An Efficient Mobile-based Middleware Architecture for Building Robust, High-performance Apps

Oscar J. Romero
Machine Learning Dept., Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA, 15145
Email: oscarr@andrew.cmu.edu

Sushma A. Akoju
Machine Learning Dept., Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA, 15145
Email: sakoju@andrew.cmu.edu

***Abstract*—As smartphones become increasingly more powerful, a new generation of highly interactive user-centric mobile apps emerge to make user's life simpler and more productive. However, the construction of such apps requires developers to spend a considerable amount of time dealing with the architecture constraints imposed by the wide variety of platforms, tools, and devices offered by the mobile ecosystem, thereby diverting them from their main goal of building such apps. Therefore, we propose a mobile-based middleware architecture that alleviates the burdensome task of dealing with low-level architectural decisions and fine-grained implementation details by focusing on the separation of concerns and abstracting away the complexity of orchestrating device sensors and effectors, decision-making processes, and connection to remote services, while providing scaffolding for the development of higher-level functional features of interactive high-performance mobile apps. We demonstrates the powerfulness of our approach vs. Android's conventional framework by comparing different software metrics.**

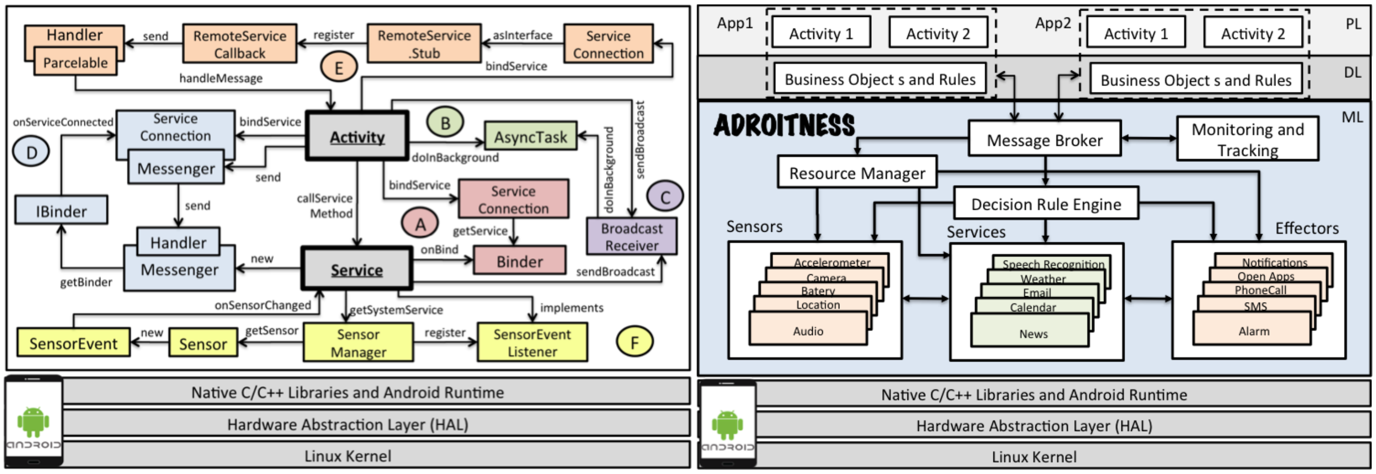
***Index Terms*—android-based middleware; mobile architecture; conversational agent; reusable architectural solutions**

I. Introduction

Mobile technology has been broadly adapted in everyday life activities ranging from business to entertainment domains with increasing demand. This ongoing evolution of mobile computing leads to larger applications and increases the need for methods reducing software complexity [1]. The Android Platform, the most used mobile platform by developers and users [2], provides a software stack that allows building robust, production-quality apps. However, acquiring a deep and proper understanding of the Android Software Stack requires a considerable amount of time for developers (approximately 2+ years [3]) due to the inherent complexity imposed by the over-engineered Android Java Framework (AJF), thereby deviating developers from their main goal: building interactive apps. A common way to deal with AJF's complexity is the use of mobile middlewares that hides the underlying complexity of the environment and masks the heterogeneity of networking technologies to facilitate app programming [4]. Despite of the fact that there exist several middlewares for Android, most of them have a significant performance footprint and imply that developers must learn additional architectural models. The paper is organized as follows: section II presents the motivation and related work, section III presents the architectural model that underlies the development of the middleware; section IV describes the implementation details of the middleware; on section V we present a comparison between our approach vs. the android conventional approach; and on section VI we present our conclusions and future directions on our research.

