Intent to Share: Enhancing Android Inter-Component Communication for Distributed Devices

Breno Dantas Cruz and Eli Tilevich
Software Innovations Lab
Dept. of Computer Science
Virginia Tech
{bdantasc,tilevich}@cs.vt.edu

ABSTRACT

Data-intensive applications in diverse domains, including video streaming, gaming, and health monitoring, increasingly require that mobile devices directly share data with each other. However, developing distributed data sharing functionality introduces low-level, brittle, and hard-to-maintain code into the mobile codebase. To reconcile the goals of programming convenience and performance efficiency, we present a novel middleware framework that enhances the Android platform's component model to support seamless and efficient inter-device data sharing. Our framework provides a familiar programming interface that extends the ubiquitous Android Inter-Component Communication (ICC), thus lowering the learning curve. Unlike middleware platforms based on the RPC paradigm, our programming abstractions require that mobile application developers think through and express explicitly data transmission patterns, thus treating latency as a first-class design concern. Our performance evaluation shows that using our framework incurs little performance overhead, comparable to that of custom-built implementations. By providing reusable programming abstractions that preserve component encapsulation, our framework enables Android devices to efficiently share data at the component level, providing powerful building blocks for the development of emerging distributed mobile applications.

ACM Reference format:
Breno Dantas Cruz and Eli Tilevich. 2018. Intent to Share: Enhancing Android Inter-Component Communication for Distributed Devices. In *Proceedings of 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems, Gothenburg, Sweden, May 2018 (MOBILESoft'18),* 11 pages.
DOI: 10.475/123_4

1 INTRODUCTION

The Android platform is inherently component-based, with mobile apps comprising sets of interacting components. Each application component encapsulates a distinct functionality that can be accessed by external clients via a simple interface. However, this component-based encapsulation breaks when devices need to share data with each other. Implementing data-sharing functionality requires application developers to write low-level network communication code that directly manipulates the shared data. The resulting communication code is not reusable and hard to maintain.

Mobile, IoT, and wearable devices generate massive amounts of data, which needs to be transferred to other mobile devices. For example, a health monitoring app, running on a smartwatch, is periodically reading its wearer's vital signs, which are then displayed on an app, running on the wearer's smartphone or tablet. The vital signs information can be transmitted either directly, peer-to-peer or via a cloud-based server. However, the resulting information flow is device-to-device, a common pattern for a growing number of distributed mobile applications.

In this paper, we put forward a middleware framework that preserves the encapsulation of the Android component model when sharing data across devices. We evolve the ubiquitous Android Inter-Component Communication (ICC) into Remote ICC (RICCi), with the RICCi API mirroring that of ICC and also providing a declarative interface for specifying how to transfer component data (i.e., `Intent` objects) across the network. RICCi enables mobile application developers to use components, hosted on remote devices, through an API that resembles as close as possible that for using components hosted on the same device. Unlike RPC-based middleware platforms (e.g., CORBA [36], gRPC [13], Java RMI [15], etc.), the RICCi programming model is not procedure-oriented but is data-centric. That is, rather than modeling distributed communication as a procedure call, RICCi introduces the concept of *remote component data*, which is transferred across devices in accordance with a given data exchange pattern. In that way, RICCi bears similarity to the RESTful architecture [8], albeit with strong encapsulation and fine-grained control over how the data is transferred from the source to the destination.

RICCi requires that latency be treated as a first-class design concept. The RICCi API includes a declarative interface for specifying how and even whether remote component data should be transferred across the network; a component's data can be *copied*, *streamed*, or remain *remote*. This way, RICCi makes it impossible for the developer to "paper over the network"[16], disregarding the differences in latency between the communication within the same device and those across different devices over the network.

In addition to reusability and encapsulation, the reference implementation of RICCi provides all the necessary low-level networking support and can route the distributed communication either directly, peer-to-peer or via a broker. The role of a broker can be played by either an edge server, connected to devices via a local area network, or a cloud service, connected through a WAN. In addition, RICCi is

MOBILESoft'18, Gothenburg, Sweden
© 2018 ACM. 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

fully portable, as it works with any standard Android installation, without requiring any changes to the standard Android platform.

As an evaluation, we implemented two versions of distributed benchmark applications: one using our reference implementation of RICCi, and the other one using a custom-coded implementation based on an existing RPC middleware platform. Based on our evaluation, RICCi is comparable in terms of the resulting performance efficiency and energy consumption, while requiring fewer lines of code of lower complexity. Despite our reference implementation being Android-specific, the insights gained from this work can be applied to other mobile platforms, enabling distributed devices to efficiently interact with each other at the component level.

The contributions of this paper are as follows:

(1) We describe the design and implementation of RICCi — a novel middleware framework for intuitive and efficient sharing of component data across Android devices; unlike existing RPC-based middleware platforms, RICCi is structured around the concept of *remote component data*.
(2) We empirically evaluate the RICCi programming model in terms of performance and expressiveness.
(3) We present guidelines for application developers on how to use RICCi effectively in the development of emerging distributed mobile applications.

The rest of this paper is structured as follows. In Section 2, we present three motivating use cases for our solution; in Section 3, we present the technical background required to understand our contributions; in Section 4, we present the design of RICCi programming interface, its architecture, middleware and broker systems. In Section 5, we explain the implementation insights of our solution. In Section 6, we explain how RICCi streamlines the implementation of the motivation scenarios. In Section 7, we evaluate the performance and software engineering characteristics of RICCi. In Section 8, we discuss the applicability of our solution. In Section 9, we discuss the related state the art. Finally, in Section 10, we outline future work directions and present concluding remarks.

2 DISTRIBUTED DATA SHARING SCENARIOS

In this section, we present three examples of mobile distributed applications to demonstrate the need for a middleware platform that can streamline their implementation.

2.1 Sharing Documents Across Mobile Devices

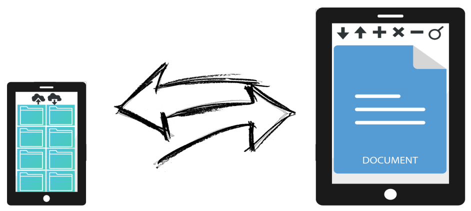

Figure 1: Copying files from smartphone to tablet

Mobile applications are often used in environments with limited network connectivity. Business professionals on the move often find themselves in locations, in which they cannot connect to cloud-based services for availability, bandwidth, or security reasons. Users can take notes on one device but edit them on another device that runs a proprietary document editing application (e.g., MS Word). Consider having to develop a document management application that synchronizes documents across multiple mobile devices (Figure 1). In the presence of a reliable network connection, a cloud-based solution can satisfy the requirements. However, in the absence of the standard network infrastructure, devices should be able to communicate with each other directly or possibly brokered by a LAN-connected edge server. Mobile application developers would like to be able to design and implement such applications without having to write special-purpose logic that addresses the aforementioned dissimilar network conditions. In addition, since Android uses components to access disk files, developers are likely to find it advantageous to retain the component-based representation for the documents when transferring them across the network.

