Towards A Personal Mobile Cloud via Generic Device Interconnections

Yong Li, *Student Member, IEEE* and Wei Gao, *Member, IEEE*

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.

Index Terms—Personal mobile cloud, device interconnection, resource sharing, OS services

1 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* [15], which incorporates and interconnects all the mobile devices owned by a user via wireless links [10], [37]. These devices are then able to flexibly share system resources with each other, augmenting the mobile computing capability provided to the user in many computation-demanding applications such as mobile Virtual Reality [30], [31]. For example in Fig. 1, when being interconnected, wearables and smartphones can greatly extend their local battery lifetime by exploiting the computational power [12], [27], [29] or GPS readings [15], [55] from the nearby stronger devices such as smart vehicles. The smart vehicles, on the other hand, can also utilize the data from various body sensors at the wearables to infer the user's real-time behavior patterns, and react accordingly [26], [45], [56].

- *Yong Li is with the Department of Electrical Engineering and Computer Science, University of Tennessee, 1520 Middle Drive, KNoxville, TN 37996. E-mail: yli118@vols.utk.edu.*
- *Wei Gao is with the Department of Electrical and Computer Engineering, University of Pittsburgh, 3700 O'Hara Street, Pittsburgh, PA 15261. E-mail: weigao@pitt.edu.*

Manuscript received 24 Apr. 2018; revised 10 Dec. 2018; accepted 5 June 2019. Date of publication 12 June 2019; date of current version 3 Nov. 2020. (Corresponding author: Yong Li.)
Digital Object Identifier no. 10.1109/TMC.2019.2922289

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 [38], [59] or a specific type of shared hardware [9], [47]. 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

Fig. 1. Interconnecting multiple mobile devices.

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 [6] 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 [14], [21].

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, 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 expect this framework to dramatically transform the way mobile computing systems are being designed and operated, so as to make significant contributions to the practical realization of personal mobile clouds. In particular, our framework retains the process of accessing remote system resources to be completely transparent from user applications, which access these resources via the corresponding service handle provided by the local OS. As a result, the large population of existing mobile systems can be enhanced and adopted towards building a personal mobile cloud, without any modification or additional efforts of software redevelopment. Instead of tackling with the details of individual hardware implementations and driver operations, developers are able to efficiently interconnect multiple remote system components through generic configurations at their local devices. The scalability and extensibility of developing the personal mobile cloud, therefore, are ensured by operating remote system resources in the same way as they are locally operated.

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 2 we provide a high-level overview about our motivation and designs. Sections 3 and 4 present the details of our resource sharing framework and application interface. Section 5 describes how we support sharing multimedia resources between mobile devices. Section 6 presents our implementations over various mobile platforms, and Section 7 presents our evaluation results based on these implementations. Section 8 discusses the related work. Finally, Section 9 discusses and Section 10 concludes the paper.

2 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.

2.1 Layered Architecture of Mobile OS

Due to the limited resources on mobile devices, mobile OSes usually adopt a hierarchical architecture, so as to isolate user applications from low-layer system implementations. Such isolation leads to more efficient system resource management, and 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.

2.2 Limitation of the Existing Schemes

Existing schemes of supporting remote resource sharing among mobile devices are implemented directly over mobile application binaries or the low-layer device drivers in the OS, and hence fail to provide generic and adaptive resource sharing in mobile systems. More specifically, despite the similarity of different applications in the way they interconnect

mobile systems and share resources between these systems, the processes of developing these diverse applications are still very different and totally separated from each other. As a result, in order to remotely access various mobile system resources, developers have to reinvent the wheel in each application for efficient interconnection and repetitively implement application wrappers to handle the communication between heterogeneous mobile devices. For example, a typical Android application sharing the sensor data and remotely playing audio tracks could take about 300[1] and 1,100[2] LoC respectively. Such lack of generality has become the major barrier hindering further widespread of resource sharing in practical mobile scenarios.

On the other hand, resource sharing at the OS driver level requires tremendous amounts of reprogramming efforts due to the diverse hardware components being equipped on mobile devices. For example, an increasing variety of sensors (e.g., accelerometer, gyroscope, compass, etc.) are nowadays available at mobile devices. To overcome the complexity of operating individual sensor drivers, mobile OSes provide a unified system service to access all the on-board sensors. However, existing schemes on OS-level resource sharing fail to exploit such unified interface and has to program the sharing functions for each driver. Doing so adds thousands of LoC to share all sensors between remote mobile systems [6].

Furthermore, such repetitive programming efforts can also be a result of customized SoC designs and subsequent incompatible hardware drivers between manufacturers. Resource sharing in this case, hence, needs to reprogram the driver implementations to make them consistent between devices, which takes about 350 and 1,000 LoC for a typical accelerometer[3] and speaker[4] driver respectively. Note that such reprogramming has to be done for every pair of mobile devices with different driver implementations.

Another key drawback of supporting remote resource sharing at the low level of OS drivers is that the runtime execution patterns of mobile applications are generally ignored, resulting in large amounts of wasteful data transmission. More specifically, low-layer driver operations synchronize resource data between mobile systems whenever the hardware status changes. Such frequency of change, however, is usually much faster than the frequency of applications' actual resource requests. For example, each user application usually specifies its own interval of requesting for location data, despite that the GPS device in Android reports location information in every second. As a result, sharing the GPS device for a remote Google GMS and Facebook application would waste 80 and 98 percent of data transmission, which request GPS data for every 5 and 60 seconds, respectively.

2.3 Our Motivation

We support generic resource sharing between remote mobile systems based on the aforementioned 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 system 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. Such transparency, therefore, enables the large amount of existing mobile applications to access remote resources without any reprogramming efforts. For example, game developers can easily project their game screens to external large displays by accessing these external displays in the same way as they operate the LCD screen of the local device. Similarly, developers of activity recognition systems will have much better flexibility in their designs, by assuming various types of body sensory data that will be opportunistically available from nearby human users.

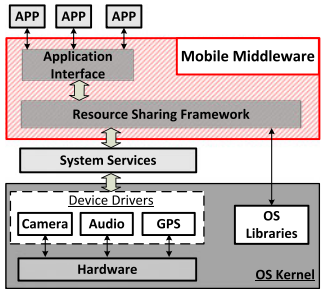


Fig. 2. The big picture of interconnecting heterogeneous mobile devices.

2.4 The Big Picture

As shown in Fig. 2, 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 system or the remote system, the service is operated through the same way.

The *Resource Sharing Framework* interacts with the local OS and communicates with the remote system services to provide remote resource access to user applications. As shown in Fig. 2, the details of low-layer device driver implementations and hardware operations in the OS kernel are completely separated from the resource sharing framework,

1. https://github.com/laobubu/sensor-android
2. https://github.com/skyrien/SoundRemote
3. https://github.com/adafruit/Adafruit_ADXL345
4. https://github.com/ggardiles/linux-speaker-driver

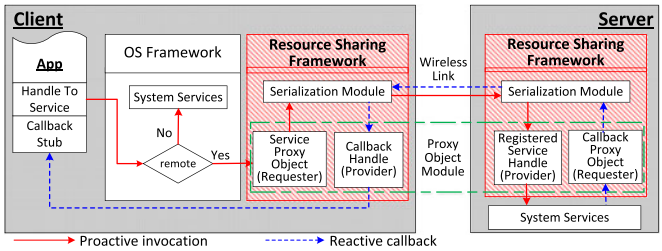


Fig. 3. Design of resource sharing framework.

and different system services are invoked in a universal manner through the same set of pre-defined APIs. Our design strategies are generic and applicable to system services in various mainstream mobile OSes (e.g., Android, iOS), which are implemented in different programming languages and system architectures.

