moderate change on the size of audio data.

D. Wireless Transmission Throughput

In this section, we evaluate the amount of wireless transmission throughput being produced by the remote resource sharing in our framework, by measuring the average amount of data being transmitted between the client and the server. In each experiment, we use the device for three minutes to synchronize accelerometer data, play audio tracks and transmit camera previews between two mobile devices.

The experimental results for accelerometer, speaker and camera are shown in Figure 15, Figure 16 and Figure 17, respectively. We can see from the figures that sharing sensors, audio devices and camera devices incur small, moderate and large amount of wireless transmission throughput, respectively. In specific, both sensors and audio devices require only less than 1 Mbps to fully support remote resource access. Therefore, any high-speed wireless network can easily meet such throughput requirement [15]. However, remote access to the camera requires much higher wireless network bandwidth due to the large size of preview images. Therefore, the bandwidth becomes the performance bottleneck of the remote camera access, especially when a high-resolution preview is applied. Efficient network scheduling protocols are a viable solution to this bottleneck [14].

VIII. Related Work

Initial research efforts on resource sharing between systems focus on thin client, which allows clients to render graphical interfaces from and send user inputs to the server [3]. However, these systems are limited to solely sharing the graphical interface. Later on, applications have been developed to share different types of system hardware resources such as the microphone, webcam, GPS and computation [12], [20]. However, each application can only share a specific type of pre-designated hardware. Sharing any other type of resource requires a significant amount of engineering work. In contrast, our proposed framework can share heterogeneous types of system resources without incurring any reprogramming efforts.

Traditional work has been focusing on resource sharing in distributed systems. ErdOS [21] exploits opportunistic access to resources in nearby devices to efficiently save local energy consumption. In addition, resource sharing enables a device to utilize its missing resource features so as to be more powerful. Resource sharing among distributed clients has also been supported at the OS layer. Mobile grid computing [13] allows mobile devices to join the grid and share their hardware resources to other devices in the grid. However, these shared resources can only be accessed through grid-specific API. Ubiquitous computing [7], [9] enhances the performance of a system task by sharing and utilizing the resources available in the network. However, these schemes lack generality and are limited to specific applications.

Rio [2] is the first systematic solution to share multiple types of hardware among mobile systems without modifying user applications. However, its implementation over a mobile system has to closely bind with the system hardware drivers, and hence has to be reprogrammed to support different models of hardware. Furthermore, it can only provide remote resource access following pre-designated configurations, and is hence incapable of adapting to the actual resource needs of mobile applications. In contrast, our scheme, which is implemented in a higher layer in the OS architecture, is able to take the run-time application behaviors into account and leave the inconsistency of hardware drivers being dealt by the OS kernel.

IX. Discussions

A. Pixel Format for Remote Camera Sharing

A particular issue in sharing graphic devices, such as camera or LCD display, is the consistency and compatibility of the pixel format between mobile devices. In specific,

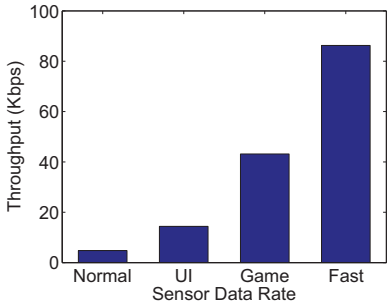

Fig. 15. Throughput for sensor sharing

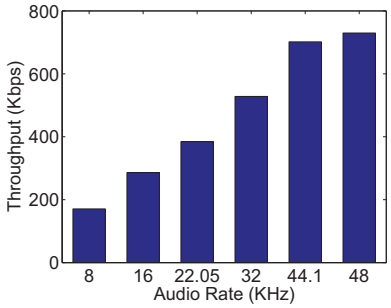

Fig. 16. Throughput for audio sharing

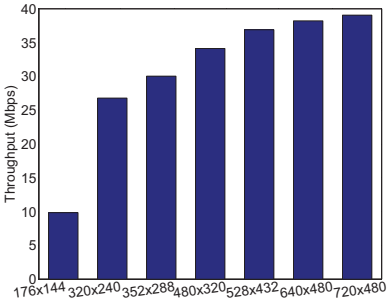

Fig. 17. Throughput for camera sharing

device vendors may define their own pixel format of graphic contents, and hence the graphic data may not be renderable by another device. For example, the preview data generated by the Samsung Galaxy S4 smartphone cannot be directly rendered by a LG Nexus 4 smartphone. The fundamental solution to this issue is to apply a graphic format that is compatible at all types of mobile devices, so that heterogeneous formats of the graphic pixels can be handled in a generic way. For example, the H.264 standard is used as the intermediate data format for memory synchronization by Miracast [1]. Instead of sending the raw graphic contents, the mobile OS first encodes these contents with H.264 codecs. Then, the H.264 data is decoded in the client and rendered to display the graphics on the screen. We will explore the possibility of incorporating such graphic data encoding into our framework in the future.

