Rio: A System Solution for Sharing I/O between Mobile Systems

Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong

Rice University, Houston, TX

Abstract

Mobile systems are equipped with a diverse collection of I/O devices, including cameras, microphones, sensors, and modems. There exist many novel use cases for allowing an application on one mobile system to utilize I/O devices from another. This paper presents Rio, an I/O sharing solution that supports unmodified applications and exposes all the functionality of an I/O device for sharing. Rio's design is common to many classes of I/O devices, thus significantly reducing the engineering effort to support new I/O devices. Our implementation of Rio on Android consists of about 7100 total lines of code and supports four I/O classes with fewer than 500 class-specific lines of code. Rio also supports I/O sharing between mobile systems of different form factors, including smartphones and tablets. We show that Rio achieves performance close to that of local I/O for audio devices, sensors, and modem, but suffers noticeable performance degradation for camera due to network throughput limitations between the two systems, which is likely to be alleviated by emerging wireless standards.

Categories and Subject Descriptors

B.4.2 [**Input/Output and Data Communications**]: Input/Output Devices; C.0 [**Computer Systems Organization**]: General—System architectures; C.2.4 [**Computer-Communication Networks**]: Distributed Systems— Client/Server; K.8.0 [**Personal Computing**]: General

Keywords

I/O; Remote I/O; I/O Sharing; Mobile; Rio

1. INTRODUCTION

A user nowadays owns a variety of mobile systems, including smartphones, tablets, smart glasses, and smart watches, each equipped with a plethora of I/O devices, such as cameras, speakers, microphones, sensors, and cellular modems. There are many interesting use cases for allowing an application running on one mobile system to access I/O devices on another system, for three fundamental reasons. (*i*) Mobile systems can be in different physical locations or orientations. For example, one can control a smartphone's high-resolution camera from a tablet to more easily capture a self-portrait. (*ii*) Mobile systems can serve different users. For example, one can a play music for another user if one's smartphone can access the other system's speaker. (*iii*) Certain mobile systems have unique I/O devices due to their distinct form factors and targeted use cases. For example, a user can make a phone call from her tablet using the modem and SIM card in her smartphone.

Unsurprisingly, solutions exist for sharing various I/O devices, e.g., camera [3], speaker [4], and modem (for messaging) [11]. However, these solutions have three fundamental limitations. First, they do not support unmodified applications. For example, IP Webcam [3] and MightyText [11] do not allow existing applications to use a camera or modem remotely; they only support their own custom applications. Second, they do not expose all the functionality of an I/O device for sharing. For example, IP Webcam does not support remote configuration of all camera parameters, such as resolution. MightyText supports SMS and MMS from another device, but not phone calls. Finally, existing solutions are I/O class-specific, requiring significant engineering effort to support new I/O devices. For example, IP Webcam [3] can share the camera, but not the modem or sensors.

In this paper, we introduce Rio (*R*emote *I/O*), an I/O sharing solution for mobile systems that overcomes all three aforementioned limitations. Rio adopts a split-stack I/O sharing model, in which the I/O stack, i.e., all software layers from the application to the I/O device, is split between the two mobile systems at a certain boundary. All communications that cross this boundary are intercepted on the mobile system hosting the application and forwarded to the mobile system with the I/O device, where they are served by the rest of the I/O stack. Rio uses *device files* as its boundary of choice. Device files are used in Unix-like OSes, such as Android and iOS, to abstract many classes of I/O devices, providing an I/O class-agnostic boundary. The device file boundary supports I/O sharing for unmodified applications, as it is transparent to the application layer. It also exposes the full functionality of each I/O device to other mobile systems by allowing processes in one system to directly communicate with the device drivers in another. Rio is not the first system to exploit the device file boundary; our previous work [22] uses device files as the boundary for

MobiSys'14, June 16–19, 2014, Bretton Woods, New Hampshire, USA.

http://dx.doi.org/10.1145/2594368.2594370.

I/O virtualization inside a single system. However, sharing I/O devices between two physically separate systems engenders a different set of challenges regarding how to properly exploit this boundary, as elaborated below.

The design and implementation of Rio must address the following fundamental challenges that stem from the I/O stack being split across two systems. (*i*) A process may issue operations on a device file that require the driver to operate on the process memory. With I/O sharing, however, the process and the driver reside in two different mobile systems with separate physical memories. In Rio, we support cross-system memory mapping using a distributed shared memory (DSM) design that supports access to shared pages by the process, the driver, and the I/O device (through DMA). We also support cross-system memory copying with collaboration from both systems. (*ii*) Mobile systems typically communicate through a wireless connection that has a high round-trip latency compared to the latency between a process and driver within the same system. To address this challenge, we reduce the number of round trips between the systems due to file operations, memory operations, or DSM coherence messages. (*iii*) The connection between mobile systems can break at any time due to mobility or reliability issues. This can cause undesirable side-effects in the OSes of all involved systems. We address this problem by properly cleaning up the residuals of a remote I/O connection upon disconnection, switching to a local I/O device of the same class, if present, or otherwise returning appropriate error messages to the applications.

We present a prototype implementation of Rio for Android systems. Our implementation supports four important I/O classes: camera, audio devices such as speaker and microphone, sensors such as accelerometer, and cellular modem (for phone calls and SMS). It consists of about 7100 Lines of Code (LoC), of which less than 500 are specific to I/O classes. It also supports I/O sharing between heterogeneous mobile systems, including tablets and smartphones. See [16] for a video demo of Rio.

We evaluate Rio on Galaxy Nexus smartphones and show that it supports existing applications, allows remote access to all I/O device functionality, requires low engineering effort to support different I/O devices, and achieves performance close to that of local I/O for audio devices, sensors, and modem, but suffers noticeable performance degradation for camera sharing due to Wi-Fi throughput limitation in our setup. With emerging wireless standards supporting much higher throughput, we posit that this degradation is likely to go away in the near future. In addition, we report the throughput and power consumption for using remote I/O devices with Rio and show that throughput highly depends on the I/O device class, and that power consumption is noticeably higher than that of local devices.

2. USE CASES

We envision two categories of use cases for Rio. The first category, already tested with Rio, consists of those that simply combine Rio with existing application (§2.1). This category is the focus of this paper. However, we envision applications developed specifically with I/O sharing in mind. Obviously, such applications do not exist today because I/O sharing is not commonly available. §2.2 presents some of such use cases.

2.1 Use Cases Demonstrated with Rio

Multi-system photography: With Rio, one can use a camera application on one mobile system to take a photo with the camera on another system. This capability can be handy in various scenarios, especially when taking self-portraits, as it decouples the camera hardware from the camera viewfinder, capture button, and settings. Several existing applications try to assist the user in taking self-portraits using voice recognition, audio guidance, or face detection [5]. However, Rio has the advantage in that the user can see the camera viewfinder up close, comfortably configure the camera settings, and press the capture button whenever ready, even if the physical camera is dozens of feet away. Alternatively, one can use the front camera, which typically has lower quality than the rear-facing one.

Multi-system gaming: Many mobile games require the user to physically maneuver the mobile system for control. Tablets provide a large screen for gaming but are bulky to physically maneuver. Moreover, maneuvers like tilting make it hard for the user to concentrate on the content of the display. With Rio, a second mobile system, e.g., a smartphone, can be used for physical maneuvers while the tablet running the game remains stationary.

One SIM card, many systems: Despite many efforts [1], users are still tied to a single SIM card for phone calls or SMS, mainly because the SIM card is associated with a unique number. With Rio, the user can make and receive phone calls and SMS from any of her mobile systems using the modem and SIM card in her smartphone. For example, if a user forgets her smartphone at home, she can still receive phone calls on her tablet at work.

Music sharing: A user might want to allow a friend to listen to some music via a music subscription application on her smartphone. With Rio, the user can simply play the music on her friend's smartphone speaker.

Multi-system video conferencing: When a user is video conferencing on her tablet, she can use the speaker or microphone on her smartphone and move them closer to her mouth for better audio quality in a noisy environment. Or she can use the camera on her smart glasses as an external camera for the tablet to provide a different viewpoint.

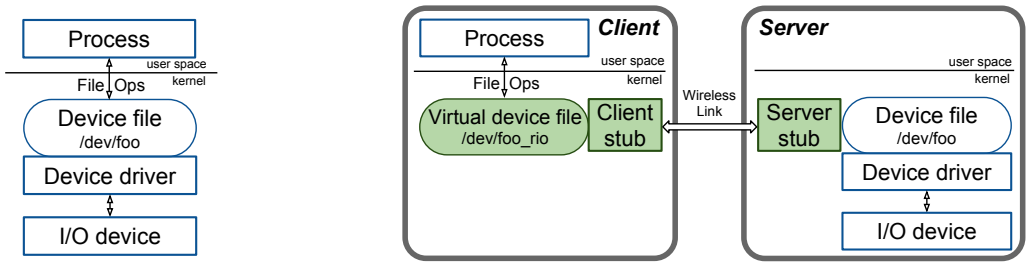
2.2 Future Use Cases of Rio

With Rio, new applications can be developed to use the I/O devices available on another system.