II. Motivation and Related Work

A. Issues with Android Java Framework (AJF): one of the main issues with Android is that it imposes some design and implementation constructs for components to interact with each other, so the resulting apps becomes top-heavy and over-engineered. Another issue with Android is its concurrency architecture, where invoking a simple network request can be a minefield of subtle problems for which even developers with substantial mobile and Java experience may not be prepared [5]. To illustrate these issues, lets consider an app that sends a network request to a remote service. This simple action spawns several architectural considerations that have to be addressed: 1) Android modifies the user interface and handles input events from one single thread (the main thread) so any task that occupies it for any significant period of time will cause the UI (Android ***Activity***) to become unresponsive; 2) the background tasks should be performed using any of the following Android components: a ***Handler*** (it provides a channel to send data to the main thread), an ***AsyncTask*** (it manages short background asynchronous operations that run on a different thread, so developers would have to deal with thread-safe references and synchronization), a ***Service*** (it has to spawn its own thread in which to do long-running work), or a Java Thread (in this case developers are completely responsible for managing the concurrency). The decision of which kind of component to use depends on several criteria, imposing strong constraints that cannot be verified automatically, e.g., is the background process tied to the UI? is this a long-lasting process? can the process be affected by the activity's lifecycle? is the process shared by multiple components? 3) Android Services define cumbersome mechanisms to communicate with each other, e.g., it is necessary to implement ***Handlers***, ***ServiceConnections*** (an interface for monitoring the state of a Service), ***Messengers*** (implementation of message-based communication across processes), ***Intents*** (an abstract description of an operation to be performed), ***IPC*** (inter-process communication) etc.; and 4) Services and Activities can share data across process boundaries by passing ***Bundles***, objects that implement Serializable or Parcelable interfaces. This process of continuous serialization/deserialization has a significant



(a) Android Java Framework

(b) Adroitness. PL: Presentation Layer, DL: Domain Layer, ML: Middleware Layer

Fig. 1: Android Java Framework (AJF) vs. Adroitness Architectural Model

performance footprint and requires developers to manually parse all the content of these bundles.

B. Android App Frameworks: there are many 3rd party libraries and frameworks for Android indicating that the standard Android APIs are inadequate [6]. One alternative is to use cross-platform (hybrid) mobile frameworks based on web technologies (JS, HTML, CSS). Cross-platform frameworks provide support to scripting languages such as JavaScript, TypeScript or Angular (e.g., Facebook ReactNative, NativeScript and Xamarin) and some others use a web engine to render elements such as HTML5, CSS, and SVG, and execute the logic in a browser instance (e.g., JQuery Mobile, TheAppBuilder, and Apache PhoneGap). Using cross-platform frameworks has advantages such as code sharing between the web and the app, leveraging developers current web language skills, and a plenty of open-source tools available. However, there are some disadvantages regarding fragmentation, compatibility, performance, UX issues, and memory, because they use a full web-rendering engine loaded just for the app and take a lot of GPU/CPU resources increasing the app's response time [7].

C. Requirements: our work mainly focuses on the following requirements 1) the middleware should significantly decrease the amount of effort (person/day) and functional size; 2) it must be latency-sensitive (events that complete in 100 milliseconds or less are believed to have imperceptible latency and do not contribute to user dissatisfaction [8]); 3) it must abstract away the complexity of underlying layers (e.g., communication, concurrency, etc.); 4) it must provide mechanisms for developers to make their apps more modular, pluggable, and easily extensible; 5) it must provide any kind of mechanism for reasoning over the data collected by the smartphone's sensors and services.