3 Resource Sharing Framework

Our design of the resource sharing framework is shown in Fig. 3. In general, our framework intercepts the requests of resource access generated from local mobile applications, and forwards these requests to another remote mobile system 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 system 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. More specifically, this module manages two types of objects: *Provider* and *Requester*. A provider is an object which implements the programming interface, and the destination to which the remote method invocation is redirected. This object is registered to the proxy object module to enable itself for remote method invocation. On the other hand, a requester is a local proxy to the remote provider and forwards the method invocation request to the remote system. Whenever a service method invocation is issued on a proxy object requester by a mobile application, it will be redirected to the remote provider.

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 ubiquitous resource access between mobile systems 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.

3.1 Invocation of Remote System Services

In this section, we describe the design details of our resource sharing framework which supports remote invocation of both Java-based and native system services. As we mentioned above, both types of system services can be remotely invoked via both proactive invocation and reactive callback.

3.1.1 Java-Based System 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* to minimize the programming efforts 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.

In particular, serialization and deserialization require recursive traversal of a memory object since its field may also be an object. We organize a memory object as a tree-based structure, with the object itself as the root and its fields as the children. As a result, we are able to traverse and reconstruct the contents of the memory object using breadth-first search over the object tree. Moreover, when we traverse the fields of a memory object, we get or set the value of a field if it is a primitive type. Otherwise the field is an object reference type, and we traverse the content of this field recursively upon all of its siblings are processed.

On the other hand, the most straightforward solution to the proxy object module is to directly use Java's built-in dynamic proxy to an interface, so as to utilize the existing Java reflection mechanism to instantiate the proxy object. However, this simple method has fundamental limitations and cannot be used in Android because user applications in Android have to access system services through the Android IPC binder mechanism, where user applications and system services serve as the binder proxy and binder stub respectively, as shown in Fig. 4. This mechanism requires that any binder stub that can respond to an invocation from the binder kernel driver must be a subclass of Binder class. Therefore, the proxy object has to be inherited from Binder class as well, so as to be registered to the binder kernel driver to correctly intercept applications' invocation to the local service binder stub.

Our approach to addressing this challenge is to 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

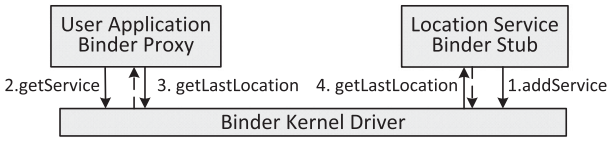


Fig. 4. Android binder mechanism with the example of location service.

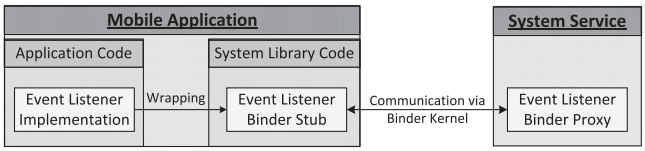


Fig. 5. Resource sharing through callbacks.

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.

In addition to the proactive invocation in which the applications invoke remote system services initiatively, 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 Fig. 5, 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 Fig. 3, 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.

3.1.2 Native System 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 manually 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. Taking the Android sensor service as an example in Fig. 6, we implement a requester and a provider which interact

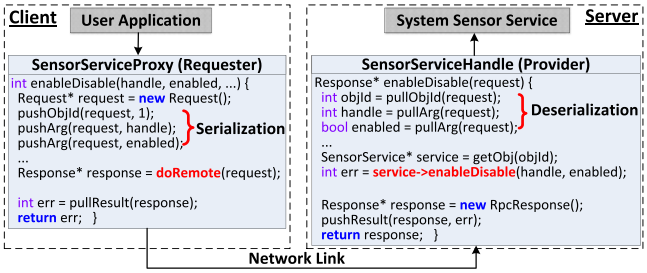


Fig. 6. Modification of native service codes with the sensor service as an example.

TABLE 1
LoC to Realize Remote Invocation

Service	Sensor Service	Audio Service	Camera Service
Serialization	63	114	137
Invocation	51	187	109

with the user application and system service, respectively. The requester serializes the arguments, invokes the remote service method and deserializes to get the invocation result. The provider deserializes to get arguments, invokes the local sensor service to enable the sensors and serializes the invocation result. These source codes are then compiled along with other OS executables.

We have quantitatively measured the amount of such manual modifications, in terms of the LoC to implement the subclasses of system services for remote invocation. As shown in Table 1, our framework incurs little engineering overhead that is less than 300 LoC for each system service.

Note that, the methodology of such manual modification is generic and applicable to all other OS services. Being different from existing schemes which share system resources over the device drivers and have to reprogram the driver implementations for each individual mobile system, our modifications are one-time efforts for each system service 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 6.

3.1.3 Supporting Concurrent Resource Sharing

Sharing Multiple Resources Concurrently. Different types of resources may be shared between two mobile devices simultaneously, resulting in concurrent invocations of multiple service methods. To support such concurrent resource sharing, each IPC provider must listen to the correct invocation event. Otherwise, invoking a wrong provider object will cause severe run-time errors.

Our approach to ensuring the invocation consistency between the provider and the requester is an identifier mapping scheme. An indexing table is maintained in the proxy object module which contains the mapping between an identifier and the provider. When a provider registers itself to proxy object module, a mapping entry will be added into the indexing table. Additionally, when a requester issues a request to remote provider, an identifier will be included in the request message specifying the destined provider. Proxy object module in the other endpoint will look up the provider

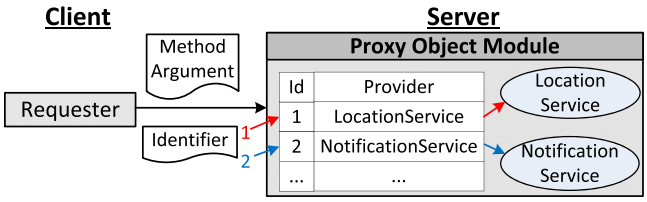


Fig. 7. Identifier mapping in the proxy object module.

according to the received identifier and make method invocation onto that provider. For example in Fig. 7, the location service and notification service register themselves to the proxy object module with identifier 1 and 2, respectively. When the requester at the sharing client issues a method invocation with identifier 1, the proxy object module at the sharing server will index the mapping table and forward the method invocation to location service. In proactive invocation, the providers are system services, and we assign an identifier to each system service in advance and keep them consistent between the sharing server and client, since the list of system services is known beforehand. On the other hand, the reactive callbacks require a dynamic identifier allocation scheme because the emergence of event listener objects is unpredictable and dependent to the run-time execution pattern of mobile applications.