2.2 Reading Input from IoT Devices

Private residences routinely feature several IoT devices installed in different locations. These sensors monitor temperature, noise, humidity, as well as provide video feeds and check for package deliveries. A house resident can use a mobile device to receive periodic readings from these IoT devices.

Figure 2: Streaming video data from IoT cameras to phone

If the Internet bandwidth is plentiful and readily available, a cloud-based service can be used to mediate the interaction (Figure 2). However, if the bandwidth is limited or expensive, developers should be able to express that the application is to use an edge server, connected via a local network, or that the devices are to communicate with each other directly peer-to-peer. Despite the complexity of having to support multiple network environments, developers may still prefer to follow a component-based design for such applications, with both the media data and its presentation logic represented as separate components. In other words, the application's business logic should not be polluted with low-level system functionality that selects the appropriate communication mechanism based on the network conditions in place. That is, the logic required to select the most suitable distributed communication pattern should be separate from the application's business logic.

2.3 Reading Properties of Temporal Data

In complex industrial environments, mobile devices are often used to monitor different complex processes and aggregate readings from various sensors. Due to the data accumulating rapidly and being of relevance only temporarily, it may be disadvantageous to transfer this temporal data to remote servers for storage, thus deeming the

Figure 3: Gauging the production process by remotely accessing properties of immovable data

data unmovable. However, various interested parties within the company may still want to query the accumulated data for some of its real-time properties, and they want to do so remotely from their mobile devices. For example, an employee can be carrying a tablet that receives input from the sensors installed throughout a manufacturing floor (Figure 3). A factory manager may want to inquire about some real-time aspect of the production process, while at a meeting in a different office. Notice, that the data is intended to be available to multiple users, as the mobile device collecting sensory data is *not* for personal use. The employee carrying the device around the manufacturing floor assumes that the collected sensory data from the floor will be accessible to the management at any time. To that end, consider having to develop an app capable of querying the unmovable data on the employee's tablet without having to transfer the data itself, with the manager's mobile device providing a viewing interface. Furthermore, developers would prefer to follow a component-based design, with the immovable data and presentation logic implemented as separate components, communicating with each other over the local network.

3 BACKGROUND

In this section, we introduce the technical background required to understand our conceptual contributions. The reader knowledgeable with key Android APIs and distributed programming concepts can safely skip this section.

3.1 Android Components

Each installed application in Android (i.e., *.apk file) typically runs in a separate process and can be composed of *Activities*, *Services*, *Content Providers*, and *Broadcast Receivers*. These four components communicate through messages called *Intents*, which are routed by the Android runtime. *Activities* are graphical components that provide a user interface to the client. The main *Activity* is instantiated upon the user launching an application. *Services* run in the background to execute long-running tasks. *Content Providers* manage access to persistent data. *Broadcast Receivers* receive and react to event notifications.

3.2 Inter-Component Communication (ICC)

Android applications are structured as a collection of components, each of which represents a reusable unit of functionality. An application can also use components of other applications, both within the same process or across processes, as long as the permission scheme in place allows it. Loosely coupled components interact with each other via ICC messages, represented as reusable `Intent`[1] objects [20, 25]. When interacting via ICC, Android components can operate both within the same application and across different ones. To render a component accessible to other applications, a developer sets its `exported` attribute to **`true`** in the manifest file[2].

3.3 Remote Method Invocation

Remote Method Invocation (RMI) provides remote procedure call (RPC [24]) functionality to Java applications. It enables any Java object on one JVM to invoke methods of an object running on another JVM [38]. The RMI abstractions render local and remote invocations almost identical to each other. The only difference is that remote methods must be declared as throwing `RemoteException`, thus requiring that the developer provide logic for handling partial failures, which are omnipresent in distributed computing.

4 DESIGN OVERVIEW

This section provides an overview of RICCi and discusses how it enhances ICC for inter-device component data sharing. In addition, this section also explains how RICCi works at runtime.

Our design requirements are two-fold. First, we aim at providing an inter-device programming abstraction that can be quickly mastered and put into practice by developers familiar with the Android abstractions for component interaction on the same device. Second, we want to ensure that the new abstraction sufficiently accommodates for the differences between the *same-device* and *device-to-device* computing environments, with a particular emphasis on recognizing the presence of latency, which is known to increase by orders of magnitude when switching from local to distributed computing environments. This recognition is essential to ensure that the resulting distributed mobile application maintains the required quality of service.

By designing RICCi, we want to enable Android developers to use the abstraction to build efficient distributed applications, while reasoning at the component level. Without properly designed programming abstractions, typical components sharing scenarios, such as the aforementioned, would require developers to write low-level code for each piece of the distributed functionality (e.g., network data exchange patterns, data marshaling, handling faults, etc.). Even though the actual exchanged data is naturally represented as Android components, the attendant encapsulation benefits would quickly disappear if these components' data is transferred across the network without proper abstractions. In contrast, RICCi enables developers to continue reasoning at the component level when implementing inter-device interactions.

4.1 General Design

RICCi evolves the built-in Android ICC programming paradigm, whose overriding design principle is to enable developers to integrate ready-made components into their applications, thereby reducing the programming effort and streamlining the development process. Hence, our design objective is to retain the software engineering benefits afforded by ICC, but also to extend them to environments that comprise multiple Android devices. In other

[1]https://developer.android.com/reference/android/content/Intent.html
[2]http://developer.android.com/guide/topics/manifest/manifest-intro.html

words, we would like to enable Android developers to leverage their familiarity with ICC to easily implement distributed applications capable of accessing components hosted by other Android devices.