Multi-user gaming: The multi-system gaming use case explained in the previous subsection combined with modifications to the application can enable novel forms of multi-user gaming across mobile systems. For example, two players can use their smartphones to wirelessly control a racing game on a single tablet in front of them. The smartphones' displays can even show in-game context menus or game controller keys, similar to those on game consoles, providing a familiar and traditional gaming experience for users.

Music sharing: If supported by the system software (e.g., audio service process in Android (§7)), a user can play the same music on her and her friend's smartphones simultaneously. With proper application support, the user can even play two different sound tracks on these two systems at the same time, much like multi-zone stereo receivers.

Multi-view video conferencing: A video conferencing application can be extended to show side-by-side video streams from the smart glasses and the tablet simultaneously. In this



(a) I/O stack in Unix-like systems

(b) Rio splits the I/O stack at the device file boundary

Figure 1: Rio splits the I/O stack at the device file boundary. The process that remotely uses the I/O device resides in the client system and interacts with a virtual device file. The actual device file, device driver, and I/O device all reside in the server system. Rio forwards file operations between the client and server. The wireless link can either be through an AP or a device-to-device connection.

fashion, the user can not only share a video stream of her face with her friend, but she can also share a stream of the scene in front of her at the same time.

Multi-camera photography: Using the cameras on multiple mobile systems, one can realize various computational photography techniques [42]. For example, one can use the cameras of her smartphone and smart glasses simultaneously to capture photos with different exposure times in order to remove motion blur [52], or to increase the temporal/spatial resolution of video by interleaving/merging frames from both cameras [41, 50]. One can even use the smartphone as an external flash for the smart glasses camera.

3. DESIGN

We describe the design of Rio, including its architecture and the guarantees it provides to mobile systems using it.

3.1 Split-Stack Architecture

Rio adopts a split-stack model for I/O sharing between mobile systems. It intercepts communications at the device file boundary in the I/O stack on one mobile system and forwards them to the other system to be executed by the rest of the I/O stack.

Unix-like OSes, such as Android and iOS, use device files to abstract many classes of I/O devices. Figure 1(a) shows the typical I/O stack in Unix-like OSes. The device driver runs in the kernel, manages the device, and exports the I/O device functions to user space processes through device files. A process in the user space then issues file operations on the device file in order to interact with the device driver. One advantage of using device files as the I/O sharing boundary is that they are common to many classes of I/O devices, reducing the engineering effort required to support various I/O classes. Moreover, such a boundary is transparent to the application layer and immediately supports existing applications. It also exposes all functionality of an I/O device to other mobile systems by allowing processes to directly communicate with the driver.

Figure 1(b) depicts how Rio splits the I/O stack. It shows two mobile systems: the *server* and the *client*. The server system has an I/O device that the client system wishes to use. Rio creates a *virtual device file* in the client that corresponds to the actual device file in the server. The virtual device file creates the illusion to the client's processes that the I/O device is present locally on the client. To use the remote I/O device, a process in the client executes file operations on the virtual device file. These file operations are intercepted by the *client stub* module, which packs the arguments of each file operation into a packet and sends it to the *server stub* module. The server stub unpacks the arguments and executes the file operation on the actual device file. It then sends back the return values of the file operation to the client stub, which returns them to the process. Note that Figure 1(b) only shows one client using a single I/O device from a single server. The design of Rio, however, allows a client to use multiple I/O devices from multiple servers. It also allows multiple clients to use an I/O device from a server. Moreover, the design allows a system to act as both a client and a server simultaneously for different devices or for different systems.

In Rio, the client process is always the initiator of communications with the server driver. This is because communications between the process and driver are always initiated by the process via a file operation. When an I/O device needs to notify a process of events, the notification is done using the `poll` file operation. To wait for an event, a process issues a blocking `poll` file operation that blocks in the kernel (and hence, in the server kernel in Rio) until the event occurs. Or, it periodically issues non-blocking `polls` to check for the occurrence of an event.

Some file operations, such as `read`, `write`, `ioctl`, and `mmap`, require the driver to operate on the process memory. `mmap` requires the driver to map some memory pages into the process address space. For this, Rio uses a DSM design that supports access to shared pages by the client process as well as the server driver and device (through DMA) (§4.1). The other three file operations often ask the driver to copy data to or from the process memory. The server stub intercepts the driver's requests for these copies and services them with collaboration from the client stub (§4.2).

3.2 Guarantees

Using I/O remotely at the device file boundary impacts three expected behaviors of file operations: reliability of connection, latency, and trust model. That is, remote I/O introduces the possibility of disconnection between the process and the driver, adds significant latency to each file operation due to wireless round trips, and allows processes and drivers in different trust domains to communicate. Therefore, Rio provides the following guarantees for the client and server.

First, to avoid undesirable side-effects in the client and

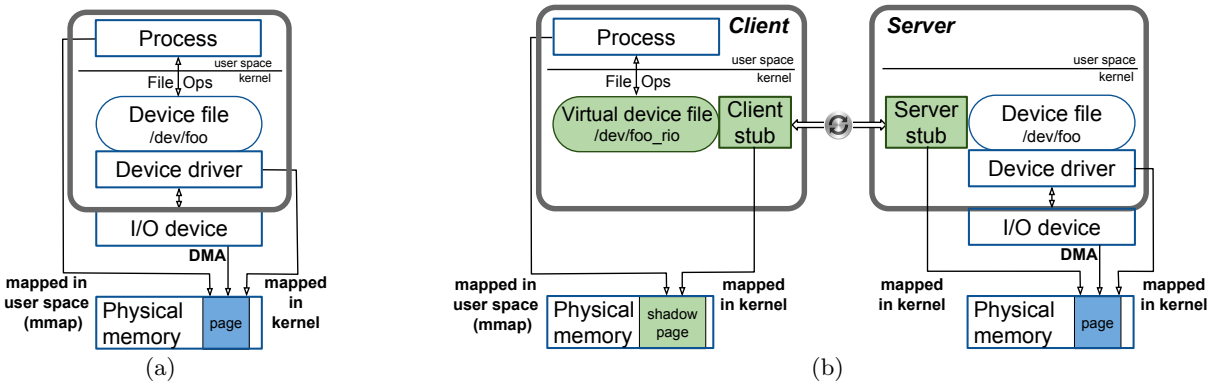


Figure 2: (a) Memory map for a local I/O device. (b) Cross-system memory map in Rio.

server resulting from an unexpected disconnection, Rio triggers a cleanup in both systems upon detecting a disconnection. Rio guarantees the server that a disconnection behaves similar to killing a local process that uses the I/O device. Rio also guarantees the client will transparently switch to a local I/O device of the same class, if possible; otherwise, Rio returns appropriate error messages to the application (§6).

Second, Rio reduces the number of round trips due to file or memory operations and DSM coherence messages (§5) in order to reduce latency and improve performance. Moreover, it guarantees that the additional latency of file operations only impacts the performance, *not the correctness*, of I/O devices. Rio can provide this guarantee because most file operations do not have a time-out threshold, but simply block until the device driver handles them. `poll` is the only file operations for which a time-out can be set by the process. In §5.4, we explain that `poll` operations used in Android for I/O devices we currently support do not use the `poll` time-out mechanism. We also explain how Rio can deal with the `poll` time-out, if used.

Finally, processes typically trust the device drivers with which they interact through the device file interface, and drivers are vulnerable to attacks by processes [15]. To maintain the same trust and security model in one mobile system, we intend the current design of Rio to be used among trusted mobile systems only. In §10, we discuss how the current design can be enhanced to maintain this guarantee while supporting I/O sharing between untrusted mobile systems.

4. CROSS-SYSTEM MEMORY SUPPORT

In order to handle file operations, the device driver often needs to operate on the process memory by executing memory operations. However, these operations pose a challenge for Rio because the process and the driver reside in different mobile systems with separate physical memories. In this section, we present our solutions.

There are three types of memory operations. The first one is `map_page`, which the driver uses to map system or device memory pages into the process address space. This memory operation is used for handling the `mmap` file operation and its supporting `page_fault` operation. Note that the kernel itself performs the corresponding `unmap_page` memory operation and not the driver. The other two types of memory operations are `copy_to_user` and `copy_from_user`, which the driver uses to copy a buffer from the kernel to the process memory and vice-versa. These two memory operations are typically used for handling `read`, `write`, and `ioctl` file operations.

4.1 Cross-System Memory Map

Cross-system memory map in Rio supports the `map_page` memory operation across two mobile systems using Distributed Shared Memory (DSM) [27, 31, 37, 47, 53] between them. At the core of Rio's DSM is a simple write-invalidate protocol, similar to [53]. The novelty of the DSM in Rio is that it can support access to the distributed shared memory pages not only by a process, but also by kernel code, such as the driver, and also by the device (through DMA).