Concurrent Access to the Same Resource. Local mobile applications and multiple sharing clients may access a particular resource at the same time. For system services that inherently support multiple local applications such as the location service and sensor service, concurrent access can be easily achieved by the aforementioned identifier mapping scheme, where each sharing client is managed in the same way as a local application. In contrast, the multimedia system services are designed to be exclusively accessed by one mobile application, and concurrent access to such resources is prohibited. When a new sharing client needs to access the multimedia resource that is already being occupied, our framework allows the mobile user to decide either to wait or to preempt the resource access, through the configurations in the Application Interface component (see Section 4).

A Mobile Device as Both the Sharing Server and Client. Although our resource sharing is based on the client-server structure, our framework allows a mobile device to freely act as a sharing server and a sharing client at the same time. More specifically, our framework devises separate daemon threads to represent the sharing server and client for remote interaction. The daemon thread as a sharing server handles the remote request for resource access and interacts with the corresponding system service to provide the resource data. On the other hand, the daemon thread as a sharing client connects to the sharing server and creates the corresponding service proxy for remote invocation of system services. During mobile system startup, our framework reads the user configurations and starts the corresponding daemon threads.

3.2 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.

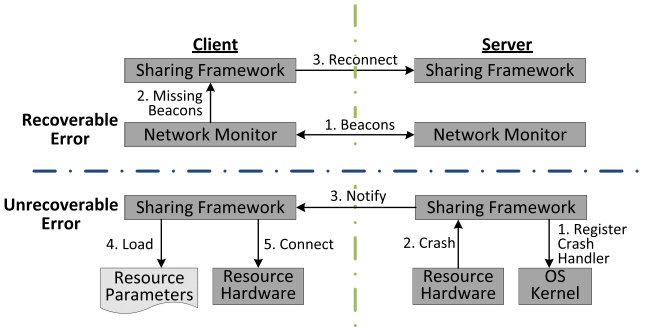


Fig. 8. Error handling in the sharing framework.

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.

3.3 Error Handling

Our proposed resource sharing framework needs to ensure correct application execution even when unexpected system errors occur and compromise remote resource access. As shown in Fig. 8, we divide the possible system errors into two categories and take different strategies accordingly. First, for system errors that can be recovered (e.g., the unstable network connection), our framework reconnects to the remote sharing server and resumes to utilize the remote resource after the error recovery. More specifically, our framework at the client sends a beacon periodically to the server. If such beacon expires, our framework at the client initiates a reconnection to the server.

On the other hand, for the unrecoverable system errors such as the resource runtime faults and unavailability, our framework opts to fall back to local resource access. More specifically, the sharing server listens to the runtime errors during resource access and registers a signal handler to the OS kernel, which would notify the sharing client before aborting the resource sharing. Meanwhile, during the invocation of remote system services, our framework logs the parameters being used by mobile applications to access the remote resource. When an unrecoverable system error is notified at the client, the framework invokes the local system services with the recorded parameters and returns local resource data to the mobile application.

4 Application Interface

Our design of the Application Interface is shown in Fig. 9. Principally, whenever a user application requests to access

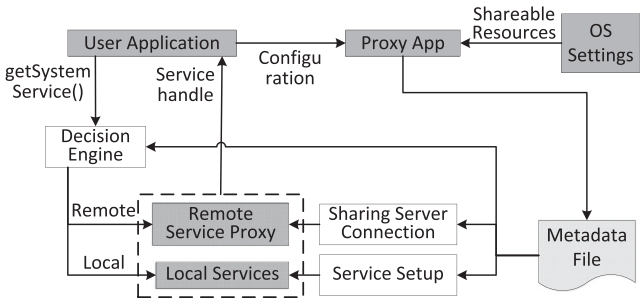


Fig. 9. Design of application interface.

a system service, the Application Interface intercepts this request and returns a handle to the appropriate system 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 systems is completely transparent to user applications.

Migration decisions of resource access and the subsequent 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 to migrate the 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 runtime to ensure correct service invocations.

Through the development of this proxy application, our design allows a user application to configure its migration decisions of system resource access in three ways. First, we allow developers to distribute their configurations along with the application 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., [49], [60]) 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.

5 Supporting Multimedia Operations

Multimedia resources are crucial components in mobile devices. For example, we may play audio tracks from a mobile device to a nearby speaker with better sound quality, or access the surrounding camera surveillance system from the mobile device for better understanding of the nearby environment. Operations over multimedia resources, however, 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 systems. In this section, we present our design to support the access to such shared memory of multimedia services between remote mobile systems. 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.

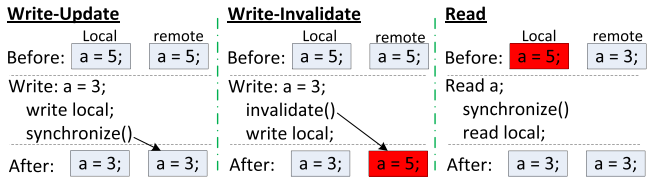


Fig. 10. Memory synchronization protocols.

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.

5.1 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 this section, we develop different techniques to operate each of these two parts, and adopt both write-update and write-invalidate protocols [43] for synchronization of shared memory. These memory synchronization protocols are illustrated in details in Fig. 10. In the write-update protocol, a synchronization message will be sent immediately after every local write, so as to always maintain the consistency between the local and the remote memory. In the write-invalidate protocol, the remote memory will retain the old value with an invalid flag, which is set by the invalidation message sent before the local write. The new value will not be synchronized until a local read occurs. We adopt the write-update protocol to synchronize memory with frequent reads and save the overhead of invalidation messages. We adopt the write-invalidate protocol over memory with infrequent reads to maximize the efficiency of memory synchronization.

5.1.1 Content Part

To efficiently operate the content part, being different from traditional approaches of Distributed Shared Memory (DSM)

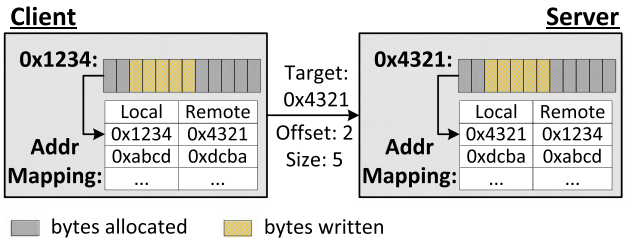


Fig. 11. Memory synchronization for content part.

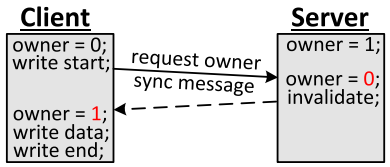


Fig. 12. Memory synchronization for control part.

[25], [50] which always share memory in fixed-size units and may result in large amounts of redundant data synchronization between mobile systems, 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 Fig. 11, 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.

Primarily, we adopt the write-update protocol for synchronizing the content part. However, write-invalidate protocol can also be adopted by maintaining a list of invalid memory fields. More specifically, during a write operation, the set of shared memory addresses being written is marked as invalid and added the list maintained at the remote system. When the other endpoint reads the memory, our framework synchronizes the memory according to the maintained list.