Figures 4 and 5 show two code snippets, in which an application requests a document stored by an Android device. Figure 4 shows the code that uses the built-in ICC to request a document stored on the same device. Meanwhile, Figure 5 shows an equivalent RICCi implementation that requests a document stored on another device. As one can observe these code snippets are remarkably similar. In fact, the only difference is that the RICCi version uses type-compatible `RemoteIntent` objects (line 3 of Figure 5) instead of `Intent` objects (line 3 of Figure 4). The `RemoteIntent`'s constructor take an additional parameter, `Transfer.COPY`, thus declaratively specifying the *by-copy* semantics for the document component's data. We have elided the required handling of remote exceptions that the distributed runtime would raise in response to detecting various failures.

```
1 public void getDocument() {
2       Intent intent = new Intent(
3       Action.OPEN_DOCUMENT);
4       intent.addCategory(CATEGORY_OPENABLE);
5       intent.setType("text/plain");
6       startActivityForResult(intent, code);
7 }
```

Figure 4: Accessing document information with ICC

```
1 public void getDocument() {
2       RemoteIntent intent = new RemoteIntent(
3       Action.OPEN_DOCUMENT, Transfer.COPY);
4       intent.addCategory(CATEGORY_OPENABLE);
5       intent.setType("text/plain");
6       startActivity(intent);
7 }
```

Figure 5: Accessing document information using RICCi

Of course, the original ICC mechanism cannot be reused as is in a distributed setting. As mentioned above, distributed applications typically experience higher latencies than centralized applications. To accommodate the increased latency, our design requires that the developer specify how the component data is to be transferred across the network. Specifically, depending on the properties of component data, it can be either copied, streamed, or retained in place to be accessed remotely. `Intent` objects, encapsulating small data volumes, can be copied across the network efficiently; `Intents` representing media files can be streamed to be played on remote devices; while `Intents` with platform-specific dependencies are "anchored" to their devices, accessed by means of remote callbacks from the accessing devices. The RICCi API includes remote exceptions that reflect various partial failure conditions, likely to occur in different deployment environments. The developer using RICCi will have to handle these failures in an application-specific manner.

Naturally, when information is transferred across the network, the issue of *security* comes to the fray, and the developer must make appropriate provisions. Because RICCi enables component-level distributed programming, it promotes the encapsulation principle. Encapsulated functionality is easier to systematically enhance with additional properties than scattered functionality. In the RICCi architecture, it is known exactly when distributed interactions commence and end, and in which patterns the data is exchanged. This foreknowledge simplifies the provisioning of various security enhancements, such as adding encryption and following secure network transfer protocols.

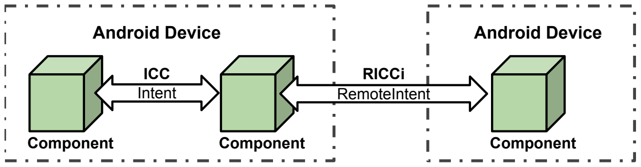

Figure 6: ICC vs. RICCi

Figure 6 shows a system diagram demonstrating the differences between ICC and RICCi.

4.2 System Architecture

Figure 7 shows an overview of a distributed mobile application developed using RICCi. In this application, the *Source Device* is the device that is making a request to access a component on another device. To that end, the *Source Device* declares and sends a `RemoteIntent` object. The *Target Device* is the device that receives and handles the request to access the component in question. The *Broker Server* is responsible for handling the connection between *Source Device* and *Target Device*. In addition, depending on the *Broker Server's* location (LAN or WAN), the broker is also responsible for relaying all the transmission between the connected *Source* and *Target* devices. We provide additional details about the Broker Server below. When devices are to communicate peer-to-peer, RICCi assumes that the devices have been properly configured for such interactions. Under these scenarios, the RICCi runtime handles all requests and responses without the aid of a broker.

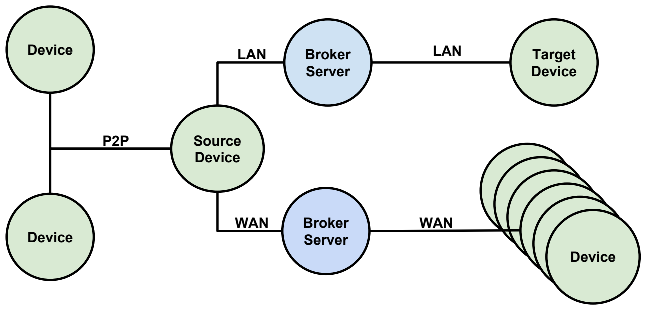

Figure 7: RICCi Architecture

RICCi Dataflow. When sharing component data across devices using RICCi, the process comprises three main phases: 1) The *Source* device creates a RemoteIntent (with the desired target information and data exchange patterns) and sends it to the *Target* device through a network. 2) The *Target* device receives the RemoteIntent, executes its defined action, and returns the results, following the specified data exchange pattern to the *Source* device. 3) The *Source* device receives the results, in a fashion prescribed by the specified exchange pattern.

4.3 System Architecture Components

RICCi runtime includes both the functionality executing on each of the participating devices as well as the distributed logic required for interacting with broker servers.

Device components: RICCi runtime is responsible for processing the component data sharing requests within a device. The runtime is also responsible for setting up a connection to Broker servers, which route the requests between the devices. We provide additional details on how RICCi connects to broker servers in Section 5. RICCi activities in both *Source* and *Target* devices comprise two threads responsible for managing incoming requests: a Messenger Thread and a Handler Thread. The Messenger Thread interacts with a transmission channel by sending and receiving data. The incoming data is transmitted to the Handler Thread. The Handler Thread is responsible for processing incoming requests and generating the appropriate responses, which are then retransmitted to the intended device by the Messenger Thread.

The Broker Server: The Broker server is responsible for delivering component data to the intended destination. Broker servers are divided into two categories: edge (LAN) and WAN brokers. A broker server is responsible for keeping track of the connected devices, resources and the component data that these devices have available for sharing. A broker server maintains a list of devices currently connected, and it analyzes the transferred messages, specifically the Bundle objects of the RemoteIntent, in order to determine their destination and data exchange patterns. RICCi runtime uses a priority queue, explained in Section 5, to determine to which broker a given device should connect at any given time. Broker servers connected via local networks are responsible for pairing up devices and setting a peer-to-peer connection between the paired devices. Each pair of devices has a Requester (making the requests) and a Provider (providing the resources). In terms of behavior, LAN brokers are slightly different from WAN brokers. LAN brokers are only responsible for pairing up the communicating devices. RICCi establishes direct connections for streaming and accessing data remotely. WAN brokers also pair up the communicating devices. In addition, they are also responsible for acting as a gateway that relays all the data transferred between any two interacting devices.

5 IMPLEMENTATION

In this section, we present additional technical details of the reference implementation of RICCi.