B. Access Control

An authentication or access control scheme is necessary among mobile systems to protect them against malicious parties which may send mobile malware along with the data sharing traffic or steal private information from users. Such access control can be supported in our framework by adding a new authentication layer before serialization. Various user or device identities, which are not limited to user passwords but can also be biomarkers, system patterns or user gestures, could be used for such authentication. On the other hand, since reactive callbacks need to be registered at the sharing server beforehand with proactive invocation, they can be authenticated in the similar way.

X. Conclusions

In this paper, we present a mobile system framework which efficiently interconnects heterogeneous mobile devices towards a personal mobile cloud and supports cooperative resource sharing among these devices. Our basic idea is to allow a mobile application to access remote system resources at another mobile device through remote invocation of existing OS services. Based on the implementation and evaluation over Android OS, we demonstrate that our framework can efficiently support generic resource access between remote mobile devices without incurring any significant system overhead.

References

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

[2] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A system solution for sharing i/o between mobile systems. In *Proceedings of ACM MobiSys*, 2014.

[3] R. A. Baratto, L. N. Kim, and J. Nieh. THINC: a virtual display architecture for thin-client computing. *ACM SIGOPS Operating Systems Review*, 39(5):277–290, 2005.

[4] 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 ACM MobiSys*, 2010.

[5] W. K. Edwards, M. W. Newman, J. Z. Sedivy, and T. F. Smith. Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. *ACM Transactions on Computer-Human Interaction*, 16(1):3, 2009.

[6] N. Fernando, S. W. Loke, and W. Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106, 2013.

[7] T. Grosse-Puppendahl, S. Herber, R. Wimmer, F. Englert, S. Beck, J. von Wilmsdorff, R. Wichert, and A. Kuijper. Capacitive near-field communication for ubiquitous interaction and perception. In *Proceedings of ACM UbiComp*, pages 231–242. ACM, 2014.

[8] P. Hamilton and D. J. Wigdor. Conductor: enabling and understanding cross-device interaction. In *Proceedings of ACM CHI*, 2014.

[9] M. Hardegger, L.-V. Nguyen-Dinh, A. Calatroni, G. Tröster, and D. Roggen. Enhancing action recognition through simultaneous semantic mapping from body-worn motion sensors. In *Proceedings of ACM ISWC*, 2014.

[10] A. Judge, P. Nixon, V. Cahill, B. Tangney, and S. Weber. Overview of distributed shared memory. In *Technical Report, Trinity College Dublin*. Citeseer, 1998.

[11] 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 IEEE INFOCOM*, 2012.

[12] Y. Li and W. Gao. Code offload with least context migration in the mobile cloud. In *Proceedings of IEEE INFOCOM*, pages 1876–1884. IEEE, 2015.

[13] A. Litke, D. Skoutas, and T. Varvarigou. Mobile grid computing: Changes and challenges of resource management in a mobile grid environment. In *Proceedings of PAKM*, 2004.

[14] H. Lu and W. Gao. Scheduling dynamic wireless networks with limited operations. In *Proceedings of IEEE ICNP*, pages 1–10. IEEE, 2016.

[15] H. Lu and W. Gao. Supporting real-time wireless traffic through a high-throughput side channel. In *Proceedings of ACM MobiHoc*, pages 311–320. ACM, 2016.

[16] J. Lu, T. Sookoor, V. Srinivasan, G. Gao, B. Holben, J. Stankovic, E. Field, and K. Whitehouse. The smart thermostat: using occupancy sensors to save energy in homes. In *Proceedings of the 8th ACM SenSys*, pages 211–224, 2010.

[17] F. Schaub, B. Könings, P. Lang, B. Wiedersheim, C. Winkler, and M. Weber. PriCal: context-adaptive privacy in ambient calendar displays. In *Proceedings of ACM UbiComp*, pages 499–510, 2014.

[18] C. Schmidt and M. Parashar. A peer-to-peer approach to web service discovery. *World Wide Web*, 7(2):211–229, 2004.

[19] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura. Serendipity: enabling remote computing among intermittently connected mobile devices. In *Proceedings of ACM MobiHoc*, pages 145–154, 2012.

[20] L. Tong, Y. Li, and W. Gao. A hierarchical edge cloud architecture for mobile computing. In *Proceedings of IEEE INFOCOM*, pages 91–99. IEEE, 2016.

[21] N. Vallina-Rodriguez and J. Crowcroft. Erdos: achieving energy savings in mobile os. In *Proceedings of the sixth international workshop on MobiArch*, pages 37–42. ACM, 2011.

[22] E. J. Wang, T.-J. Lee, A. Mariakakis, M. Goel, S. Gupta, and S. N. Patel. Magnifisense: Inferring device interaction using wrist-worn passive magneto-inductive sensors. In *Proceedings of ACM UbiComp*, pages 15–26, 2015.

[23] Y.-L. Zheng, X.-R. Ding, C. C. Y. Poon, B. P. L. Lo, H. Zhang, X.-L. Zhou, G.-Z. Yang, N. Zhao, and Y.-T. Zhang. Unobtrusive sensing and wearable devices for health informatics. *Biomedical Engineering, IEEE Transactions on*, 61(5):1538–1554, 2014.

[24] F. Zhu, M. W. Mutka, and L. M. Ni. Service discovery in pervasive computing environments. *IEEE Pervasive computing*, (4):81–90, 2005.