5.1.2 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 Fig. 12, 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. In particular, the memory synchronization message is sent along with the ownership request to reduce the round trips.

Our framework supports both the write-update protocol and write-invalidate protocol for the control part. Both the applications and system services are able to choose the synchronization protocol for individual memory fields, through a generic API that registers the synchronization metadata of the control part during shared memory allocation. Such metadata is then used to instruct our framework to perform the customized memory synchronization.

5.1.3 Optimization

In practice, we observe that a write to the content part is usually followed by an update to the control part. For example, the next write position in the control part is updated after the application writes data into the content buffer. Therefore, we further optimize our memory synchronization approach and allow the system services to synchronize the content part and control part at the same time. A system service can delay the synchronization of the content part until the corresponding control part begins to synchronize. Then, the content data can be sent along with the synchronization message of the control part, reducing the additional cost of multiple round trips through wireless links.

5.2 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 is 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.

Without loss of generality, 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.

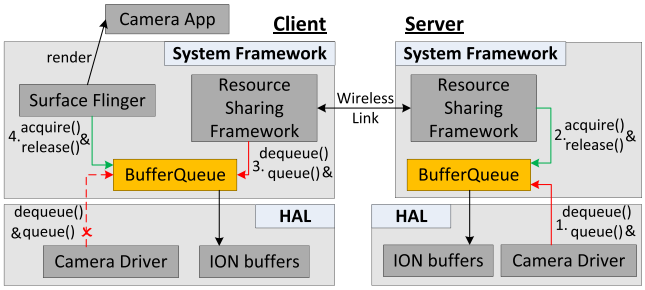


Fig. 13. Operating the graphic buffer for remote camera access.

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 Fig. 13, 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 in the camera app.

6 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.[5] 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 systems, and the implementation details are listed in Table 2. 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.

In particular, to hide the complex details of network connections and transmissions due to the heterogeneous network interfaces, our framework deploys a network manager module to provide unified high-level interface for network communication. As a result, our implementation supports both WiFi and Bluetooth for interconnecting mobile devices, as long as both endpoints possess the same network interface.

6.1 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 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.

TABLE 2
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

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.

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.

6.2 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 without any 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 mobile platforms.

Deployment of our framework on smartphones and tablets is trivial since they are directly supported by Android

5. https://github.com/crittercism/dexmaker/

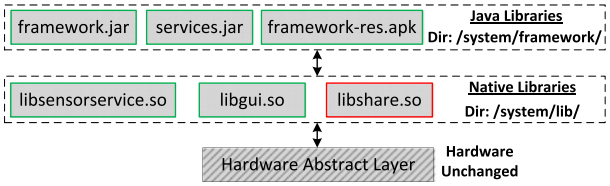


Fig. 14. Porting sensor service libraries in Android wear.

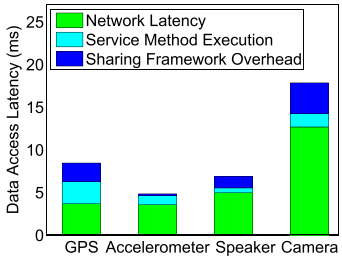


Fig. 15. Latency of accessing shared resources.

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 are not complete yet. Therefore, 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 damage the 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 Fig. 14, 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 codes of these system services are also available in AOSP and the service libraries are dynamically linked and loaded.

7 Performance Evaluation

In this section, we evaluate the performance of our proposed designs on sharing resources between remote mobile systems. 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 systems, and demonstrate that resources at remote mobile systems can be efficiently accessed without consuming significant amounts of energy. Last, we measure the communication overhead of sharing different types of resources, and report the amount of wireless network bandwidth that is required to support resource sharing in our framework. Our experiments are performed by sharing GPS, accelerometer sensor, speaker and camera between mobile devices.

7.1 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 40 Mbps 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[6] to gather the real-time information about the devices' power consumption.

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.

7.2 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 3,000 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 Fig. 15. 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 6.1, 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

6. https://www.msoon.com/LabEquipment/PowerMonitor/

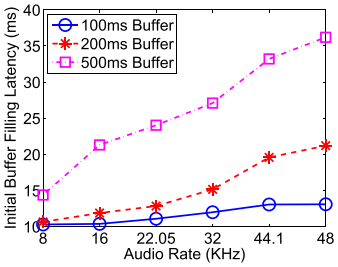


Fig. 16. Latency of initial audio buffer filling.

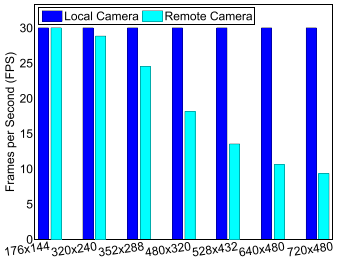


Fig. 17. Client FPS for real-time camera preview.

length of audio segments with respect to the audio rate. Each experiment plays 20 audio tracks. The experimental results are shown in Fig. 16. We can see that the initial latency of buffer filling increases along with the buffer size, but is efficiently controlled within 40 ms 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 2,000 frames. From the experimental results in Fig. 17, 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 through the wireless link. 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.

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

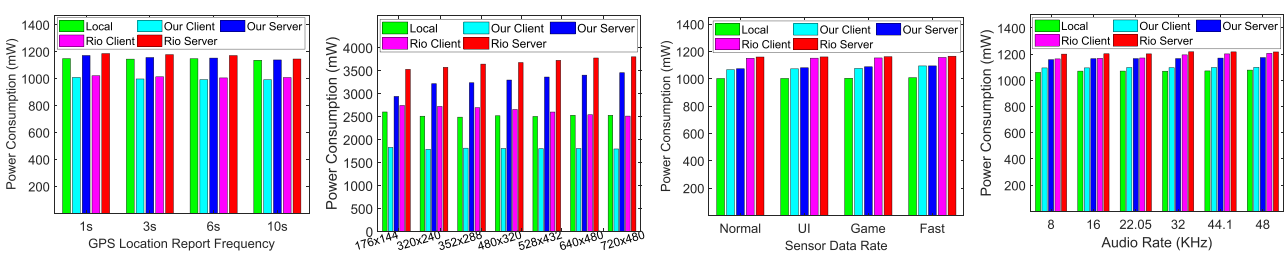
The experimental results for sharing the GPS and camera are shown in Figs. 18a and 18b respectively. In contrast to Rio [6] which consumes a tremendous amount of additional energy for resource sharing between mobile system, our framework reduces the power consumption by 13 and 29 percent respectively, 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 Figs. 18c and 18d respectively. From the figure we can see that the client consumes a small amount of extra power (7 and 4 percent 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.