5.1 Data Exchange Logic

In abstract terms, RICCi facilitates the development of distributed Android applications. To that end, RICCi manages the connection between the interacting devices and the specifics of how the component data is transferred across the network. RICCi runtime transfers, serializes, and compresses component data, while handling the connection setup between the devices.

RemoteIntent objects are used by the reference implementation to transport component data in a way prescribed by a given data exchange pattern. RemoteIntent and its contents are serialized using GSON [3] to be represented as JSON objects. When created, a RemoteIntent can receive a *data exchange pattern* and an *action*. Actions define what kind of operations the RemoteIntent will perform, while data exchange patterns define how the results will be transmitted. Figure 8 shows the declaration of a RemoteIntent with an action (Intent.ACTION_PICK) line 2 and a data exchange pattern Transfer.Stream line 3. Developers can choose between three data exchange patterns: COPY, REMOTE or STREAM. The created RemoteIntent can be transmitted to a Target device by calling the RICCi implementation of the startActivity() method, which processes and sends RemoteIntents across the network by means of the sockets API.

```
1 RemoteIntent rIntent = new RemoteIntent(
2           Intent.ACTION_PICK, Transfer.STREAM);
3 startActivity(rIntent);
```

Figure 8: RemoteIntent declaration

The *Target* device (i.e., providing the resources) receives the RemoteIntent and tags it as an incoming message and forwards it to an IntentService. Developers are required to add the IntentService extending RICCiIntentService as a service in the manifest file, so the Android system would allow RICCi runtime to run. This service is responsible for determining how to handle RemoteIntents according to their data exchange patterns. After determining how to transfer the requested component data, the IntentService sends the contents of the RemoteIntent to a modified BroadcastReceiver, which calls the method startActivity ForResult() with the contents of the RemoteIntent, and then returns standard return codes. The results are returned to the IntentService, which follows the specified data exchange pattern to transmit the results back to the Target Device.

5.2 Data Exchange Patterns

This subsection provides additional details on the usage of data exchange patterns.

When specifying the **Copy** data exchange pattern, the developer creates a RemoteIntent with parameter Transfer.COPY. This parameter is used by RICCi runtime and by the Broker server in order to handle the requests. The Copy data exchange pattern signals RICCi runtime to return an exact copy of the data from the resulting operation of a component in the target device. The contents of the resulting Intent are placed inside a Bundle object in a RemoteIntent, which is sent back to the Source device. For example, RICCi runtime collects the *data* Intent resulting from the onActivityResult, which references context-specific information stored at the target device. RICCi then uses the reference stored at

[3]https://github.com/google/gson

the *data* Intent to directly access all the information that describes the user contact, such as number, email, etc., and it sends a copy to the Source device.

When specifying the **Stream** data exchange pattern, the developer creates a RemoteIntent with parameter Transfer.STREAM. This parameter is used by both RICCi runtime and Broker server as metadata, to determine how data should be accessed. In case of a WAN broker, this metadata also signifies that the server needs to gateway the incoming data stream. The Stream data exchange pattern signals the *Target*'s RICCi runtime that it needs to access a file and stream its contents. For example, if the resulting Intent is a reference to a media file, the Target's RICCi runtime locates the file, creates a ServerSocket, and sends its IP and port to the Source device. The Source device then connects to the server and requests the file transmission. Finally, the Target device serializes the accessed media file and starts the process of transmitting it to the Source device.

When specifying the **Remote** data exchange pattern, the developer creates a RemoteIntent with parameter Transfer.REMOTE and a RemoteAssistant object. The RemoteAssistant is used by RICCi runtime to execute the defined remote operations, while the RemoteIntent identifies the *Target* device's component. The Target device specifies a port number and waits for the Source device to interact with the object. RICCi runtime provides a gateway to perform remote operations.

For the implementation of this feature we used LipeRMI [1] as the current version of Java RMI is not supported by Android OS. LipeRMI is a plug-in replacement for the native Java RMI facility for the Android platform. The RemoteAssistant object is used by the *Source* device's RICCi runtime to perform the remote method invocations. The results of those operations are placed in a RemoteResultsHolder object, which is made available for access after the interaction is completed. By default the contents of the transmission are placed in a temporary file for streaming, however, the developers can modify RICCi's implementation to save the file in the device.

We are aware that developers may modify and enhance their applications to provide greater functionality and flexibility for usage of the remote pattern. To that end, RICCi features a code generator that synthesizes code compatible with the RICCi remote data exchange pattern, including both classes and the manifest file. The code generator enables developers to explore the features of RICCi to a greater extent to provide the desired method invocations. The code generator receives as input an ImmovableIntent object and generates a stub and skeleton classes that serve as client and server-side proxies, respectively. Their methods provide the logic required to dispatch a remote callback method on the Source Device and dispatch them to the Target Device. The generator needs to be rerun every time the number or type of methods in a given ImmovableIntent class changes.

5.3 Network Topology Selection

One of the features of RICCi runtime is to select the most advantageous network topology for two Android devices to exchange component data. Recall that the devices can interact either directly peer-to-peer or via a broker, provided by either a LAN-connected edge server or a WAN-connected cloud-based service. To determine which of the three topologies to select for a given inter-device component sharing, RICCi follows a predetermined set of customizable priorities. However, developers can change and customize the network selection logic at will. To that end, RICCi provides class RicciAppCompatActivity that developers can extend, providing a custom implementation of method connectWebSocket(). This method receives an array of IP addresses of the available servers as input, from which it initially connects its device either directly to another device or to a broker server. In the reference implementation of the method, connectWebSocket() sequentially checks over each provided IP for the first server available to serve as a broker. Developers using RICCi are required to call the connectWebSocket() within the method onCreate() in the MainActivity class of their distributed mobile application.

The RICCi runtime sets up the connection topology between a device and a broker as follows. First the device calls the connectWebSocket() method to check the list of possible available servers (1 & 2). If the server is available, RICCi performs a handshake operation and connects to the broker server (3), in which it provides its current device information, IP address, and possible capabilities to the server. The server receives the setup requests and persists the device's information (4). Finally, the server concludes the setup process and starts awaiting requests (5). In case of a request, the server goes over the list of connected devices and creates a pair **requester** (device making a request) and **provider** (device providing the information). Based on the server's location, the devices either communicate directly peer-to-peer or through a gateway.