Figure 2 illustrates the cross-system memory map in Rio. When intercepting a `map_page` operation from the server driver, the server stub notifies the client stub, which then allocates a shadow memory page in the client (corresponding to the actual memory page in the server) and maps that shadow page into the client process address space. The DSM modules in these two stubs guarantee that the process, the driver, and the device have consistent views of these pages. That is, updates to both the actual and shadow pages are consistently available to the other mobile system.

We choose a write-invalidate protocol in Rio's DSM for efficiency. Compared to update protocols that proactively propagate the updates to other systems [27], invalidate protocols do so only when the updated data is needed on the other system. This minimizes the amount of data transmitted between the client and server, and therefore minimizes the resource consumption, e.g., energy, in both systems. With the invalidate protocol, each memory page can be in one of three possible states: read-write, read-only, or invalid. Although the invalidate protocol is the default in Rio, we can also use an update protocol if it offers performance benefits.

We use 4 KB pages (small pages) as the coherence unit because it is the unit of the `map_page` memory operation, meaning the driver can map memory as small as a single small page into the process address space. When many pages are updated together, we batch them altogether to improve performance (§5.3).

To manage a client process's access to the (shadow) page, we use the page table permission bits, similar to some existing DSM solutions [37]. When the shadow page is in the read-write state, the page table grants the process full access permissions to the page, and all of the process's read

and write instructions execute natively with no extra overhead. In the read-only state, only write to these pages cause page faults, while both read and write cause page faults in the invalid state. Upon a page fault, the client stub triggers appropriate coherence messages. For a read fault, it fetches the page from the server. For a write fault, it first fetches the page if in invalid state, and then sends an invalidation message to the server.

To manage the server driver's access to the page, we use the page table permission bits for kernel memory since the driver operates in the kernel. However, unlike process memory that uses small 4 KB pages, certain regions of kernel memory, e.g., the identity-mapped region in Linux, use larger pages, e.g., 1 MB pages in ARM [23], for better TLB and memory efficiency. When the driver requests to map a portion of a large kernel page into the process address space, the server stub dynamically breaks the large kernel page into multiple small ones by destroying the old page table entries and creating new ones, similar to the technique used in K2 [38]. With this technique, the server stub can enforce different protection modes against kernel access at the granularity of small pages, rather large pages. To minimize the side-effects of using small pages in the kernel, e.g., higher TLB contention, the server stub immediately stitches the small pages back into a single large page when the pages are unmapped by the process.

To manage the server I/O device's access to the page through DMA, the server stub maintains an explicit state variable for each page, intercepts the driver's DMA request to the device, updates the state variable, and triggers appropriate coherence messages. Note that it is not possible to use page table permission bits for a device's access to the page since devices' DMA operations bypass the page tables.

Rio provides sequential consistency. This is possible for two reasons. First, the DSM module triggers coherence messages immediately upon page faults and DMA completion, and maintains the order of these messages in each system. Second, processes and drivers use file operations to coordinate their own access to mapped pages.

4.2 Cross-System Copy

Cross-system memory copy in Rio supports `copy_from_user` and `copy_to_user` memory operations between two mobile systems. We achieve this through collaboration between the server and client stubs. When intercepting a `copy_from_user` or `copy_to_user` operation from the driver, the server stub sends a request back to the client stub to perform the operation. In the case of `copy_from_user`, the client stub copies the data from the process memory and sends it back to the server stub, which copies it into the kernel buffer determined by the driver. In the case of `copy_to_user`, the server stub copies and sends the data from the kernel buffer to the client stub, which then copies it to the corresponding process memory buffer. Figure 3(b) illustrates the cross-system copy for a typical `ioctl` file operation.

When handling a file operation, the driver may execute several memory copy operations, causing that many round trips between the mobile systems because the server stub has to send a separate request per copy operation. Large numbers of round trips can degrade the I/O performance significantly. In §5.1, we explain how we reduce these round trips to only one per file operation by pre-copying the copy data in the client stub for `copy_from_user` operations, as well as batching the data of `copy_to_user` operations in the server stub.

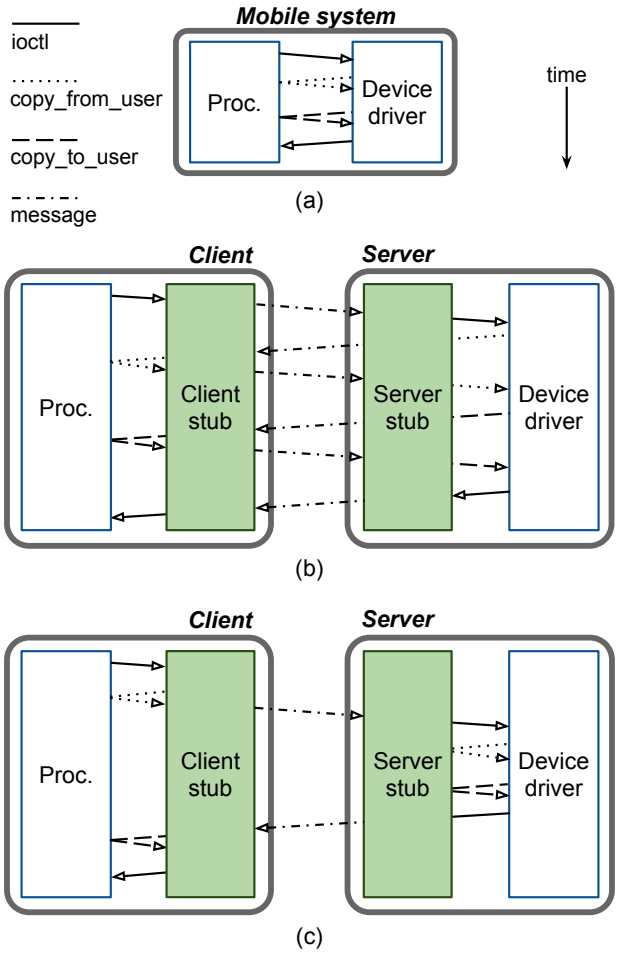


Figure 3: Typical execution of an `ioctl` file operation for (a) a local I/O device, (b) a remote I/O device with unoptimized Rio (§4.2), and (c) a remote I/O device with optimized Rio (§5.1). As the figure shows, optimized Rio reduces the number of round trips from 3 to 1. The reduction can be even more significant if the file operation requires more copy memory operations.

5. MITIGATING HIGH LATENCY

The connection between the client and the server typically has high latency. For example, Wi-Fi and Bluetooth have about 1-2 ms round-trip latency at best [51], which is significantly higher than the few microseconds of latency typical of native communications between a process and device driver (i.e., syscalls). In this section, we discuss the challenges resulting from such high latency and present our solutions to reduce its effect on I/O performance by reducing the number of round trips due to copy memory operations, file operations, and DSM coherence messages.

5.1 Round Trips due to Copies

Round trips due to `copy_from_user` and `copy_to_user` memory operations present a serious challenge to Rio's per-

formance since a single file operation may execute several copy memory operations in succession. For example, a single `ioctl` in Linux's PCM audio driver may execute four `copy_from_user` operations. To solve this problem, we use the following two techniques. (*i*) In the client stub, we determine and pre-copy all the data needed by the server driver and transmit it together with the file operation. With this technique, all `copy_from_user` requests from the driver are serviced locally inside the server. (*ii*) In the server stub, we buffer and batch all data that the driver intends to copy to the process memory and transmit it to the client along with the return values of the file operation. With this technique, all `copy_to_user` operations can be executed locally in the client. Figure 3(c) illustrates these techniques.

Pre-copying the data for driver `copy_from_user` requests requires the client stub module to determine *in advance* the addresses and sizes of the process memory data buffers needed by the driver. This is trivial for the `read` and `write` file operations, as this information is embedded in their input arguments. However, doing so for `ioctl` is non-trivial as the `ioctl` input arguments are not always descriptive enough. Many well-written drivers embed information about some simple driver memory operations in one of the `ioctl` input arguments, i.e., the `ioctl` command number. In such cases, we parse the command number in the client stub to infer the memory operations, similar to [22]. There are cases, however, that the command number does not contain all necessary information. For these cases, we use a static analysis tool from our previous work, [22], that analyses the driver's source code to extract a small part of the driver code, which can then be executed either offline or at runtime in the client stub to infer the parameters of driver memory operations. Finally, to maintain a consistent view of the process memory for the driver, Rio updates the pre-copied data in the server stub upon buffering the `copy_to_user` data if the memory locations overlap.

5.2 Round Trips due to File Operations

File operations are executed synchronously by each process thread, and therefore, each file operation needs one round trip. To optimize performance, the process should issue the minimum number of file operations possible. Changing the number of file operations is not always possible or may require substantial changes to the process source code, e.g., the I/O service code in Android (§7), which is against Rio's goal of reducing engineering effort. However, minimal changes to the process code can occasionally result in noticeable reductions in file operation issuance, justifying the engineering effort. §7.3 explains one example for Android audio devices.