7.4 Wireless Communication Overhead

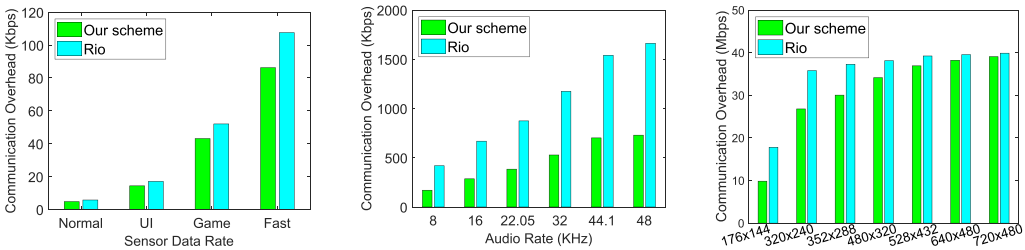
In this section, we evaluate the wireless communication overhead caused by the remote resource sharing in our framework, by measuring the average amount of data being transmitted between the client and the server. In each experiment, we use the device for three minutes to synchronize accelerometer data,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

The experimental results for accelerometer, speaker and camera are shown in Figs. 19a, 19b and 19c, respectively. We can see from the figures that sharing sensors, audio devices and camera devices incur small, moderate and large amount of wireless data transmission, respectively. In specific, both sensors and audio devices require only less than 1 Mbps to fully support remote resource access. Therefore, the Bluetooth link or a high-throughput side channel [36]



(a) GPS sharing (b) Camera sharing (c) Sensor sharing (d) Audio sharing

Fig. 18. Power consumption for remote resource sharing.



(a) Sensor sharing (b) Audio sharing (c) Camera sharing

Fig. 19. Communication overhead for remote resource sharing.

can easily meet such bandwidth requirement. 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 [35], [53].

In order to evaluate the correlation between the network bandwidth and performance of camera sharing, we measure the average data access latency and achieved FPS for the camera preview with different network bandwidth. We use the system built-in *tc* tool to limit the maximal available bandwidth. The experimental results over 2,000 frames are shown in Fig. 20. From the figure we can see that the performance of camera sharing almost linearly scales with the network bandwidth. In specific, as shown in Fig. 20a, the preview images can be delivered to the sharing client in lower latency when the network bandwidth increases, because of the reduced transmission delay for the image data. Therefore, more preview frames can be transmitted and higher FPS is achieved on the sharing client, which reaches about 22 FPS at a resolution of 480 × 320 when the network bandwidth is 60 Mbps as shown in Fig. 20b. In particular, for the preview at a low preview resolution of 176 × 144, further increasing the network bandwidth makes no impact on the FPS, which instead is limited by the image capture rate of camera hardware.

7.5 Concurrent Resource Sharing

We further evaluate the performance of our proposed framework with concurrent resource sharing, where multiple clients simultaneously access the shared resource or one mobile client accesses multiple shared resources at the same time.

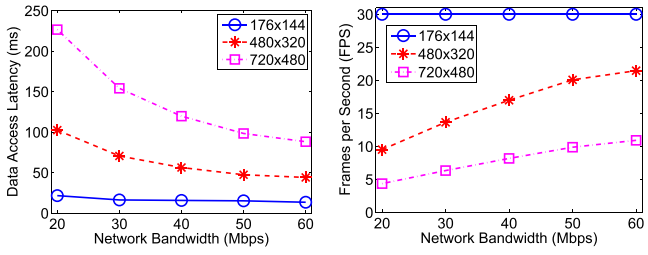
7.5.1 Concurrent Access from Multiple Clients

Multimedia Resources. In practice, the speaker and camera are designed to be exclusively accessed from one mobile application and the access is preempted if another application is trying to access the resource at the same time. Our experimental results in Fig. 21a show that starting a new sharing client with preemption takes around 131 ms and 266 ms for the speaker and camera, respectively. Such latency is dominated by the long latency of reinitializing the resource hardware and by the multiple round trips of inquiring and setting hardware configuration parameters. Therefore, as shown in Fig. 21b, the background workloads at the sharing server have only a minor impact on such preemption latency.

Sensor and GPS Resources. On the other hand, the resource data of the accelerometer and GPS can be used among multiple applications. Therefore, our experiments allow concurrent access to only the accelerometer or GPS, at an interval of 20 ms and 1 second, respectively. Each experiment is running for 3 minutes.

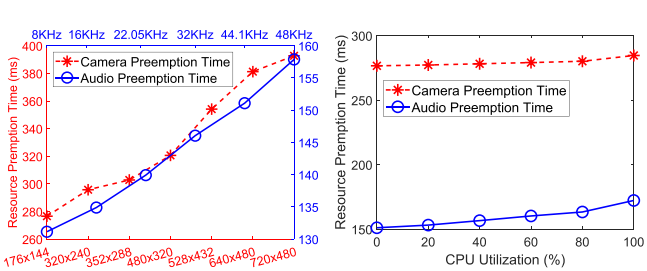
We first measure the average CPU utilization ratio at the sharing server. As shown in Fig. 22a, the CPU utilization ratio is increasing when more sharing clients are concurrently accessing the shared services. More specifically, the growth of CPU utilization ratio for GPS sharing is not linear to the client count and will slow down, because the location service is single-threaded and the remote callback for each request has to be done in a sequential order. Therefore, our resource sharing framework incurs limited computational burden to the CPU for GPS sharing on the server, which is less than 20 percent even with 50 sharing clients. On the other hand, the increase of CPU utilization ratio for sensor sharing is more significant due to the frequent transmission of sensor data, which reaches to 64 percent with 50 sharing clients. Such increase mainly results from the overhead of operating system to process the small but frequent data packets, since the sharing server needs to deliver a sensor sample to each client for every 20 ms.

We also evaluate the impact of concurrent requests on the responsiveness of the sharing server, by measuring the average time for delivering the resource data to the client. As depicted in Fig. 22b, our framework can support 10-20



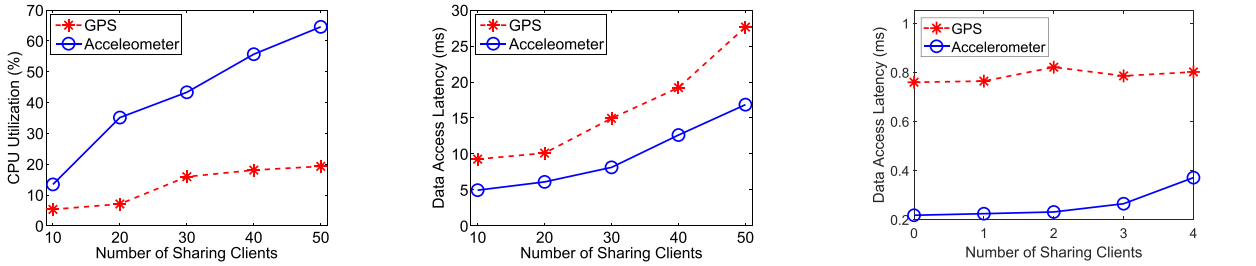
(a) Latency of previewing images (b) FPS for camera preview

Fig. 20. Performance of remote camera access with different network bandwidth.



(a) Different resource intensities (b) Different server workloads

Fig. 21. Multimedia resource sharing with preemption.