One alternate system design would be to integrate Service Discovery (SD) [21] as a more flexible way for clients and servers to locate each other. In fact, there are several SD implementations (e.g., [14]) that can be integrated with the reference implementation of RICCi. One can also develop a simpler discovery component based on multicast. We plan to explore these alternate designs as a future work direction.

5.4 Broker Implementation

The RICCi broker system was implemented following the defined architecture presented in Section 4.3. In order to successfully manage messages and connected devices, the system stores each device information in Device objects and stores the connection information in Session objects. When a device makes a request, the broker system pairs it with an available second device and keeps the information regarding requester (device making data request) and provider (device providing data). All messages transmitted between connected devices have their metadata checked, in order to determine data exchange pattern, place of origin, and destination. Depending on the information being carried, the broker will behave differently. In case of remote or stream messages, LAN broker servers transfer the local network IP from the requester device to the provider device, so that devices can use RICCi to establish the device-to-device connection. In case of copy, the broker acts as a gateway. For WAN broker servers, the broker acts as a gateway by relaying all messages between connected devices.

6 MOTIVATING SCENARIOS REVISITED

In this section we, revisit the motivating scenarios from Section 2 and explain how RICCi facilitates their implementation.

6.1 Sharing Documents Across Mobile Devices

In this scenario, users have to be able to share text documents between their mobile devices without relying on having an Internet connection. To that end, RICCi can be used to implement an application that *copies* documents between devices by building on the ICC logic for accessing files within the same device. Because RICCi mirrors the ICC interfaces, the developer would have to indicate that the documents need to be copied across the network by using the RICCi's *Copy* data exchange pattern. RICCi automatically determines which communication mode to select by analyzing the current network conditions and available resources (more details in Section 4.3). For example, in the absence of any Internet connectivity, the devices would talk to each other via WiFi Direct; if no WiFi Direct connection is established, but both devices are connected to the Internet, they would share the documents over a cloud-based service; finally, if both devices are configured to talk to a local edge server, they would use it to broker their communication as well.

The issue in hand is that using RICCi enables mobile application developers to keep the same application logic for sharing the documents under any of the aforementioned network environments. Furthermore, the logic for opening a document component on another device is almost identical to that of opening a document on the same device. Any Android programmer, familiar with the ubiquitous ICC abstraction, should be able to quickly become productive when using RICCi.

6.2 Reading Input from IoT Devices

In this scenario, data must be streamed from IoT devices to a smartphone. To that end, developers can use RICCi to implement an application that uses the *Stream* data exchange pattern to stream the sensory data from the collecting devices to the ones displaying the information to the user. The RICCi runtime would determine which communication mode to select, based on the current network and server availability. If the user is in the vicinity of the IoT sensors (i.e., inside the house), the data would be streamed directly from device to device. If the user stepped outside into the yard, RICCi would leverage the house's edge server to mediate the communication. Finally, if the user tries to access the streamed data from a remote location, a cloud-based service would mediate the streaming session.

6.3 Reading Properties of Temporal Data

In this scenario, factory managers must be able to access data models on their subordinates' tablets, albeit without copying the models, which cannot be transferred across the network efficiently. To that end, the immovable collected sensory data can be represented as an Android component, exposing several remotely invocable methods that provide access to the data's various properties. Developers can use the *Remote* data exchange pattern to remotely access these methods, which are implemented as so-called *remote callbacks*. In other words, in this case, RICCi moves the code to the data, with the provided, automatically generated query interface mirroring the methods of the Android component that encapsulates the immovable data.

7 EVALUATION

This section describes the research questions, methodology, and experiments we conducted to evaluate RICCi.

7.1 Empirical Study

For the empirical evaluation, we deployed RICCi on a Google Nexus 6 and a Huawei KIW-L24 devices, with the target OS Android Marshmallow 6.0.1. For power measurements, we deployed RICCi on a LG Volt LS740 and used a Monsoon power monitor[4] to gather the energy consumption information.

Our evaluation objectives are to evaluate the differences between RICCi-based and hand-coded applications in terms of their respective software engineering metrics (i.e., lines of code, complexity) and runtime performance (i.e., time and energy consumption).

Therefore, our evaluation is driven by the following two research questions:

- **RQ1**: When sharing component data across devices, how do the Software Engineering metrics of a RICCi-based solution compare to that of a hand-coded solution?
- **RQ2**: When sharing component data across devices, how do the runtime performance and energy efficiency of a RICCi-based solution compare to that of a hand-coded solution?

The purpose of **RQ1** is to determine how a hand-coded solution compares to RICCi while performing the same set of operations, in terms of software metrics. To answer RQ1, we implemented three distributed mobile applications that exchange component data. Each application was implemented in two versions: (1) using one of RICCi data exchange patterns, (2) hand-coding the same functionality. We collected software engineering metrics for both versions of each application and then compared and contrasted them.

The rationale behind **RQ2** is to determine whether a RICCi-based or a hand-coded solution more efficient with respect to runtime performance and energy consumption. To answer RQ2, our three subject applications implement the functionality required to copy, stream, and remotely access data. For each subject, we implemented a RICCi-based and a hand-coded versions. To measure the streaming operations, we used a media file 3.64MB in size.

For copy, we implemented an application for sharing contact information. For stream, we implemented an application for streaming audio files. For remote access, we implemented an application that access methods of a `Intent` object containing contact information.

7.2 Evaluation Design

To measure the total number of lines of code (LoC) and code complexity, we used the Android Studio plug-in MetricsReloaded [19]. To measure runtime performance, we collected the total time taken by system operations and measured energy used by a device when executing a given component data sharing scenario. In order to