5.3 Round Trips due to DSM Coherence

As mentioned in §4.1, we use 4 KB pages as the DSM coherence unit in Rio. However, when there are updates to several pages at once, such a relatively small coherence unit causes several round trips for all data to be transferred. In such cases, transmitting all updated pages together in a single round trip is much more efficient.

5.4 Dealing with Poll Time-outs

`poll` is the only file operations for which the issuing process can set a time-out. Since Rio adds noticeable latency to each file operation, it can break the semantics of `poll` if a relatively small time-out threshold is used. So far in our Android implementation, all I/O classes we support do not use `poll` time-out (i.e., the process either blocks indefinitely until the event is ready or uses non-blocking `poll`s). If `poll` is used with a time-out, the time-out value should to be adjusted for remote I/O devices. This can be done inside the kernel handler for `poll`-related syscalls, such as `select`, completely transparent to the user space. Using the heartbeat round-trip time (§6), the client stub can provide a best estimate of the additional latency that the syscall handler needs to add to the requested time-out value. Processes typically rely on the kernel to enforce the requested `poll` time-out; therefore, this approach guarantees that the process function will not break in the face of high latency. In the unlikely case that the process uses an external timer to validate its requested time-out, the process must be modified to accommodate additional latency for remote I/O devices.

6. HANDLING DISCONNECTIONS

The connection between the client and the server may be lost at any time due to mobility. If not handled properly, the disconnection can cause the following problems: render the driver unusable, block the client process indefinitely, or leak resources, e.g., memory, in the client and server OSes. When faced with a disconnection, the server and client stubs take the appropriate actions described below.

We use a time-out mechanism to detect a disconnection. At regular intervals, the client stub transmits heartbeat messages to the server stub, which immediately transmits back an acknowledgement. If the client stub does not receive the acknowledgement before a certain threshold, or the server does not hear from the client, they both trigger a disconnection event. We do not use the in-flight file operations as a heartbeat because file operations can take unpredictable amounts of time to complete in the driver. Determining the best heartbeat interval and time-out thresholds to achieve an acceptable trade-off between overhead and detection accuracy is part of future work.

For the server, network disconnection is equivalent to killing a local process that is communicating with the driver. Therefore, just as the OS cleans up the residuals of a killed process, the server stub cleans up the residuals of the disconnected client process. For each `mmap`ed area and each file descriptor, the server stub invokes the driver's `close_map` handler and `release` file operation handler respectively, in order for the driver to release the allocated resources. Finally, it releases its own bookkeeping data structures.

We take two actions in the client upon disconnection. First, we clean up the residuals of the disconnected remote I/O in the client stub, similar to the cleanup process in the server. Next, we try to make the disconnection as transparent to the application as possible. If the client has a local I/O device of the same class, we transparently switch to that local I/O device after the disconnection. If no comparable I/O device is present, we return appropriate error messages supported by the API. These actions require class-specific developments, and §7.3 explains how we achieve this for sensors. Switching to local I/O is possible for three of the I/O classes we currently support, including camera, audio, and sensors such as accelerometer. For the modem, the disconnection means that a phone call will be dropped or not initiated, or that an SMS will not be sent; all behaviors are understandable by existing applications.

Type	Total LoC	Component	LoC
Generic	6618	Server stub	2801
		Client stub	1651
		Shared between stubs	647
		DSM	1192
		Supporting Linux kernel code	327
Class-specific	498	Camera:	
		- HAL	36
		- DMA	134
		Audio device	64
		Sensor	128
		Cellular modem	136

Table 1: Rio code breakdown.

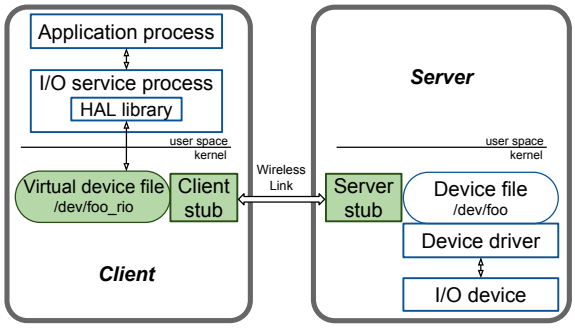


Figure 4: Rio's architecture inside an Android system. Rio forwards to the server the file operations issued by the I/O service process through HAL. Rio supports unmodified applications but requires small changes to the class-specific I/O service process and/or HAL.

7. ANDROID IMPLEMENTATION

We have implemented Rio for Android OS and ARM architecture. The implementation currently supports four classes of I/O devices: sensors (e.g., accelerometer), audio devices (e.g., microphone and speaker), camera, and modem (for phone calls and SMS). It consists of about 7100 LoC, fewer than 500 of which are I/O class-specific as shown in Table 1. We have tested the implementation on Galaxy Nexus smartphones running CyanogenMod 10.1 (Android 4.2.2) atop Linux kernel 3.0, and on a Samsung Galaxy Tab 10.1 tablet running CyanogenMod 10.1 (Android 4.2.2) with Linux kernel 3.1. The implementation can share I/O between systems of different form factors: we have demonstrated this for sharing sensors between a smartphone and a tablet.

Figure 4 shows the architecture of Rio inside an Android system. In Android, the application processes do not directly use the device files to interact with the driver. Instead, they communicate to a class-specific *I/O service process* through class-specific APIs. The I/O service process loads a *Hardware Abstraction Layer (HAL)* library in order to use the device file to interact with the device driver. Rio's device file boundary lies below the I/O service processes, forwarding its file operations to the server. As we will explain in the rest of this section, we need small modifications to the HAL or I/O service process, but no modifications to the applications are needed.

7.1 Client & Server Stubs

The client and server stubs are the two main components of Rio and constitute a large portion of Rio's implementation. Each stub has three modules. The first module supports interactions with applications and device drivers. In the client stub, this module intercepts the file operations and packs their arguments into a data structure; in the server stub, it unpacks the arguments from the data structure and invokes the file operations of the device driver. This module is about 3000 LoC and is shared with our previous work, Paradice [22]. The second module implements the communication with the other stub by serializing data structures into packets and transmitting them to the other end. Finally, the third module implements Rio's DSM, further explained in §7.2.

We use in-kernel TCP sockets for communication between the client and server stubs [10]. We use TCP to ensure that all the packets are successfully received, otherwise the device, driver, or the application might break.

To handle cross-system memory operations, the server stub intercepts the driver's kernel function calls for memory operations. This includes intercepting 7 kernel functions for `copy_to_user` and `copy_from_user` and 3 kernel functions for `map_page`. Using this technique allows us to support *unmodified drivers*.

7.2 DSM Module

Rio's DSM module is shared between the client and the server. It implements the logic of the DSM protocol, e.g., triggering and handling coherence messages. The DSM module is invoked in two cases: page faults and DMA. We instrument the kernel fault handler to invoke the DSM module when there is a page fault. Additionally, the DSM module must handle device DMA to DSM-protected pages. We monitor the driver's DMA requests to the device and invoke the DSM module upon DMA completion.

To monitor the driver's DMA requests to devices, we instrument the corresponding kernel functions. These functions are typically I/O bus-specific and will apply to all I/O devices using that I/O bus. Specialized instrumentation is needed if the driver uses non-standard interfaces. For example, the camera on the TI OMAP4 SoC inside Galaxy Nexus smartphones uses custom messages between the driver and the Imaging Subsystem (ISS) component, where the camera hardware resides [49]. We instrumented the driver responsible for communications with the ISS to monitor the DMA requests, only with 134 LoC.

When we receive a DMA completion notification for a memory buffer, we may use a DSM update protocol to immediately push the updated buffers to the client, an optional optimization. Moreover, we update the whole buffer in one round trip. These optimizations can improve performance as they minimize the number of round trips between mobile systems (§5.3); as such, we used them for camera frames.

As described in §4.1, certain regions of the kernel's address space, namely the identity-mapped region, use large 1 MB pages known as *Sections* in the ARM architecture. To split these 1 MB Sections into smaller 4 KB pages for use with our DSM module, we first walk the existing page tables to obtain a reference to the Section's first-level descriptor (a PGD entry). We then allocate a single new page that holds 512 second-level page table entries (PTEs), one for each page; altogether, these 512 PTEs reference two 1 MB Sections of virtual memory. We populate each second-level PTE with the correct page frame number and permission bits from the original Section. Finally, we change the first-level descriptor

entry to point to our new table of second-level PTEs and flush the corresponding cache and TLB entries.

7.2.1 Support for Buffer Sharing using Android ION

Android uses the ION memory management framework to allocate and share memory buffers for multimedia applications, such as those using the GPU, camera, and audio [2]. The sharing of ION buffers creates unique challenges for Rio, as demonstrated in the following example.