(a) CPU utilization of sharing server (b) Resource data delivery time to client (c) Local resource data delivery time at server

Fig. 22. Performance of sharing server with concurrent sharing requests.

sharing clients concurrently requesting the shared resources, without noticeably increasing the delay of resource data delivery. When the number of sharing clients further increases, the delivery time of location data increases notably, because the location data is delivered as a remote callback and the sharing server needs to coordinate network transmissions in a synchronized way to ensure the completeness and atomicity of transmitting all execution results. Therefore, the remote location callbacks at the sharing server need to wait and compete for the availability of wireless channel. On the other hand, the delay of sensor data delivery is much smaller, because sensor data is directly delivered through a TCP channel and its delay is only determined by the network traffic condition.

In practice, a sharing server could access the local resource simultaneously with concurrent sharing clients. We evaluate how the local performance of the sharing server could be affected in such cases by evaluating the average data access latency. As shown in Fig. 22c, resource sharing has no impact for the sharing server to access local GPS data because of its low frequency. On the other hand, concurrent sharing clients would increase the delay to deliver the sensor data locally for the sharing server. However, such increase is negligible because of the small overhead of the framework, which is only 0.15 ms with 4 concurrent sharing clients.

7.5.2 *Access Multiple Resources at a Client*

We evaluate the performance of our proposed framework by simultaneously accessing multiple resources on the client with different intensity levels. The specifications of resource usage at each intensity level is defined in Table 3. Particularly, we either access all resources at the same intensity level or mix the resource usage with high intensity for GPS, accelerometer and speaker but a different intensity for camera. Each experiment is running for 3 minutes.

We first measure the average CPU utilization ratio at the sharing client when only one particular or all resources are accessed remotely. As shown in result Fig. 23a, the CPU utilization ratio grows when the resource usage is more intensive because more resource data has to be processed and transmitted between endpoints. While such CPU utilization ratio is only about 36 percent even when all resources are remotely accessed with high usage intensity, which indicates that the framework can support remote resource access concurrently on the mobile without significant overhead. Also, from the figure we can see that remote camera access imposes largest computational overhead than other resources, due to the frequent and excessive amount of pixel data to be processed and transmitted.

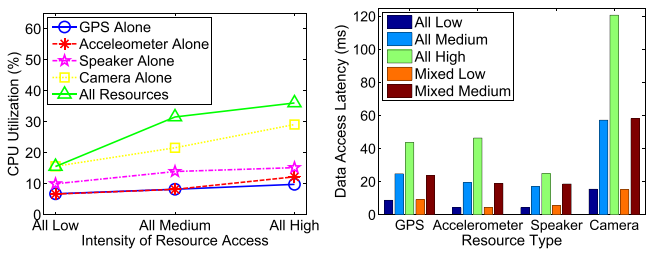
TABLE 3
Resource Access Specifications with Different Intensities

Access Intensity	GPS	Accelerometer	Speaker	Camera
All Low	6s	Normal	8 KHz	176 × 144
All Medium	3s	UI	22.05 KHz	480 × 320
All High	1s	Game	44.1 KHz	720 × 480
Mixed Low	1s	Game	44.1 KHz	176 × 144
Mixed Medium	1s	Game	44.1 KHz	480 × 320

We also evaluate how the concurrent access to multiple resources would impact the timeliness of resource data delivery. As shown in Fig. 23b, it takes longer time to deliver the resource data to the sharing client when the intensity level of resource usage increases, which increases the transmission latency due to the congestion of wireless traffic by the resource data. In particular, the data access latency is negligibly affected by 1-2 ms when the intensity level of the GPS, accelerometer and speaker is increased to high while keeping the remote camera access the same intensity. However, the latency is significantly increased by more than 10 ms when the intensity of remote camera access increases. Therefore, the large bulk of image data from camera sharing is the dominant factor of the congestion of wireless traffic.

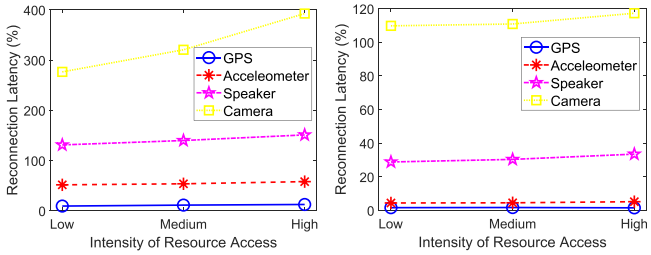
7.6 Error Handling

We have evaluated the efficiency of our framework to handle the recoverable and unrecoverable system errors by resetting the network connection and throwing runtime exceptions during remote resource access. During the experiment, we set different parameters of resource access as defined in Table 3. As shown in Fig. 24, our framework incurs low latency to reconnect to the system resource and resume the application execution. In particular, reconnecting to a remote resource takes longer time because it has to rebuild the channel for remote resource data exchange and reconfigure the hardware parameters with multiple round trips. Also, reconnecting to



(a) CPU utilization (b) Resource data delivery time

Fig. 23. Performance of sharing client with concurrent resource access.



(a) Remote reconnection (b) Local reconnection

Fig. 24. Latency to reconnect the system resources when error occurs.

the multimedia resources takes larger latency than other resources because these resources are more expensive to initialize and setup the shared buffer for data exchange.

8 Related Work

Mobile Code Offloading. Mobile offloading supports remote execution of mobile applications at the cloud to reduce the local power consumption, by utilizing the cloud to run the computationally intensive workloads. COMET [18] mirrors the application VMs by leveraging the underlying memory model of Android runtime and hence allows flexible workload migration between local devices and remote backend cloud. Mobile cloudlet [32] aggregates a set of cooperating mobile devices and pools their computing resources for services. To ensure the cloudlet performance, [16] exploits the social contact patterns of mobile users to mitigate the impact of user mobility, and [40] analyzes incentive strategies to promote user participation. Nonetheless, our major focus is to interconnect heterogeneous mobile devices and allow remote sharing of I/O hardware among them. This focus, hence, is orthogonal to the mobile code offloading which aims to utilize the remote computing power.

Application-Level Resource Sharing. Initial research efforts on resource sharing between systems focus on thin client, which allows clients to render graphical interfaces from and sends user inputs to the server. Typical examples of the thin client include the X window system [48], Citrix MetaFrame XP [3], Microsoft Remote Desktop [13], THINC [9] and VNC [46]. However, these systems are limited to solely sharing the graphical interface. Later on, applications have been developed to share various system hardware resources such as the microphone [2], webcam [1], GPS [4] and computation [17], [51], [54]. However, these applications can only share a specific type of device designated by the application developer. Sharing any other type of resource requires significant amount of engineering work. In contrast, our proposed resource sharing framework can share all types of system resources and expose the full sharing functionality to mobile applications.

The recent proliferation of ubiquitous computing aims to make computing available anytime and anywhere [23], [57], so that people can be seamlessly integrated to the environment. Extensive research efforts have been devoted and a lot of applications have been developed to allow mobile devices to connect to and share resources with nearby devices. CoMon [28] devises a cooperative ambience monitoring platform to share the sensing information among cooperators and reduce their energy consumption. Various smart home designs [20], [24], [38] adaptively adjust the room temperature configuration by sensing the human occupancy with simple sensors or predicting the user's journey time to home with GPS. Other designs [52], [59] utilize the wearable devices to acquire pervasive health information or monitor user's health condition at real-time. LocAFusion [22] and Action2Activity [34] recognize users' action more accurately with the assistance of wearable sensors. Nonetheless, these applications are suffering from the high complexity and large amounts of engineering efforts of remote resource acquisition. Instead, our work significantly reduces such overhead of application development through generic resource sharing, so that developers can have unified access to remote system resources in the same way as they use the local system resource.