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

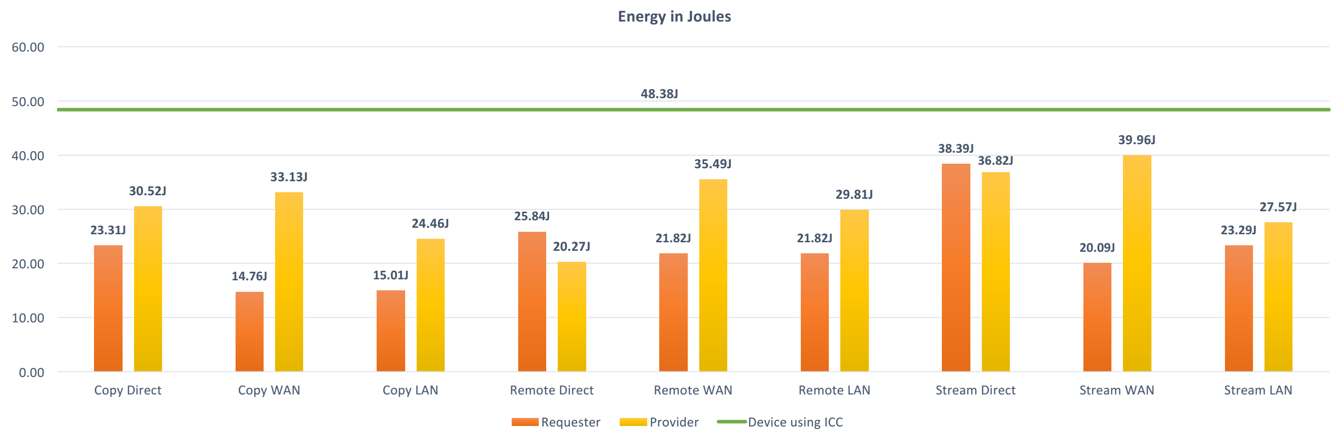

Figure 9: Average energy consumption in Joules

Table 1: Metrics

Exchange Pattern	Version	LoC	v(G)avg	v(G)tot
Copy	Hand Written	4819	2.14	506
	RICCi	3730	1.9	306
Stream	Hand Written	6735	2.25	755
	RICCi	3719	1.91	303
Remote	Hand Written	7791	2.06	882
	RICCi	3874	1.83	322

Table 2: Average time in milliseconds to perform operation for different data exchange pattern

Network	Exchange Pattern	Total Time	Transmission Time
WAN	Copy	2867.64ms	321ms
	Stream	8183.68ms	2578.28ms
	Remote	2165.29ms	231ms
LAN	Copy	3851.64ms	306.33ms
	Stream	8522.64ms	330.66ms
	Remote	3618.36ms	326.08ms
Direct	Copy	2616.68ms	3186.43ms
	Stream	5691.56ms	2490.18ms
	Remote	3179.58ms	1204.42ms

measure energy consumption, we used the Monsoon power monitor to collect the idle power and the average power of a LG Volt LS740, while performing each of the data exchange patterns. We then subtracted the idle power from the measured power during the experiments to determine the actual energy consumed by the distributed functionality. We calculated the energy consumption for a time interval of 70 seconds.

7.3 Results

Table 1 presents the numbers of lines of code (LoC), average cyclomatic complexity (v(G)avg), and total complexity (v(G)tot) for the total of all non-abstract methods in each project. These values show that the hand-coded versions for copy, stream, and remote are larger and more complex than their RICCi-based counterparts.

Table 2 shows the performance values of a RICCi-based application in terms of the time it took to complete all requests as required by different data exchange patterns under three dissimilar network environments. These results show that the time required to complete the component sharing requests is closely related to the amount of data transferred and latency of the underlying network.

Figure 9 shows the values of the average energy consumed by each of the data exchange patterns in joules. The information refers to requesters devices (devices making the requests) and providers (remote devices providing the resources). We also show average energy consumption for a basic ICC operation while retrieving contacts from one device. Figure 9 shows that for all cases, the average energy consumption for RICCi requesters and providers devices is lower than for performing an operation (accessing a phone contact) using ICC 48.38J. We speculate that the higher energy consumption for ICC is related to the necessity to perform more operations than when operating within a single device, while for RICCi the energy costs come from data transfer operations. In addition, for most data exchange patterns, the provider device will be consuming more energy than the requester counterpart. The two exceptions are for the remote and stream transfers, which use direct (peer-to-peer) communication.

7.4 Threats to Validity

As any study, our evaluation also carries threats to validity. We identify the following sources of validity threats.

Internal Threats: We implemented all the applications used for the experiments. It is possible that different hand-coded solutions would show different performance characteristics. In addition, the size of the files transferred during the experiments can also impact the performance.

External Threats: In terms of power consumption, Android phones have background processes that may require additional energy. To mitigate this threat, we ended all background processes, and performed multiple runs, to properly estimate the amount of energy consumed by each test. Android OS is being continuously updated, so newer versions may be better optimized, particularly w.r.t. energy consumption, than the one that we used for our experiments, thus improving the runtime performance results across the board. In terms of time, different networks offer dissimilar transmission latencies, and different devices with dissimilar hardware capabilities can provide unequal response times. In terms of accuracy of our power measurement procedures, our readings are directly related to the accuracy of the power monitoring devices used in the experiments.

7.5 Summary of the Results

We derived several insights from our experimental study. First, RICCi is not only comparable to hand-coded solutions in terms of complexity and in terms of lines of code, but also can provide a smaller code size and less complex solutions for mobile application developers to maintain. Second, the runtime performance in terms of the total time can vary, being affected by the latency of the network in place and the amount of transferred data. Finally, we found that RICCi operations on each of the participating devices can consume less energy than the equivalent operations over ICC within a single device, an insight motivating the use of distributed processing to reduce the amount of energy consumed by individual devices.

8 APPLICABILITY

In this section, we present guidelines for Android developers, with the goal of helping them apply RICCi effectively to share component data across devices.

8.1 Usage Guidelines

RICCi can be quite effective in those scenarios, in which component data type and size may constantly change during the development process. For example, in the initial development stages, data being transferred was in small volumes and in plain-text form. However, after some iterations, the nature of the requirements changed, now requiring that the application stream large video files between devices. With RICCi, developers will be relieved from the necessity to re-implement from scratch the functionality required to stream larger files. Instead, the required change in component data sharing semantics can be accomplished by changing a couple of lines of code, due to the declarative nature of the RICCi programming model. Based on the changed data exchange pattern, RICCi runtime can automatically handle the changes in the component data's size and transmission procedure. In terms of maintainability, our measurements show that the distributed functionality implemented using RICCi in most cases takes fewer lines of code. However, it is always less complex, to implement than the corresponding hand-coded version. In terms of security, our component-based design preserves encapsulation, thus facilitating security enhancements. All the functionality is encapsulated in individual components, which talk to each other using pre-defined data exchange patterns.

In addition, RICCi can be useful to a development team that lacks sufficient time required to learn how to modify some legacy code that handles data distribution over the network. Developers may find that the similarities of RICCi to ICC make it easier to understand how it works and how to interact with the required distributed component data. In terms of maintainability, RICCi allows developers to override and modify the behavior of virtually every piece of RICCi functionality. In other words, RICCi offers developers the ability to easily change the distributed functionality of a mobile application, as required by changes in project requirements.