The camera HAL allocates buffers using ION and passes the ION buffer handles to the kernel driver, which translates them to the physical addresses of these buffers and asks the camera to copy new frames to them. Once the frames are written, the HAL is notified and forwards the ION buffer handle to the graphics framework for rendering. Now, imagine using a remote camera in Rio. The same ION buffer handles used by the camera HAL in the client need to be used by both the server kernel driver and the client graphics framework, since the camera frames from the server are rendered on the client display.

To solve this problem, we provide support for global ION buffers that can be used both inside the client and the server. We achieve this by allocating an ION buffer in the server with similar properties (e.g., size) to the one allocated in the client; we use the DSM module to keep the two buffers coherent.

7.3 Class-Specific Developments

Most of Rio's implementation is I/O class-agnostic; our current implementation only requires under 500 class-specific LoC.

Resolving naming conflicts: In case the client has an I/O device of the same class that uses device files with similar names as those used in the server, the virtual device file must assume a different name (e.g., `/dev/foo_rio` vs. `/dev/foo` in Figure 1(b)). However, the device file names are typically hard-coded in the HAL, necessitating small modifications to use a renamed virtual device file for remote I/O.

Optimizing performance: As discussed in §5.2, sometimes small changes to the I/O service code and HAL can boost the remote I/O performance significantly by reducing the number of file operations. For example, the audio HAL exchanges buffered audio segments with the driver using `ioctl`s. The HAL determines the size of the audio segment per `ioctl`. For local devices (with very low latency), these buffered segments can contain as low as 3 ms of audio, less than a single round-trip time in Rio. Therefore, we modify the HAL to use larger buffering segments for remote audio devices. Although this slightly increases the audio latency, it significantly improves the audio rate for remote devices. §8 provides measurements to quantify this trade-off. This modification only required about 30 LoC.

Support for hot-plugging and disconnection: Remote I/O devices can come and go at any time; in this sense, they behave similarly to hot-plugging/removal of local I/O devices. Small changes to the I/O service layer may be required to support hot-plugging and disconnection of remote I/O devices. For example, the Android sensor service layer opens the sensor device files (through the HAL) in the phone initialization process and only uses these file descriptors to read the sensor values. To support hot-plugging remote sensors, we modified the sensor service layer to open the virtual device files and use their file descriptors too when remote sensors are present. Upon disconnection, we switch back to using local sensors to provide application transparency.

Avoiding duplicate I/O initialization: Some HAL libraries, including sensor and cellular modem's, perform initialization of the I/O device upon system boot. However, since the I/O device is already initialized in the server, the client HAL should not attempt to initialize the I/O device. Not only this can break the I/O device, it can also break the HAL because the server device driver rejects initialization-based file operations. Therefore, the HAL must be modified in order to avoid initializing a device twice. Achieving this was trivial for the open-source sensor HAL. However, since the modem's HAL is not open-source, we had to employ a workaround that uses a second SIM card in the client to initialize its modem's HAL. It is possible to develop a small extension to the modem open-source kernel driver in order to fake the presence of the SIM card and allow the client HAL to initialize itself without a second SIM card.

Sharing modem for phone calls: Through the device file interface, Rio can initiate and receive phone calls. However, it requires further development to relay the incoming and outgoing audio for the phone call. A phone call on Android works as follows. The modem HAL uses the modem device file to initiate a call, or to receive one. Once the call is connected, with instructions from the HAL, the modem uses the speaker and microphone for incoming and outgoing audio. The modem directly uses the speaker and microphone; therefore, the audio cannot be automatically supported by Rio's use of the modem device file. To overcome this, we leveraged Rio's ability to share audio devices in order to relay the audio. There are two audio streams to be relayed. The audio from the user (who is using the client device) to the target phone (on the other side of the phone call) and the audio from the target phone to the client. For the former stream, we record the audio on the client using the client's own microphone and play it back on the server's speaker through Rio. The modem on the server then picks up the audio played on the speaker and transmits it to the target phone. For the latter stream, we record the incoming audio data on the server using server's microphone remotely from the client using Rio, and then play it back on the client's own speaker for the user. We used CyanogenMod 10.2 for relaying the audio for phone calls since CyanogenMod 10.1 does not support capturing audio during a phone call.

Currently, we can only support one audio stream at a time. The main reason for this is that supporting both streams will result in the user at the client hearing herself since the audio played on the server's speaker will be picked up by the microphone. While we currently have to manually arbitrate between the two streams, it is possible to support automatic arbitration by measuring the audio intensity on the two streams.

7.4 Sharing between Heterogeneous Systems

Because the device file boundary is common to all Android systems, Rio's design readily supports sharing between heterogeneous systems, e.g., between a smartphone and a tablet. However, the implementation has to properly deal with the HAL library of a shared I/O device because it may be specific to the I/O device or to the SoC used in the server. Our solution is to port the HAL library used in the server to the client. Such a port is easy for two reasons. First, Android's interface to the HAL for each I/O class is the same

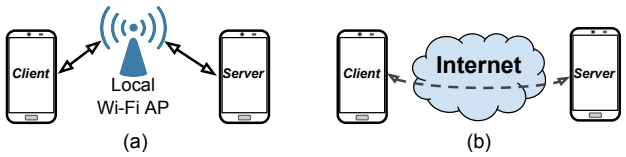


Figure 5: We use two different connections in our evaluation. In (a), the phones are connected to the same AP. This connection represents that used between mobile systems that are close to each other. In (b), the phones are connected over the Internet. This connection represents that used between mobile systems at different geographical locations.

across Android systems of different form factors. Second, all Android systems use the Linux kernel and are mostly shipped with ARM processors. For example, we managed to port the Galaxy Nexus smartphone sensors HAL library to the Samsung Galaxy Tab 10.1 tablet by compiling the smartphone's HAL inside the tablet's source tree.

8. EVALUATION

We evaluate Rio and show that it supports legacy applications, allows access to all I/O device functionality, requires low engineering effort to support different I/O devices, and achieves performance close to that of local I/O for audio devices, sensors, and modem, but exhibits performance drops for the camera due to network throughput limitations. We further discuss that future wireless standards will eliminate this performance issue. In addition, we report the throughput and power consumption for using remote I/O devices with Rio and show that throughput highly depends on the I/O device class, and that power consumption is noticeably higher than that of local devices. Finally, we demonstrate Rio's ability to handle disconnections.

All experiments are performed on two Galaxy Nexus smartphones. We use two connections of differing latencies for the experiments. The first connection (Figure 5(a)) is over a wireless LAN between mobile systems that are close to each other, e.g., both in the same room. We connect both phones to the same Wi-Fi access point. This connection has a latency with median, average, and standard deviation of 4 ms, 8.5 ms, and 16.3 ms, and a throughput of 21.9 Mbps. The second connection (Figure 5(b)) is between mobile systems at different geographical locations, one at home and one 20 miles away at work. We connect these two phones through the Internet using external IPs from commodity Internet Providers. This connection has a latency with median, average, and standard deviation of 55.2 ms 57 ms, and 20.9 ms, and a throughput of 1.2 Mbps. All reported results use the first LAN connection, unless otherwise stated.

8.1 Non-Performance Properties

First, Rio supports existing unmodified applications. We have tested Rio with both stock and third-party applications using different classes of I/O devices.

Second, unlike existing solutions, Rio exposes all functionality of remote I/O devices to the client. For example, the client system can configure every camera parameter, such as resolution, zoom, and white balance. Similarly, an application can configure the speaker with different equalizer effects.

Third, supporting new I/O devices in Rio requires low engineering effort. As shown in Table 1, we only needed 128, 64, 170, and 136 LoC to support sensors, audio devices (both speaker and microphone), camera, and the modem (for phone calls and SMS) respectively.

8.2 Performance Benchmarks

In this subsection, we measure the performance of different I/O classes in Rio and compare them to local performance. Unless otherwise stated, we repeat each experiment three times and report the average and standard deviation of measurements. The phones are rebooted after each experiment.

Audio devices: We evaluate the performance of the speaker and the microphone measuring the audio (sample) rate at different buffering sizes, and hence, different audio latency. Using larger buffering sizes reduces the interactions with the driver but increases the audio latency. Audio latency is the average time it takes a sample to reach the speaker from the process (and vice-versa for microphone), and is directly determined by the buffering size used in the HAL.

Figure 6 shows the achieved rate for different buffering sizes (and hence different latencies) for the speaker and the microphone when accessed locally or remotely through Rio. We use a minimum of 3 ms for the buffering size as it is the smallest size used for local speakers in Android in low latency mode. The figure shows that such a small buffering size degrades the audio rate in Rio. This is mainly because the HAL issues one `ioctl` for each 3 ms audio segment, but the `ioctl` takes longer than 3 ms to finish in Rio due to the network's long round-trip time. However, the figure shows that Rio is able to achieve the desired 48 kHz audio rate at slightly larger buffering sizes of 9 and 10 ms for microphone and speaker, respectively. We believe that Rio achieves acceptably low audio latency because Android uses a buffering size of 308 ms for non-low latency audio mode for speaker, and uses 22 ms for microphone (it uses 3 ms for low latency speaker).