III. Architectural Model

Figure 1 illustrates an architectural model comparison between AJF vs. *ADROITNESS* in the development of a conventional mobile app. In Figure 1.a, we have identified 6 different scenarios for Activities and Services to communicate with each other: A) the Service is merely a local background worker running in the same process as the Activity, so developer has to create a Binder class and return it to the Activity so it can use it to directly access public methods available in the Service; B) the Activity uses an AsyncTask to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers; C) Both Activities and Services communicate to each other by sending and receiving broadcast messages through a BroadcastReceiver; D) Activities need to interact with Services running on different processes or apps using IPC, so in this case the developer instantiates the Messenger class inside the Service and defines a Handler that responds to different types of Message objects, also, this Messenger shares an IBinder with a ServiceConnection object, allowing the Activity to send commands to the Service using Message objects; E) the Activity interacts with a Remote Service using AIDL (Android Interface Definition Language) where a RemoteService.Stub object returns an instance of the RemoteService to the ServiceConnection so it can then register callbacks that will monitor the service, then a handler is used to send/receive message objects that implement the interface Parcelable which is used for marshalling purposes; and F) an Activity needs to read data from built-in sensors (e.g., Accelerometer) so it connects to a Service that implements the SensorEventListener, then it gets an instance of SensorManager to register itself and starts listening to particular sensor events. It is worth noting that these scenarios are even more complex since they require additional effort that we have omitted for the sake of simplicity (such as registering Services and BroadcastReceivers on AndroidManifest, allowing permissions, access to native libraries and hardware, etc.). On the other hand, *ADROITNESS* abstracts away the complexity of these 6 scenarios and simplify them to a single mechanism that connects Activities to the underlying Services, Sensors and Effectors (SSE) through a middleware layer that exposes only specific behavior to subscribe, post and receive messages to/from those components. Using the Clean Architecture principles for better separation of concerns and better modularization, *ADROITNESS* allows to decouple the system into well-defined layers such as Presentation layer (i.e., Activities, GUI), Domain layer (i.e., business objects and rules) and Middleware layer

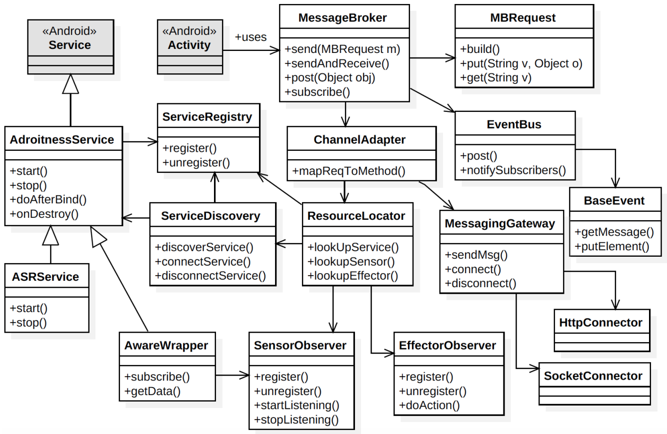


Fig. 2: Adroitness Class Diagram.

(i.e., SSE). The Middleware is divided into four sub-layers: 1) a set of controllers that orchestrate the operation of SSE; 2) a Resource Manager that serves as a service discovery mechanism, resource locator, and dependency injector; 3) a Decision Rule Engine that creates synergies (rules) among those SSE; and 4) a communication layer composed by a Message Broker component and multiple Channel Adapters. This layer uses minimal android dependencies, meaning that no Handlers, AsyncTasks, Messengers, BroadcastReceivers, Binders, nor ServiceConnections are used, instead, a lightweight but powerful concurrency and communication model is proposed, as described in further sections. It is worth noting that the only dependency between app GUI (Activity) and *ADROITNESS* on the class diagram in Figure 2 is the MessageBroker, this centralization reduces complexity and increases maintainability.

A. Sensing and Acting Viewpoint: Sensors allow *ADROITNESS* to detect external changes (e.g., variation in acceleration), user's events (e.g, gestures), and events among phones (e.g., phone1 notifies its proximity to phone2); whereas *Effectors* perform actions as the result of making a decision (e.g, make a phone call). *ADROITNESS* extends the Android SensorFramework (as described by scenario F in Figure 1.a) and adds high-order functions while abstracting away the atomic operations, e.g., the Accelerometer sensor is equipped with a mechanism for detecting free-fall so developers do not need to check whether the 3-axis vector sum is equal to 0.

B. Service-Orientation Viewpoint: since sensors' and effectors' extensibility is limited by phone's hardware, we enhanced them by using *ADROITNESS Services* (which extend Android Services), that is, application components that perform discrete functions either locally (in the phone) or remotely (on a server). *ADROITNESS* was designed based on a Service-Oriented Architecture (SOA) in order to promote loose coupling between services. We defined a Resource Manager pattern in charge of: 1) maintaining a service registry which contains information about how to dispatch requests to services; 2) carrying out service discovery operations by using a resource locator pattern; 3) registering pluggable services that can be added or removed dynamically by using dependency injection; and 4) executing a Service Manager that controls the services lifecycle (start, destroy, bind, etc.). Our SOA architecture is empowered