8.2 Limitations

RICCi may be inadequate to all projects. In some situations, the requirements may render this programming mechanism inapplicable. For instance, since RICCi facilitates the sharing of component data across the network, its performance is closely related to the available network's capacities. Developers deeply experienced in low-level networking and with rigid requirements in terms of security and access latencies would be advised to stay away from using distributed abstractions, such as RICCi. Instead, they can achieve comparable or even superior performance characteristics if they manually implement their own low-level, hand-optimized solutions.

9 RELATED WORK

Our work is related to numerous research efforts that focus on facilitating distributed computing by means of programming frameworks and abstractions. It would be unrealistic to compare and contrast our work with all these prior efforts. Hence, we limit the related work coverage to the most relevant or recent approaches.

The remote procedure call (RPC) [2] has been the foremost programming mechanism for constructing distributed systems. RPC provides programming abstractions and runtime support to make invoking methods in a different address space as similar as possible to invoking those in the same address space. In distributed computing, RPC transfers each remote call from a client to execute on a server, while returning back the results. Examples of object-oriented variants of RPC include DCOM [3], CORBA [36], and Java RMI [15]. Mobile Plus [26] provides an RPC-based middleware to enable Android devices to share resources; it does so by modifying the Android OS, so as to properly support all the required functionalities with adequate performance. Since one of the key design objectives of RICCi is portability, our reference implementation runs entirely in user space and requires no changes to the Android platform.

In Android, each process runs in a separate address space. To enable processes to communicate with each other within the same device, Android features the Binder framework, which provides RPC-based communication between client and server processes. Binder passes data to remote method calls, returning the results to the client's calling thread. Android even offers Android Interface Definition Language (AIDL)[5], similar to CORBA IDL, for developers to define a set of remote methods to be invoked between client and server processes [11, 12, 33].

[5]https://developer.android.com/guide/components/aidl.html

Despite the ubiquity of RPC, this distributed programming paradigm has been critiqued as not realistically reflecting the actual differences between the local and remote computing models [37]. In a way, the RPC abstraction may be too powerful, making remote calls look indistinguishable from local calls, and thus encouraging the questionable "papering over the network" designs for distributed applications [16]). Because remote calls can take orders of magnitude longer than local calls to execute, treating them uniformly is only possible in low-latency distributed systems with high resource availability. Perhaps, that is why RPC-based systems have been particularly successful as a communication mechanism between different processes on the same machine.

We have designed RICCi to support a distributed programming model that is distinct from RPC. Instead, RICCi extends the Inter Component Communication (ICC) framework, which is fundamentally asynchronous and event-based. When implementing RICCi-based applications, developers are required to take into account the remote data' properties (i.e. its size and privacy) in order to determine how and whether to transfer it across the network. As a result, RICCi is not an RPC-based but a remote data-based distributed programming abstraction. Internally, RICCi does make use of RPC to implement its remote access data exchange pattern, in which the data remains stationary and instead accessed by means of remote callbacks, implemented using an extant RPC system.

Regarding related work on middleware approaches for resource sharing, ShAir [7] is a latency-tolerant data sharing approach that uses P2P communication for distribute mobile applications. While similar in its objectives, RICCi maintains strong encapsulation for P2P communication, in which the shared resources remain in the form of components. RICCi also shares goals with Sip2Share [4], which is an Android framework that extends parts of the built-in mechanism for sharing services and activities. A distinguishing characteristic of RICCi is the precise control over how remote data is transferred across the network; it provides this control by means of three declarative data exchange patterns, which specify how exactly to transfer remote data. μMAIS [30] also focuses on similar objectives, but unlike RICCi, it exports distributed resources in the form of Web Services. Mist [34] follows a publish-subscribe paradigm to provide a latency-tolerant middleware framework. Similarly to RICCi, Mist focuses on delivering *data elements* across the network. However, RICCi also differentiates its data delivery policies based on the type and volume of the transferred data. MobiClique [28] leverages nearby existing social network contacts to form ad hoc networks for data sharing. Following an ad hoc approach to form networks or to share component data can be an interesting future work direction.

Another popular mechanism for constructing distributed mobile applications is code offloading, which generally either relies on the opportunistic execution of code by remote servers [9, 32] or on automated application partitioning [35]. In the presence of network connectivity to bind mobile and cloud resources, the potential of code offloading lies in the ability to sustain energy-intensive applications by partitioning their power-intensive functionality to run at a remote server (e.g., Jade [31], CloudAware [27]). Multiple research efforts have proposed offloading strategies to empower smartphone applications with cloud-based resources [5, 6, 10, 17, 18]. Massari *et al.* [22] propose to use code-offloading techniques to use available computational power from discarded cellphones. Even though RICCi is not based on code offloading, it can be used as a middleware platform to offload local components to run at a remote device. We plan to explore this possibility as a future work direction.

In terms of network-aware applications, Pinte *et al.* [29] study several major techniques and present examples of adding network-awareness and distribution to mobile applications. To explore how to access distributed Android resources, Nakao *et al.* [23] study how feasible would it to be to use the inter-process communication mechanism to invoke remote services.

10 CONCLUSIONS AND FUTURE WORK

We have presented the design and implementation of RICCi, a distributed framework for sharing component data across Android devices. RICCi provides programming abstractions and runtime support to facilitate the implementation of inter-device component data sharing for Android application by enhancing the Android ubiquitous ICC mechanism. The RICCi declarative programming model features a set of pre-defined data exchange patterns for developers to specify the semantics for passing component data across the network. By focusing on remote data rather than procedures, RICCi treats distributed communication latency as a first class software design parameter. Even with respect to possible security risks, the encapsulation benefits of RICCi can streamline the implementation of security enhancements. As an evaluation, we have compared the performance, complexity, and code size of RICCi-based and hand-coded versions of three distributed mobile applications that share data between applications on different Android devices. We have found that RICCi-based implementations provide comparable performance and lower code complexity than their hand-coded counterparts.

We have identified several possible future work directions. First, we would like to explore the possibility of extending RICCi to add support for heterogeneous device-to-device environments, in which, for example, Android devices can exchange components of iOS devices and vice versa. Second, we would like to thoroughly explore the security threats to RICCi-based applications and possible defenses. Third, we would like to explore how RICCi can facilitate the adaptation of distributed legacy code to be able to share component data without breaking encapsulation. And lastly, we would like to conduct controlled user studies with Android developers to determine how to improve and refine the usability and expressiveness of RICCi.