We also measure the performance of audio devices with Rio when mobile systems are connected via the aforementioned high latency connection, e.g., for making a phone call remotely at work using a smartphone at home. Our measurements show that Rio achieves the desired 48 kHz for the microphone using buffering sizes as small as 85 ms. However, for the speaker, Rio can only achieve a maximum sampling rate of 25 kHz using a 300 ms buffer (other buffer sizes performed poorly). We believe this is because the speaker requires a higher link throughput, as demonstrated in §8.3.

Camera: We measure the performance of both a real-time streaming camera preview and that of capturing a photo. In the first case, we measure the frame rate (in frames per second) that the camera application can achieve, averaged over 1000 frames for each experiment. We ignore the first 50 frames to avoid the effects of camera initialization on performance. Figure 7(a) shows that Rio can achieve acceptable performance (i.e., >15 FPS) at low resolutions. The performance at higher resolutions is bottlenecked by network throughput between the client and server. Rio spends most of its time transmitting frames rather than file operations. However, streaming camera frames are uncompressed, requiring, for example, 612 KB of data *per frame* even for VGA (640×480) resolution, necessitating about 72 Mbps of throughput to maintain 15 FPS at this resolution.

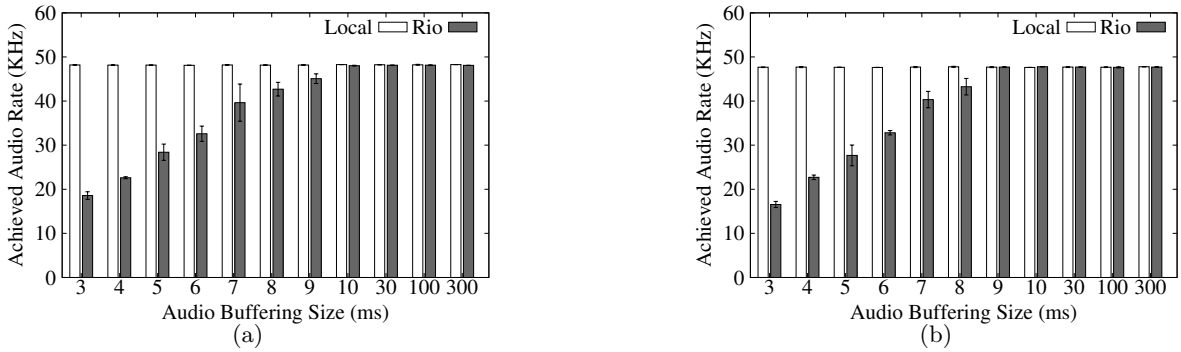


Figure 6: Performance of (a) speaker and (b) microphone. The X axis shows the buffering size in the HAL. The larger the buffer size, the smoother the playback/capture, but the larger the audio latency. The Y axis shows the achieved audio rate.

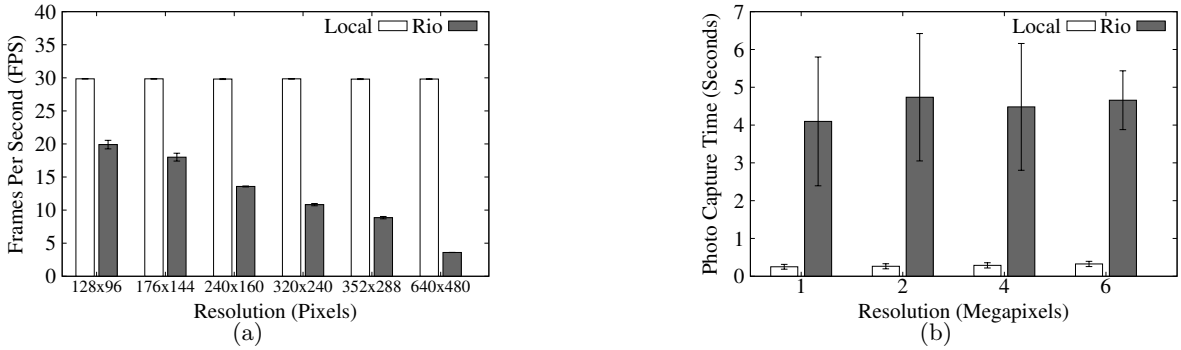


Figure 7: Performance of a real-time streaming camera preview (a) and photo capture (b) with a 21.9 Mbps wireless LAN connection between the client and server. Future wireless standards with higher throughput will improve performance without requiring changes to Rio.

We believe that the lower resolution camera preview supported by Rio is acceptable given that Rio supports capturing photos at maximum resolutions. Rio will support higher real-time camera resolutions using future wireless standards. For example, 802.11n, 802.11ac, and a 802.11ad can achieve around 200 Mpbs, 600 Mbps, and 7 Gbps of throughput respectively [19, 21]. Such throughput capabilities can support real-time camera streaming in Rio at 15 FPS for resolutions of 1280×720 and 1920×1080, which are the highest resolutions supported on Galaxy Nexus. Moreover, Rio can incorporate compression techniques, either in software or using the hardware-accelerated compression modules on mobile SoCs, to reduce the amount of data transferred for real-time camera, although we have not yet explored this optimization.

To evaluate photo capture, we measure the time from when the capture request is delivered to the camera HAL from the application until the HAL notifies the application that the photo is ready. We do not include the focus time since it is mainly dependent on the camera hardware and varies for different scenes. We take three photos in each experiment (totalling 9 photos in three experiments). Figure 7(b) shows the capture time for local and remote cameras using Rio. It shows that Rio adds noticeable latency to the capture time, mostly stemming from the time taken to transfer the raw images from the server to the client. However, the user only needs to point the camera at the targeted scene very briefly (similar to when using a local camera), as the image will be immediately captured in the server. This means that the *shutter lag is small*. It is important to note that the camera HAL in Galaxy Nexus uses the same buffer size regardless of resolution, hence the capture time is essentially resolution-independent. The buffer size is 8 MB, which takes about 3.1 seconds to transfer over our 21.9 Mbps connection. As with real-time camera streaming, future wireless standards will eliminate this overhead, providing latency on par with local camera capture.

Sensors: To evaluate remote sensor performance, we measure the average time it takes for the sensor HAL to obtain a new accelerometer reading. We measure the average time of 1000 samples for each experiment. Our results of 10 experiments show that the sensor HAL obtains a new local reading in 64.8 ms on average (with a standard deviation of 0.1 ms) and a new remote reading via Rio in 70.5 ms on average (with a standard deviation of 1.9 ms). The sensor HAL obtains a new reading by issuing a blocking `poll` operation that waits in the kernel until the data is ready, at which point the HAL issues a `read` file operation to read the new value. Rio causes overhead in this situation because one and a half round trips are required for the blocking `poll` to return and for the `read` to complete. Fortunately, this overhead is negligible in practice and does not impact the user experience.

Modem: We measure the time it takes the dialer and messaging applications to start a phone call and to send an SMS, respectively. We measure the time from when the

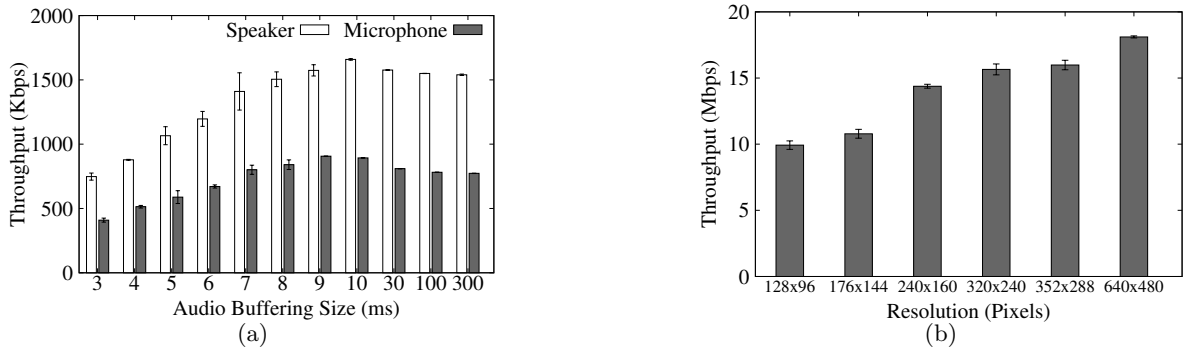


Figure 8: Throughput for using (a) audio devices (speaker and microphone) and (b) the camera (for video streaming) with Rio. Note that the Y axis is in Kbps and Mbps for (a) and (b), respectively.

user presses the "dial" or "send SMS" button until the notification appears on the receiving phone. Our measurements show that local and remote modems achieve similar performance, as the majority of time is spent in carrier networks (from T-Mobile to AT&T). For local and remote modems, the phone call takes an average of 7.8 and 7.9 seconds (with standard deviations of 0.7 and 0.3 seconds) while SMS takes an average of 6.2 and 5.9 seconds (with standard deviations of 0.3 and 0.6 seconds), respectively.