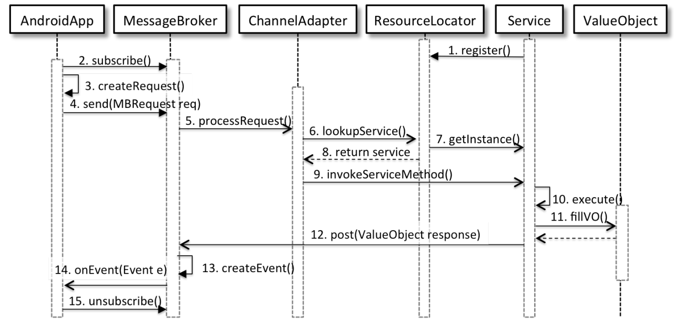


Fig. 3: Sequence Diagram for end-to-end message passing.

by the use of an event-driven mechanism that allows fast decoupled interaction between Android Services and Activities. *ADROITNESS* provides a set of pre-defined pluggable services (e.g, weather, calendar, email, Automatic Speech Recognition – ASR, access to Knowledge Bases, just to name a few, but developers can extend this set of services and add customized services that are hooked into the middleware.

C. Messaging and communication Viewpoint: Message Broker: this component is in charge of routing, transforming, aggregating and decomposing messages. MessageBroker makes transparent the communication between activities and SSE, that is, developers only have to create a request (MBRequest) instance and pass it to the message broker, then it will deliver the request to the corresponding SSE. Finally, the message broker uses an event bus to perform this communication through a publish/subscribe mechanism. Channel Adapter: it acts as a messaging client to the messaging system and invokes *ADROITNESS* functions via a service-supplied interface. Figure 3 illustrates the the interaction among these classes.

E. Decision Rule Engine Viewpoint: the Decision Rule Engine (DRE) is a rule-based system in charge of creating, validating, executing and specializing rules created by the user or developers [9]. The DRE extends the *ADROITNESS*'s scope by aggregating different SSE, that is, services that can accomplish more complex processes and provide higher-level abstractions that allow developers to easily assemble entire use cases and interactions. A Rule is formally defined as a set of conditions so that `<left-side><operator><right-side>` and a set of actions so that `<event-action>:<component>:<method>:<params>`. The three rules on Listing 1 illustrates a use case for the free-fall scenario, where the phone's alarm is triggered when the accelerometer sensor detects the phone is free-falling.

```
RULE: Rule1
  IF      Event.what equals Sensor.ACC.motion
  THEN    Event.post : Sensor.ACC : doVectorSum : [ACC.3-axis]
RULE: Rule2
  IF      Event.what equals Sensor.ACC.doVectorSum
  AND     Sensor.ACC.vectorSum equals 0
  THEN    Event.post : Sensor.ACC : freeFall
RULE: Rule3
  IF      Event.what equals Sensor.ACC.freeFall
  THEN    Event.post : Effector.ALARM : ring : [notification]
```

Listing 1: Rules free-fall use case

D. Concurrency Viewpoint: in order to improve *ADROITNESS*'s latency footprint, throughput and interactivity, we defined a clean concurrency model that radically eliminate the use of over-engineered AJF constructs and replace them with a pure Thread-based model that uses thread pools and async executors. Also, it uses a message-passing mechanism (the event bus) to pass messages between threads instead of sharing or accessing objects simultaneously, that way it is not necessary to protect the code by using locks, monitors and synchronized blocks that are computationally expensive.

IV. Implementation

In this section, we provide some details about the libraries we used for the implementation of *ADROITNESS*[1]. We use the GreenRobot's EventBus framework [10], an Android optimized event bus that simplifies communication between AJF components by decoupling event senders and receivers, removing dependencies, and using a pub/sub pattern for loose coupling. Also, we used ZMQ messaging framework for communication with external servers. ZMQ is a high-performance asynchronous messaging library aimed at use in distributed and concurrent applications with minimal latency footprint. Using this library, *ADROITNESS* guarantees extremely low-latency responses, even when external servers, thanks to it access sockets directly. Using ZMQ, we could abstract away low-level communication details, such as dealing with different socket types, connection handling, framing, and even routing. Finally, we extended the set of plug-ins provided by AWARE [11], a framework dedicated to instrument, infer, log and share mobile context information. For instance, AWARE provides an effector for processing TTS (text-to-speech) outputs, however, it lacks of a sensor for processing ASR inputs. *ADROITNESS* not only includes an effector for TTS but also provides a extensible API that allows to plug different kind of ASR implementations (e.g., Google ASR, Microsoft Bing, CMU Pocket-sphinx, etc.)