AVAILABILITY

The reference implementation of RICCi as well as all the subject applications and benchmarks can be accessed via an online appendix at: https://github.com/brenodan/ricci

ACKNOWLEDGEMENTS

Author Breno Dantas Cruz is supported by the Capes foundation Science Without Borders program, process number 13003-13-5. This research has been supported in part by the US National Science Foundation through the Grants #1649583 and 1717065.

REFERENCES

[1] Felipe Santos Andrade. 2006. LipeRMI a light weight internet approach for remote method invocation. (2006). http://lipermi.sourceforge.net/

[2] Andrew D Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)* 2, 1 (1984), 39–59.

[3] Nat Brown and Charlie Kindel. 1998. Distributed component object model protocol–DCOM/1.0. *Online, November* (1998).

[4] Gerardo Canfora and Fabio Melillo. 2012. Sip2Share-A Middleware for Mobile Peer-to-Peer Computing. *ICSOFT* 12 (2012), 445–450.

[5] Eric Chen, Satoshi Ogata, and Keitaro Horikawa. 2012. Offloading Android applications to the cloud without customizing Android. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on.* IEEE, 788–793.

[6] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems.* ACM, 301–314.

[7] Daniel J Dubois, Yosuke Bando, Konosuke Watanabe, and Henry Holtzman. 2013. ShAir: Extensible middleware for mobile peer-to-peer resource sharing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* ACM, 687–690.

[8] Roy T Fielding and Richard N Taylor. 2000. *Architectural styles and the design of network-based software architectures.* University of California, Irvine Doctoral dissertation.

[9] Huber Flores, Pan Hui, Sasu Tarkoma, Yong Li, Satish Srirama, and Rajkumar Buyya. 2015. Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine* 53, 3 (2015), 80–88.

[10] Huber Flores and Satish Srirama. 2013. Mobile code offloading: should it be a local decision or global inference?. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services.* ACM, 539–540.

[11] Google. 2011. Android Interface Definition Language (aidl). (2011). http://developer.android.com/guide/developing/tools/aidl.html

[12] Google. 2011. Binder Java Documentation. (2011). http://developer.android.com/reference/android/os/Binder.html

[13] Google. 2017. gRPC a high performance, open-source universal RPC framework. (2017). https://grpc.io

[14] IETF Zeroconf Working Group. 1999. Zero Configuration Networking (Zeroconf). (1999). http://www.zeroconf.org/

[15] RMI Javasoft Java. 1997. Java Remote Method Invocation Specification. *Sun Microsystems* (1997).

[16] Samuel C Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. 1994. A note on distributed computing. (1994).

[17] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE.* IEEE, 945–953.

[18] Young-Woo Kwon and Eli Tilevich. 2012. Energy-efficient and fault-tolerant distributed mobile execution. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on.* IEEE, 586–595.

[19] Bas Leijdekkers. 2016. Android Interface Definition Language (AIDL). (2016). https://plugins.jetbrains.com/plugin/93-metricsreloaded

[20] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTa: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 280–291.

[21] Mikael Linden, David Simonsen, Andreas Åkre Solberg, Ingrid Melve, and Walter M Tveter. 2009. Kalmar union, a confederation of Nordic identity federations. In *TERENA Networking Conference.*

[22] Giuseppe Massari, Michele Zanella, and William Fornaciari. 2016. Towards Distributed Mobile Computing. In *Mobile System Technologies Workshop (MST), 2016.* IEEE, 29–35.

[23] Kazuhiro Nakao and Yukikazu Nakamoto. 2012. Toward remote service invocation in Android. In *9th International Conference on Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC).* IEEE, 612–617.

[24] B J Nelson. 1981. *Remote procedure call.* Ph.D. Dissertation. Pittsburgh Univ., Pittsburgh, PA. http://cds.cern.ch/record/132187 Presented on May 1981.

[25] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13).* 543–558.

[26] Sangeun Oh, Hyuck Yoo, Dae R Jeong, Duc Hoang Bui, and Insik Shin. 2017. Mobile Plus: Multi-device Mobile Platform for Cross-device Functionality Sharing. (2017).

[27] Gabriel Orsini, Dirk Bade, and Winfried Lamersdorf. 2015. Computing at the mobile edge: designing elastic android applications for computation offloading. In *IFIP Wireless and Mobile Networking Conference (WMNC), 2015 8th.* IEEE, 112–119.

[28] Anna-Kaisa Pietiläinen, Earl Oliver, Jason LeBrun, George Varghese, and Christophe Diot. 2009. MobiClique: middleware for mobile social networking. In *Proceedings of the 2nd ACM workshop on Online social networks.* ACM, 49–54.

[29] Kevin Pinte, Dries Harnie, and Theo DfiHondt. 2011. Enabling cross-technology mobile applications with network-aware references. In *Coordination Models and Languages.* Springer, 142–156.

[30] Pierluigi Plebani, Cinzia Cappiello, Marco Comuzzi, Barbara Pernici, and Sandeep Yadav. 2012. MicroMAIS: executing and orchestrating Web services on constrained mobile devices. *Software: Practice and Experience* 42, 9 (2012), 1075–1094.

[31] Hao Qian and Daniel Andresen. 2015. Extending mobile device's battery life by offloading computation to cloud. In *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on.* IEEE, 150–151.

[32] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for VM-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009).

[33] Thorsten Schreiber. 2011. Android binder. *A shorter, more general work, but good for an overview of Binder. http://www .nds .rub .de/media /attachments /files/2012/03/binder.pdf* (2011).

[34] Magnus Skjegstad, Frank T Johnsen, Trude H Bloebaum, and Torleiv Maseng. 2012. Mist: A reliable and delay-tolerant publish/subscribe solution for dynamic networks. In *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on.* IEEE, 1–8.

[35] Eli Tilevich and Yannis Smaragdakis. 2009. J-Orchestra: Enhancing Java Programs with Distribution Capabilities. *ACM Trans. Softw. Eng. Methodol.* 19, 1, Article 1 (Aug. 2009), 40 pages. DOI:http://dx.doi.org/10.1145/1555392.1555394

[36] Steve Vinoski. 1997. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine* 35, 2 (1997), 46–55.

[37] Steve Vinoski. 2005. RPC under fire. *IEEE Internet Computing* 9, 5 (2005), 93–95.

[38] Ann Wollrath, Roger Riggs, and Jim Waldo. 1996. A Distributed Object Model for the Java^{TM} System. *Computing Systems* 9 (1996), 265–290.