8.3 Throughput

We measure the wireless throughput required for using different classes of I/O devices remotely with Rio. Our results show that using sensors, audio devices, and camera remotely requires small, moderate, and large throughput, respectively.

We quantify the throughput by measuring the amount of data transmitted between the client and the server. We measure the number of bytes transmitted over the TCP socket, therefore, our results does not include the overhead due to the TCP header and headers of lower layers. We run each experiment for one minute and measure the throughput. For microphone, we record a one minute audio segment. For speaker, we play a one minute audio segment. For camera, we stream frames for one minute, and for accelerometer, we receive samples for one minute. We report each experiment three times and report the average and standard deviation. We reboot the phone before each experiment.

Figure 8(a) shows the results for the speaker and microphone. It shows that audio devices require a moderate throughput (hundreds of Kbps) and therefore Rio's performance for these devices is not throughput bounded. The throughput increases as we increase the audio buffering size, caps when the buffering size is 9-10 ms, and then decreases for larger buffering sizes. This trend can be explained as follows: at low buffering sizes, the audio performance on Rio is bounded by the link latency due to the high number of round trips. As a result, fewer audio samples are exchanged, resulting in lower throughput. As the buffering size increases, more audio samples are exchanged, hence requiring higher throughput. The number of audio samples exchanged (i.e., the audio rate) is maximized when the buffering size is 9-10 ms. For larger buffering sizes, the number of audio samples are fixed but the Rio's communication overhead decreases, resulting in a lower overall throughput. Moreover, the results show that speakers uses twice the throughput used by the microphone. This is because audio samples for the speaker are twice the size of the audio samples used by the microphone in our experiments.

Figure 8(b) shows the throughput results for video streaming from a remote camera using Rio. We show that the camera requires high throughput, which is why the link throughput is a performance bottleneck in our setup. At high resolutions, the link is almost saturated with sending the frame content; Rio's overhead is small (since the number of frames is small). For lower resolutions, more frames are transmitted and therefore the effect of link latency becomes more noticeable. This is why Rio fails to saturate the link throughput at these resolutions.

We also measure the throughput when using the accelerometer remotely with Rio. Our measurements show the average throughput is 52.05 Kbps with a standard deviation of 1.72. This shows that the throughput for sensors is small and therefore we can leverage low throughput, low energy links (such as Bluetooth) for these devices. Also, most of this throughput comes from Rio's overhead since accelerometer data are small.

8.4 Power Consumption

We evaluate the overhead of Rio in terms of both systems' power consumption. For each I/O device, we measure the average power consumption of the client and server in Rio and also the power consumption of the system when the I/O device is used locally. In each experiment, we use the device for one minute, similar to §8.3, and measure the average power consumption using the Monsoon Power Monitor [13]. We repeat each experiment three times and report the average and standard deviation of the experiment. The display consumes a large amount of power; therefore, we try to keep the display powered off whenever possible. More specifically, for the accelerometer and audio devices, we turn off the display on both the client and the server and also on the local system. For camera, we turn off the display only on the server but not on the client or the local system (because the camera frames are being displayed).

Figure 9 shows that Rio consumes noticeably more power than local devices. Considering the sum of the power consumption of client and server for Rio, Rio consume about 4× the power consumed by local accelerometer and audio devices, and 2× the power consumed by the local camera. These are expected results as Rio spans over two systems and uses the Wi-Fi interface. For camera, the source of power

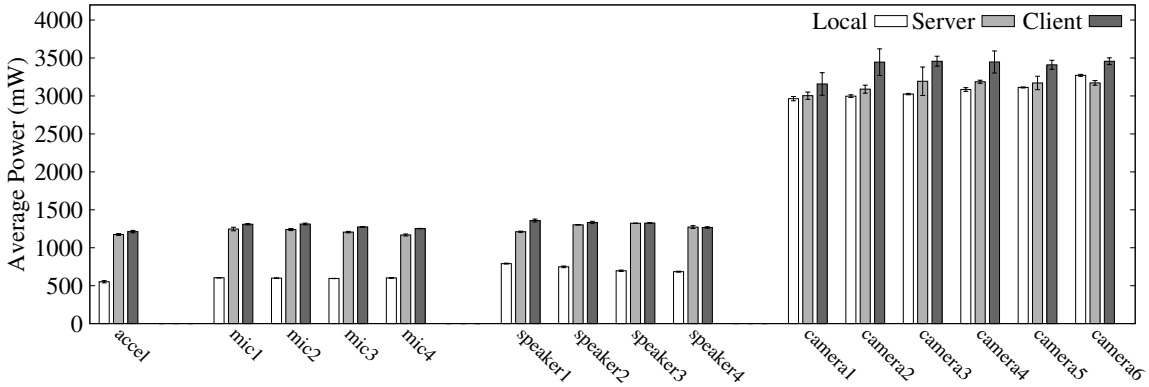


Figure 9: Average system power consumption of when accelerometer, microphone, speaker, and camera (for video streaming) are used locally and remotely with Rio, respectively. For Rio, the power consumption for both the server and the client are shown. Scenarios 1 to 4 for audio devices correspond to audio buffering sizes of 3 ms, 10 ms, 30 ms, and 300 ms, respectively. Scenarios 1 to 6 for camera correspond to resolutions of 128×96, 176×144, 240×160, 320×240, 352×288, and 640×480, respectively.

consumption of the local camera scenario is from the camera itself, the display, the CPU, and even the GPU, which is used for rendering the frames onto the screen. For Rio, the source of power consumption on the client is from the Wi-Fi interface, the display, the CPU, and the GPU, and for the server, from the Wi-Fi interface, the CPU, and the camera. For other devices, the main source of power consumption for the local scenario is the device itself and the CPU. For Rio, the source of power consumption on the client is from the Wi-Fi interface and the CPU, and on the server is from the Wi-Fi interface, the CPU, and the device.

8.5 Handling Disconnections

We evaluate Rio's ability to react to disconnections for the accelerometer. We play a game on the client using the server's accelerometer. Without warning, we disconnect the server and the client, and then trigger a disconnection event after a customizable threshold. Rio then transparently switches to using the local accelerometer so that we can continue to play the game using the client's own accelerometer.

9. RELATED WORK

The value of I/O sharing has been recognized by others for both mobile and non-mobile systems. However, existing solutions have three limitations: They do not support unmodified applications, do not expose all I/O device functions to the client, or are not generic to I/O classes. One possible advantage of these solutions is the incorporation of I/O class-specific optimizations. It is important to note that Rio can adopt I/O class-specific optimizations as well, if needed. For example, we have already incorporated one such optimization for audio devices (§7.3) where we modified the audio buffering size to improve the audio rate. Another possible optimization is the compression of camera frames before transmission.

I/O sharing for mobile systems: Existing I/O sharing solutions for mobile systems all suffer from the fundamental limitations described above. For example, IP Webcam [3] turns a mobile system's camera into an IP camera, which can then be viewed from another mobile system through a custom viewer application. The client system cannot configure all camera parameters, such as resolution; These parameters must be manually configured on the server. Wi-Fi Speaker [4] allows music to be played on a mobile system's speaker from a PC. It does not, however, support sharing the microphone. MightyText [11] allows the user to send SMS and MMS messages from a PC or a mobile system using the SIM card and modem in another system. It does not support phone calls.

Screen sharing: Applications like Miracast [12] allow one system to display its screen on another system's screen. Thin client solutions also display content received from a server machine on a client. Examples are the X window system [45], THINC [25], Microsoft Remote Desktop [29], VNC [43], Citrix Metaframe [6], and Sun Ray [46]. None of these solutions use the device file boundary; their choice of boundary is usually graphics-specific or even application-specific. For example, X sets the boundary between the application and X server. As a result, these solutions cannot support other classes of I/O devices.

Other I/O sharing solutions: Remote file systems [14,35, 44], network USB devices [7,17,20,34], Wireless Displays [9], remote printers [18], and IP cameras [8] support I/O sharing as well. These solutions are also specific to one I/O class, e.g., storage. Participatory and cooperative sensing systems collect sensor data from registered or nearby mobile systems [30,36]. These systems use custom applications installed on mobile systems and are therefore more limited than Rio, which supports a variety of I/O devices. Indeed, these systems can incorporate Rio to more easily collect sensor data from other systems.

Computation offloading: There is a large body of literature regarding offloading computation from mobile systems [32], e.g., Cyber Foraging [24], MAUI [28], and COMET [33]. I/O sharing, as is concerned in this work, invites a different set of research challenges and has a focus on system support rather than programming support. Nevertheless, both computation offloading and I/O sharing benefit from existing techniques for distributed systems. For example, both Rio and COMET employ DSM, albeit with different designs.