V. Evaluation

We performed an empirical metric-based comparison between *ADROITNESS* and AJF in the development of a conversational agent [12], [13]. For latency experiments, we measured the time for sending and receiving 1, 10^1, 10^2, 10^3 and 10^4 messages. In general, *ADROITNESS*'s performance surpassed AJF's performance in a high rate when sending/receiving 1 message ($\cong 95\%$) and then gradually decreased while the number of messages increased but still surpassing AJF at a rate of $\cong 54\%$. We also measured the *Cyclomatic Complexity* of both approaches, which is defined as the number of linearly independent paths within a graph that represents the source code flows. Based on our analysis we deduced that both *ADROITNESS* and AJF have low complexity, however, the improvement rate demonstrated that *ADROITNESS* reduced the complexity on $\cong 31\%$ in comparison with AJF. In terms of size metric, we used Function Points (FP), a widely accepted industry standard for functional sizing. Based on the results we could observe that the app is $\cong 53\%$ smaller in functionality when using *ADROITNESS* instead of AJF (see Figure 4), which in turn reduced the estimated amount of effort (a team of 5 persons would take 54.6 days if using AJF while taking 25.8 days if using *ADROITNESS*).

[1] https://github.com/ojrlopez27/Adroitness-Mobile

```java
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        MessageBroker mb = MessageBroker.getInstance(this.getApplication());
        mb.subscribe( subscriber: this);
        mb.send( caller: this, MBRequest.build(Constants.MSG_START_ASR));
        //...
    }

    public void onEventMainThread(final NLGEvent event) {
        System.out.println("NLG output: " + event.getOutput());
    }
    //...
}
```

Fig. 4: Code snippet for a two-steps use case realization using *ADROITNESS* (in the 1st step, ASR is started on GUI's onCreate method, in the 2nd step, a NLG -Natural Language Generator- event is handled by onEventMainThread method). *ADROITNESS* requires 15 lines of code (loc) while AJF more than 750 loc.

VI. Conclusions and Future Work

In this paper, we have presented *ADROITNESS*, an architectural middleware to support the construction of mobile apps. Our contributions are: a) we used the clean architecture approach to guarantee a better separation of concerns; b) middleware's architecture abstracts away the low-level design and implementation details such as communication and concurrency model; c) latency was improved by avoiding the use of overengineered AJF components and replacing them by a lightweight threading model, and a high-performance cache instead of using serialization/deserialization mechanisms; d) a Decision Rule Engine that binds SSE and facilitates the composition of more complex behaviors; and e) we demonstrated that *ADROITNESS* reduced the complexity, size, and effort of apps implementation while improving the performance. For the future work, we will create a semantic layer in order to improve the service discovery process, provide more accurate and relevant information to higher-level layers, and make inferences about user's context. Also, we will implement a machine learning mechanism to discover and refine the rules orchestrated by DRE.

Acknowledgment

This research was supported in part by Yahoo! and Verizon through the CMU-Yahoo InMind project.

References

[1] J. Dehlinger, "Mobile application software engineering: Challenges and research directions," in *Mobile Sw. Eng.*, vol. 2, 2011, pp. 29–32.

[2] [Online]. Available: https://www.gartner.com/newsroom/id/3859963

[3] [Online]. Available: https://www.infoq.com/news/2010/07/Mobile-Survey

[4] Z. Sanaei, S. Abolfazli, A. Gani, and R. Buyya, "Heterogeneity in mobile cloud computing: taxonomy and open challenges," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 369–392, 2014.

[5] Z. Mednieks, G. B. Meike, L. Dornin, and Z. Pan, *Enterprise Android: Programming Android Database Applications for the Enterprise*, 2013.

[6] S. Barnett, R. Vasa, and A. Tang, "A conceptual model for architecting mobile applications," in *Conf. Software Architecture*, 2015, pp. 105–114.

[7] J. B. Jorgensen, B. Knudsen, L. Sloth, J. R. Vase, and H. B. Christensen, "Variability handling for mobile banking apps on ios and android," in *13th IEEE/IFIP Conference on Software Architecture*, 2016, pp. 283–286.

[8] B. Shneiderman, *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India, 2010.

[9] L. Tomazini, O. Romero, and E. H., "An architectural approach for developing intelligent personal assistants supported by NELL," in *ENIAC'17*.

[10] [Online]. Available: https://github.com/greenrobot/EventBus

[11] [Online]. Available: http://www.awareframework.com/

[12] Y. Matsuyama, R. Zhao, O. Romero, and et al., "Socially-aware animated intelligent personal assistant agent." in *SIGDIAL*, 2016, pp. 224–227.

[13] O. J. Romero, R. Zhao, and J. Cassell, "Cognitive-inspired conversational-strategy reasoner for socially-aware agents," in *26th Int. Joint Conf. on Artificial Intelligence, IJCAI*, 2017, pp. 3807–3813.