Programming Support for Data Intensive Distributed Mobile Applications at the Edge

Breno Dantas Cruz
Software Innovations Lab
Virginia Tech
bdantasc@cs.vt.edu

ABSTRACT

Mobile, IoT, and wearable devices have been transitioning from passive consumers of remote data to active generators of massive amounts of data. Mobile apps often need to move data, generated on one device, to other nearby devices for processing. For example, when reading its wearer's health vitals, a health monitoring app on a wearable device needs to transfer the data to the wearer's smartphone for display and analysis. This local processing of data by means of nearby computing resources has been promoted as a solution to network bandwidth bottlenecks, and commonly referred to as edge computing. Despite the critical dependence of edge computing on the device-to-device data sharing functionality, its mainstream implementations introduce low-level and hard-to-maintain code into the mobile codebase. To address this problem, this research introduces Remote ICC (RICCi), a novel middleware framework that provides programming support for data-intensive mobile applications at the edge, thereby reconciling programming convenience and performance efficiency. RICCi builds upon the native Android Inter-Component Communication (ICC) to simultaneously support seamless and efficient inter-device data sharing via a convenient and familiar programming model. To reach these design objectives, RICCi innovates in the middleware space by offering distributed programming abstractions that are data-oriented rather than procedure-oriented, thereby elevating latency into a first-class design concern for developing distributed mobile apps.

ACM Reference Format:
Breno Dantas Cruz. 2018. Programming Support for Data Intensive Distributed Mobile Applications at the Edge. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3197231.3198443

1 INTRODUCTION

The Android platform promotes encapsulation via its component-based programming model, in which a mobile app comprises a set of interacting components. These components communicate with each other via clean client interfaces. Maintaining such component-based encapsulation for device-to-device communication would provide great software engineering benefits, but the current technologies used in that space introduce low-level and hard-to-maintain code to the mobile codebase.

To raise the abstraction level and improve the maintainability of device-to-device Android communication, we have created the RICCi [1] middleware framework. RICCi enables mobile apps to access components on remote devices by handling low-level functionality, such as connection management and data transmission. The RICCi API mirrors that of Android ICC, the built-in mechanism for intra-component interactions on the same device.

Unlike RPC-based middleware platforms (e.g., gRPC [2], Java RMI [3], CORBA [6], etc.), RICCi offers a *data-oriented* rather than *procedure-oriented* programming model to the developer. That is, rather than modeling distributed communication as a procedure call, RICCi features *remote component data*, which is transferred across devices in a fashion specified by the developer to treat latency as a first-class design concept. Specifically, the RICCi API provides *exchange patterns*, which include *copy*, *stream*, or *remote-access*.

Figure 1 compares the apps architectures of ICC and RICCi. ICC connects components within a *single* device, while RICCi connects components hosted by *different* devices.

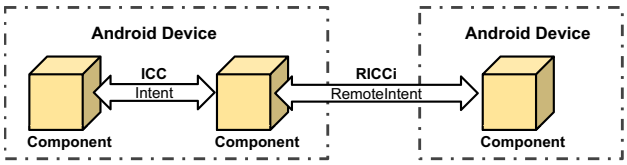

Figure 1: System using ICC and RICCi

2 OVERVIEW

RICCi provides programming support to facilitate the development of distributed Android apps by managing the constituent details of transferring the data (e.g., serialization, handling, and marshaling). Its runtime can both transfer component data directly peer-to-peer or via Broker servers, over a LAN or a WAN.

Figure 2 shows how RICCi can be used to stream a media file. The RICCi API mirrors that of ICC, but RICCi replaces native `Intent` objects [4, 5] with type-compatible `RemoteIntent` objects (line 3). The `RemoteIntent` constructor takes the extra *data exchange pattern*, `Transfer.STREAM`, to declaratively specify the *by-stream* semantics for this data transfer. Other transfer semantics are *by-copy* (`Transfer.COPY`) and *by-remote-access* (`Transfer.REMOTE`). The required handling of remote exceptions is omitted.

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3198443

```
1  RemoteIntent intent = new RemoteIntent(
2  Action.OPEN, Transfer.STREAM);
3  intent.addCategory(CATEGORY_OPENABLE);
4  intent.setType("*/.mp3");
5  startActivity(intent);
```

Figure 2: Accessing *by-copy* a document using RICCi

Table 1: Collected Software Engineering Metrics

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

3 EVALUATION

Broadly speaking, the key goal of this evaluation is to compare RICCi-based apps from their hand-coded counterparts in terms of their respective software engineering metrics and performance. Therefore, we pose the following two research questions:

- **RQ1**: How do the software engineering metrics of a RICCi-based variant compare to that of a hand-coded one, when implementing the inter-device component data sharing?
- **RQ2**: How do the energy and performance efficiency of a RICCi-based variant compare to those of a hand-coded one, in the inter-device component sharing functionality?

To answer these questions, we implement two versions of distributed benchmark applications. The first uses the RICCi reference implementation, while the other is a custom-built variant that relies on an existing RPC middleware platform.

Table 1 shows the values for the following software engineering metrics collected using MetricsReloaded[1]: 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 RICCi-based implementations were less complex and more concise than their hand-coded counterparts.

Table 2 shows the average time that the RICCi-based implementations took to access remote component data under different network environments. The results show the time taken to complete the requests is correlated with the amount of data transferred and the latency of the underlying network. In addition, in terms of energy consumption, RICCi-based apps consumed less energy than their hand-coded counterparts. Specifically, we measured the energy consumed by a file transfer operation. In RICCi, we applied the *by-stream* exchange pattern over peer-to-peer communication. Hand-coded implementations consumed 48.38J on average, with the maximum consumption for RICCi-based implementation consuming only 38.39J at the client side and 36.92J at the server side.

[1]https://plugins.jetbrains.com/plugin/93-metricsreloaded

Table 2: Transfer latency for different exchange patterns

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

3.1 Summary of the Results

The experimental study reveals several insights. First, RICCi-based implementations are not only comparable to hand-coded solutions in terms of complexity and size, but they often take less code of lower complexity to implement, thus facilitating maintenance. Second, the total runtime performance tends to vary, as influenced by the network latency and the volumes of transferred data. RICCi operations on each participating device consume less energy than the equivalent operations over ICC on the same device, thus showing how distributed processing can reduce the amount of energy consumed by each participating device.

4 FUTURE WORK

RICCi is a work in progress, with several promising future work directions. First, we plan to study the security threats and possible defenses of RICCi-based apps, as distributed communication introduces vulnerabilities that can be exploited. Second, we plan to explore how RICCi can facilitate the adaptation of distributed legacy code for remote component data sharing while maintaining strong encapsulation.

Acknowledgements

The author is supported by the Capes foundation SWB program #13003-13-5. This research is supported in part by NSF through the grants #1649583 and 1717065.

REFERENCES

[1] Breno Dantas Cruz and Eli Tilevich. 2018. Intent to Share: Enhancing Android Inter-Component Communication for Distributed Devices. *MobileSoft* (2018).
[2] Google. 2017. gRPC a high performance, open-source universal RPC framework. (2017). https://grpc.io
[3] RMI Javasoft Java. 1997. Java Remote Method Invocation Specification. *Sun Microsystems* (1997).
[4] 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.
[5] 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 *the 22nd USENIX Security Symposium (USENIX Security 13)*. 543–558.
[6] Steve Vinoski. 1997. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications mag.* 35, 2 (1997), 46–55.