OS-Level Resource Sharing. Resource sharing among distributed clients has also been supported at the OS layer. Mobile grid computing [33], [39] allows mobile devices to join the grid and share their hardware resources to other mobile devices in grid. ErdOS [55] exploits the social connections between users and opportunistically accesses resources in nearby devices so as to efficiently save local energy consumption. In order to optimize such energy saving, frameworks [11], [42] utilize a central cloud server to arbitrate the decision and destination of the shared resource destination. However, these shared resources can only be accessed through framework-specific API and thus cannot be used directly by applications.

Rio [6] is the first systematic solution to share multiple types of hardware I/O devices among mobile systems without modifying user applications. It adopts the device driver files in the OS kernel as the boundary of forwarding the I/O operations to another mobile device, and uses DSM for memory mapping operations. However, the implementation of Rio over any specific mobile system has to closely bind with its hardware drivers, and hence has to be reprogrammed in order to support different types and models of hardware. Furthermore, it also ignores the specific execution patterns of mobile applications and thus is incapable of adapting to the actual resource needs of these 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.

Remote Method Invocation. Remote method invocation [41] is a basic IPC approach between remote systems and has been widely used in distributed systems [7], [8]. Such technique has also been utilized for migrating computational workloads between mobile devices [12], [27], but has never been used for sharing system hardware and software components. Such generic sharing of mobile OS services, on the other hand, is challenging due to the unique multi-layered architecture and heterogeneous mechanisms of service invocation in mobile OS [19].

9 Discussions

9.1 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, 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 [58] is used as the intermediate data format for memory synchronization by Miracast [5]. 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.

9.2 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. Similar to [44], 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.

9.3 Limitation in Remote Resource Access

One limitation of supporting remote resource sharing through OS services is that control operations to remote system resources are not supported if they are not pre-defined in the corresponding system service APIs. For example, some hardware configurations are implemented by the *ioctl* system call in the device driver, which is meant to be set by OS libraries and inaccessible to system services. However, such limitation has no impact on correctly accessing the remote system services. Instead, it creates an additional layer of security to protect the shared resources from inappropriate operations or modifications by a malicious client application.

10 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.

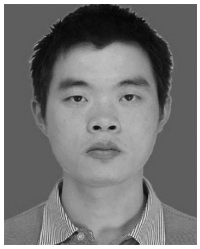
Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under grant number CNS-1617198, CNS-1812399, and CNS-1812407.

References

[1] Android IP Webcam application. 2018. [Online]. Available: https://play.google.com/store/apps/details?id=com.pas.webcam
[2] Android Wi-Fi Microphone application. 2018. [Online]. Available: https://play.google.com/store/apps/details?id=com.senstic.PocketAudioMicrophone
[3] Citrix Metaframe. 2019. [Online]. Available: https://www.citrix.com/
[4] Glympse - Share GPS location. 2018. [Online]. Available: https://play.google.com/store/apps/details?id=com.glympse.android.glympse
[5] Miracast. 2018. [Online]. Available: http://www.wi-fi.org/discover-wi-fi/wi-fi-certified-miracast
[6] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A system solution for sharing I/O between mobile systems," in *Proc. ACM Annu. Int. Conf. Mobile Syst. Appl. Services*, 2014, pp. 343–343.
[7] P. Angin and B. Bhargava, "An agent-based optimization framework for mobile-cloud computing," *J. Wireless Mobile Netw. Ubiquitous Comput. Depend. Appl.*, vol. 4, no. 2, pp. 1–17, 2013.
[8] P. Bahl, R. Y. Han, L. E. Li, and M. Satyanarayanan, "Advancing the state of mobile cloud computing," in *Proc. 3rd ACM Workshop Mobile Cloud Comput. Services*, 2012, pp. 21–28.
[9] R. A. Baratto, L. N. Kim, and J. Nieh, "THINC: A virtual display architecture for thin-client computing," *ACM SIGOPS Operating Syst. Rev.*, vol. 39, no. 5, pp. 277–290, 2005.
[10] R. Chen and W. Gao, "Enabling cross-technology coexistence for extremely weak wireless devices," in *Proc. IEEE INFOCOM*, 2019, pp. 253–261.
[11] L. Chunlin and L. LaYuan, "Cost and energy aware service provisioning for mobile client in cloud computing environment," *J. Supercomput.*, vol. 71, no. 4, pp. 1196–1223, 2015.
[12] 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 *Proc. ACM Annu. Int. Conf. Mobile Syst. Appl. Services*, 2010, pp. 49–62.
[13] B. Cumberland, G. Carius, and A. Muir, *Microsoft Windows NT Terminal Server Edition Technical Reference: Technical Reference*. Redmond, WA, USA: Microsoft Press, 1999.
[14] 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 Trans. Comput.-Human Interaction*, vol. 16, no. 1, 2009, Art. no. 3.
[15] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Comput. Syst.*, vol. 29, no. 1, pp. 84–106, 2013.
[16] H. Flores, R. Sharma, D. Ferreira, V. Kostakos, J. Manner, S. Tarkoma, P. Hui, and Y. Li, "Social-aware hybrid mobile offloading," *Pervasive Mobile Comput.*, vol. 36, pp. 25–43, 2017.
[17] W. Gao, Y. Li, H. Lu, T. Wang, and C. Liu, "On exploiting dynamic execution patterns for workload offloading in mobile cloud applications," in *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, 2014, pp. 1–12.
[18] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 93–106.
[19] T.-M. Gronli, J. Hansen, G. Ghinea, and M. Younas, "Mobile application platform heterogeneity: Android vs windows phone vs iOS vs Firefox OS," in *Proc. IEEE 28th Int. Conf. Adv. Inf. Netw. Appl.*, 2014, pp. 635–641.
[20] M. Gupta, S. S. Intille, and K. Larson, "Adding GPS-control to traditional thermostats: An exploration of potential energy savings and design challenges," in *Proc. Int. Conf. Pervasive Comput.*, 2009, pp. 95–114.
[21] P. Hamilton and D. J. Wigdor, "Conductor: Enabling and understanding cross-device interaction," in *Proc. ACM SIGCHI Conf. Human Factors Comput. Syst.*, 2014, pp. 2773–2782.
[22] 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 *Proc. ACM Int. Symp. Wearable Comput.*, 2014, pp. 99–106.
[23] J. Hightower and G. Borriello, "Location systems for ubiquitous computing," *Comput.*, vol. 8, pp. 57–66, 2001.
[24] C.-C. J. Huang, R. Yang, and M. W. Newman, "The potential and challenges of inferring thermal comfort at home using commodity sensors," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2015, pp. 1089–1100.