10. CONCLUDING REMARKS

We presented Rio, an I/O sharing solution for mobile systems that adopts a split-stack model at the device file boundary. We demonstrated that Rio overcomes the limitations of existing solutions by supporting unmodified applications, exposing all I/O device functionality to clients, and reducing development effort. We presented an implementation of Rio for Android and showed that it achieves adequate performance for various sharing scenarios and that it supports heterogeneous mobile systems. We next offer some insights into the limitations of the current design and implementation of Rio and possible solutions to overcome some of them.

Supporting more classes of I/O devices: Our current implementation supports four classes of I/O devices. It is possible to extend it to support graphics, touchscreen, and GPS, since they also use the device file interface. There are, however, two classes of I/O that Rio's design cannot support: network and block devices. This is because these I/O devices do not use the device file interface for communications between the process and the driver. Network devices use sockets along with the kernel networking stack and block devices use kernel file systems.

Sharing I/O with untrusted systems: In this paper, we assumed that the systems sharing I/O through Rio trust each other not to be malicious (§3.2). For such scenarios, Rio can simply adopt an authentication mechanism where it asks the client and server's owner(s) to authenticate the I/O sharing, after which Rio assumes that both systems are not malicious. However, supporting I/O sharing between untrusted systems creates new challenges for Rio, which fall into two categories. (*i*) *Protecting the server.* As also discussed in [22], device drivers are buggy, and malicious applications can abuse these bugs through the device file interface to compromise the driver protection domain [15]. In Rio, this means that a malicious process in the client can compromise the server. In order to solve this problem, the device driver and the device need to be sandboxed in a protection domain in the server, using techniques similar to Paradice [22], Nooks [48], and VirtuOS [40]. (*ii*) *Protecting the client.* An untrusted server can issue spurious memory copy operations to the client in order to compromise the client. The client stub can simply protect against this threat by strictly checking the memory copy operations requested by the server, similar to [22]. Note that the server can also snoop the client's data that are shared with the I/O device, e.g., the audio buffers. Since the server is completely untrusted, we cannot provide any isolation for the client's data. This is indeed an inherent problem to any I/O sharing systems, and not only to Rio.

Energy use by Rio: Using an I/O device via a wireless link obviously incurs more energy consumption than using a local one, as demonstrated in §8.4. In this work, we did not address energy optimizations for Rio. Rather, we note that most of the performance optimizations in Rio, e.g., those described in §5, lead to more efficient use of the wireless link and therefore to reduced energy consumption. We also note that Rio's quest to reduce latency rules out the use of the standard 802.11 power-saving mode. On the other hand, many known techniques that trade a little latency for much more efficient use of the wireless link can benefit Rio, e.g., data compression [26] and μPM [39].

Supporting iOS: iOS also uses device files and hence can be supported in Rio. Sharing I/O devices between iOS systems should require similar engineering effort reported in this paper for sharing I/O devices between Android systems. However, sharing I/O devices between iOS and Android systems require potentially non-trivial engineering effort, mainly because these two systems have different I/O stack components and API.

Acknowledgments

The work was supported in part by NSF Awards #1054693, #1065506, and #1218041. The authors would also like to thank Sreekumar Nair, then at Nokia Research, who contributed to an early prototype of I/O sharing for x86/Linux machines that exchanges file operations and memory copy operations over sockets. The authors also thank the anonymous reviewers and their shepherd, Professor Gaetano Borriello, for their useful comments.

11. REFERENCES

[1] http://www.theverge.com/2011/06/09/google-voice-skype-imessage-and-the-death-of-the-phone-number/.
[2] Android ION Memory Allocator. http://lwn.net/Articles/480055/.
[3] Android IP Webcam application. https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en.
[4] Android Wi-Fi Speaker application. https://play.google.com/store/apps/details?id=pixelface.android.audio&hl=en.
[5] Applications for taking self-portraits. http://giveawaytuesdays.wonderhowto.com/inspiration/10-iphone-and-android-apps-for-taking-self-portraits-0129658/.
[6] Citrix Metaframe. http://www.citrix.com.
[7] Digi International: AnywhereUSB. http://www.digi.com/products/usb/anywhereusb.jsp.
[8] Dropcam. https://www.dropcam.com/.
[9] Intel WiDi. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-wireless-display.html.
[10] Linux ksocket. http://ksocket.sourceforge.net/.
[11] MightyText application. http://mightytext.net.
[12] Miracast. http://www.wi-fi.org/wi-fi-certified-miracast%E2%84%A2.
[13] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.
[14] Network File System. http://etherpad.tools.ietf.org/html/rfc3530.
[15] Privilege escalation using NVIDIA GPU driver bug. http://www.securelist.com/en/advisories/50085.
[16] Rio Project Homepage (including a video demo). http://www.ruf.rice.edu/~mobile/rio.html.
[17] USB Over IP. http://usbip.sourceforge.net/.
[18] Web Services on Devices: Devices That Are Controlled on the Network. http://msdn.microsoft.com/en-us/library/windows/desktop/aa826001(v=vs.85).aspx.
[19] Wireless LAN at 60 GHz - IEEE 802.11ad Explained. In *Agilent White Paper*.
[20] Wireless USB. http://www.usb.org/wusb/home/.
[21] 802.11ac: The Fifth Generation of Wi-Fi. In *Cisco White Paper*, 2012.

[22] A. Amiri Sani, K. Boos, S. Qin, and L. Zhong. I/O Paravirtualization at the Device File Boundary. In *Proc. ACM ASPLOS*, 2014.

[23] ARM. Architecture Reference Manual, ARMv7-A and ARMv7-R edition. *ARM DDI*, 0406A, 2007.

[24] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying Cyber Foraging for Mobile Devices. In *Proc. ACM MobiSys*, 2007.

[25] R. A. Baratto, L. Kim, and J. Nieh. THINC: A Remote Display Architecture for Thin-Client Computing. In *Proc. ACM SOSP*, 2004.

[26] K. C. Barr and K. Asanović. Energy-Aware Lossless Data Compression. In *Proc. ACM MobiSys*, 2003.

[27] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. ACM SOSP*, 1991.

[28] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. ACM MobiSys*, 2010.

[29] B. C. Cumberland, G. Carius, and A. Muir. *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference*. Microsoft Press, 1999.

[30] T. Das, P. Mohan, V. Padmanabhan, R. Ramjee, and A. Sharma. PRISM: Platform for Remote Sensing Using Smartphones. In *Proc. ACM MobiSys*, 2010.

[31] G. S. Delp. The Architecture and Implementation of MEMNET: a High-Speed Shared Memory Computer Communication Network. *Doctoral thesis, University of Delaware*, 1988.

[32] J. Flinn. Cyber Foraging: Bridging Mobile and Cloud Computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 2012.

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

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

[35] P. J. Leach and D. Naik. A Common Internet File System (CIFS/1.0) Protocol. *IETF Network Working Group RFC Draft*, 1997.

[36] 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. ACM MobiSys*, 2012.

[37] K. Li. Ivy: A Shared Virtual Memory System for Parallel Computing. In *Proc. Int. Conf. Parallel Processing*, 1988.

[38] F. X. Lin, Z. Wang, and L. Zhong. K2: A Mobile Operating System for Heterogeneous Coherence Domains. In *Proc. ACM ASPLOS*, 2014.

[39] J. Liu and L. Zhong. Micro Power Management of Active 802.11 Interfaces. In *Proc. ACM MobiSys*, 2008.

[40] R. Nikolaev and G. Back. VirtuOS: An Operating System with Kernel Virtualization. In *Proc. ACM SOSP*, 2013.

[41] S. C. Park, M. K. Park, and M. G. Kang. Super-Resolution Image Reconstruction: a Technical Overview. *IEEE Signal Processing Magazine*, 2003.

[42] R. Raskar, J. Tumblin, A. Mohan, A. Agrawal, and Y. Li. Computational Photography. In *Proc. STAR Eurographics*, 2006.

[43] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 1998.

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

[45] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics (TOG)*, 1986.

[46] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture. In *Proc. ACM SOSP*, 1999.

[47] 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. In *Proc. ACM ASPLOS*, 1994.

[48] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. ACM SOSP*, 2003.

[49] Texas Instruments. Architecture Reference Manual, OMAP4430 Multimedia Device Silicon Revision 2.x. SWPU231N, 2010.

[50] B. Wilburn, N. Joshi, V. Vaish, E. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz, and M. Levoy. High Performance Imaging Using Large Camera Arrays. *ACM Transactions on Graphics (TOG)*, 2005.

[51] R. Woodings and M. Pandey. WirelessUSB: a Low Power, Low Latency and Interference Immune Wireless Standard. In *Proc. IEEE Wireless Communications and Networking Conference (WCNC)*, 2006.

[52] L. Yuan, J. Sun, L. Quan, and H. Shum. Image Deblurring with Blurred/Noisy Image Pairs. *ACM Transactions on Graphics (TOG)*, 2007.

[53] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 1992.