[25] A. Judge, P. Nixon, V. Cahill, B. Tangney, and S. Weber, "Overview of distributed shared memory," Tech. Rep., Trinity College Dublin, 1998.
[26] S.-J. Jung, H.-S. Shin, and W.-Y. Chung, "Driver fatigue and drowsiness monitoring system with embedded electrocardiogram sensor on steering wheel," *IET Intell. Transport Syst.*, vol. 8, no. 1, pp. 43–50, 2014.
[27] 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 *Proc. IEEE INFOCOM*, 2012, pp. 945–953.
[28] Y. Lee, Y. Ju, C. Min, S. Kang, I. Hwang, and J. Song, "CoMon: Cooperative ambience monitoring platform with continuity and benefit awareness," in *Proc. 10th Int. Conf. Mobile Syst. Appl. Services*, 2012, pp. 43–56.
[29] Y. Li and W. Gao, "Code offload with least context migration in the mobile cloud," in *Proc. IEEE INFOCOM*, 2015, pp. 1876–1884.
[30] Y. Li and W. Gao, "MUVR: Supporting multi-user mobile virtual reality with resource constrained edge cloud," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2018, pp. 1–16.
[31] Y. Li and W. Gao, "DeltaVR: Achieving high-performance mobile VR dynamics through pixel reuse," in *Proc. 18th Int. Conf. Inf. Process. Sensor Netw.*, 2019, pp. 13–24.
[32] Y. Li and W. Wang, "Can mobile cloudlets support mobile applications?" in *Proc. IEEE INFOCOM*, 2014, pp. 1060–1068.
[33] A. Litke, D. Skoutas, and T. Varvarigou, "Mobile grid computing: Changes and challenges of resource management in a mobile grid environment," in *Proc. Int. Conf. Practical Aspects Knowl. Manage.*, 2004, pp. 1–7.
[34] Y. Liu, L. Nie, L. Han, L. Zhang, and D. S. Rosenblum, "Action2Activity: Recognizing complex activities from sensor data," in *Proc. 24th Int. Conf. Artif. Intell.*, 2015, pp. 1617–1623.
[35] H. Lu and W. Gao, "Scheduling dynamic wireless networks with limited operations," in *Proc. IEEE 24th Int. Conf. Netw. Protocols*, 2016, pp. 1–10.
[36] H. Lu and W. Gao, "Supporting real-time wireless traffic through a high-throughput side channel," in *Proc. ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, 2016, pp. 311–320.
[37] H. Lu and W. Gao, "Continuous wireless link rates for internet of things," in *Proc. 17th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw.*, 2018, pp. 48–59.
[38] 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 *Proc. 8th ACM Conf. Embedded Netw. Sensor Syst.*, 2010, pp. 211–224.
[39] L. W. McKnight, J. Howison, and S. Bradner, "Guest editors' introduction: Wireless grids–distributed resource sharing by mobile, nomadic, and fixed devices," *IEEE Internet Comput.*, vol. 8, no. 4, pp. 24–31, Jul./Aug. 2004.
[40] E. Miluzzo, R. Cáceres, and Y.-F. Chen, "Vision: mClouds-computing on clouds of mobile devices," in *Proc. 3rd ACM Workshop Mobile Cloud Comput. Services*, 2012, pp. 9–14.
[41] B. J. Nelson, "Remote procedure call," Ph.D. thesis, Carnegie Mellon Univ., 1981.
[42] T. Nishio, R. Shinkuma, T. Takahashi, and N. B. Mandayam, "Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud," in *Proc. 1st Int. Workshop Mobile Cloud Comput. Netw.*, 2013, pp. 19–26.
[43] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Comput.*, vol. 24, no. 8, pp. 52–60, 1991.
[44] S. Oh, H. Yoo, D. R. Jeong, D. H. Bui, and I. Shin, "Mobile plus: Multi-device mobile platform for cross-device functionality sharing," in *Proc. 15th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2017, pp. 332–344.
[45] A. Pantelopoulos and N. G. Bourbakis, "A survey on wearable sensor-based systems for health monitoring and prognosis," *IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.)*, vol. 40, no. 1, pp. 1–12, Jan. 2010.
[46] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual network computing," *IEEE Internet Comput.*, vol. 2, no. 1, pp. 33–38, Jan./Feb. 1998.
[47] F. Schaub, B. Könings, P. Lang, B. Wiedersheim, C. Winkler, and M. Weber, "PriCal: Context-adaptive privacy in ambient calendar displays," in *Proc. ACM Int. Conf. Pervasive Ubiquitous Comput.*, 2014, pp. 499–510.
[48] R. W. Scheifler and J. Gettys, "The X window system," *ACM Trans. Graph.*, vol. 5, no. 2, pp. 79–109, 1986.
[49] C. Schmidt and M. Parashar, "A peer-to-peer approach to web service discovery," *World Wide Web*, vol. 7, no. 2, pp. 211–229, 2004.
[50] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," *ACM SIGOPS Operating Syst. Rev.*, vol. 29, pp. 297–306, 1994.
[51] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," in *Proc. ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, 2012, pp. 145–154.
[52] M. Terroso, R. Freitas, J. Gabriel, A. Torres Marques, and R. Simoes, "Active assistance for senior healthcare: A wearable system for fall detection," in *Proc. 8th Iberian Conf. Inf. Syst. Technol.*, 2013, pp. 1–6.
[53] L. Tong and W. Gao, "Application-aware traffic scheduling for workload offloading in mobile clouds," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[54] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *Proc. IEEE INFOCOM*, 2016, pp. 91–99.
[55] N. Vallina-Rodriguez and J. Crowcroft, "ErdOS: Achieving energy savings in mobile OS," in *Proc. 6th Int. Workshop MobiArch*, 2011, pp. 37–42.
[56] 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 *Proc. ACM Int. Conf. Pervasive Ubiquitous Comput.*, 2015, pp. 15–26.
[57] M. Weiser, "Some computer science issues in ubiquitous computing," *Commun. ACM*, vol. 36, no. 7, pp. 75–84, 1993.
[58] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H. 264/AVC video coding standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 560–576, Jul. 2003.
[59] 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," *IEEE Trans. Biomed. Eng.*, vol. 61, no. 5, pp. 1538–1554, May 2014.
[60] F. Zhu, M. W. Mutka, and L. M. Ni, "Service discovery in pervasive computing environments," *IEEE Pervasive Comput.*, vol. 4, no. 4, pp. 81–90, Oct.–Dec. 2005.



Yong Li Student Member, IEEE received the BE degree from East China Normal University, in 2008, majoring in software engineering. Currently, he is working toward the PhD degree in the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville. His research interests include mobile cloud computing and software systems.

Wei Gao Member, IEEE received the BE degree in electrical engineering from the University of Science and Technology of China, in 2005, and the PhD degree in computer science from Pennsylvania State University, in 2012. He is currently an associate professor with the Department of Electrical and Computer Engineering, University of Pittsburgh, and was an assistant professor with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville from 2012 to 2017. His research interests include wireless and mobile network systems, mobile social networks, cyber-physical systems, and pervasive and mobile computing. He received the U.S. National Science Foundation (NSF) CAREER award in 2016.