Background of OpenHarmony

OpenHarmony is designed with a layered architecture. As illustrated in Fig. 1, it consists of four layers. From bottom to top, the four layers are (1) the Kernel Layer, (2) the System Service Layer, (3) the Framework Layer, and (4) the Application Layer. We now briefly detail these four layers to help readers better understand this work.

Kernel Layer. The kernel layer of OpenHarmony contains two main sub-systems, namely a kernel sub-system that powers an operating system kernel (such as the Linux Kernel) for scheduling the software execution of the whole system and a driver sub-system that is responsible for connecting the software stack with the various hardware. Observant readers may have noticed that, unlike other systems, there is a special component called the Kernel Abstract Layer (KAL) in the Kernel Layer of OpenHarmony. This component is indeed a special OpenHarmony feature that is designed to support multi-kinds of mobile devices. For different devices, OpenHarmony may select different OS kernels (e.g., Linux or LiteOS) to power the system. KAL is proposed to mitigate such a difference, aiming at offering the same capabilities for the upper software layers.

System Service Layer. The system service layer is the core part of OpenHarmony that provides the actual implementation of all the system services required to run OpenHarmony apps. Except for supporting basic capabilities such as the ones related to security control or providing intelligent functions, it also includes components related to common software services such as Events and Notifications, device-specific software services such as the ones dedicated to IoT devices or wearable devices, as well as hardware-related services such as sensors and location services.

Framework Layer. The framework layer provides an interface for developers to implement OpenHarmony applications and such an interface is often provided within a Software Development Kit (SDK). As shown in Fig. 1, generally speaking, this layer provides similar capabilities as the system service layer. However, this layer is specifically required as it keeps app code from directly accessing system services, which might be abused by third-party developers if not controlled. Indeed, through the framework layer, system services do not need to be exposed to third-party developers and how they should be called or scheduled can be pre-defined. This layer is also very important as it defines the set of APIs needed to be seen by third-party apps. This set of APIs needs to be appropriate as defining fewer APIs may make the implementation of OpenHarmony apps difficult while defining more APIs would increase the complexity and subsequently the maintainability of the framework.

Application Layer. The application layer is the place where OpenHarmony apps are located. There are two types of apps: system apps and third-party apps. The former one should be provided by OpenHarmony itself, covering the basic functions that allow the OpenHarmony system to be practically usable. The latter ideally should be supported by third-party developers that help the Openharmony system to provide a good user experience, which is the key to the success of the OpenHarmony ecosystem.

0.1 The App Development Framework

We now briefly introduce OpenHarmony's app development framework. There are actually two versions of app development frameworks supported by OpenHarmony to develop third-party apps:

2025. XXXX-XXXX/2025/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

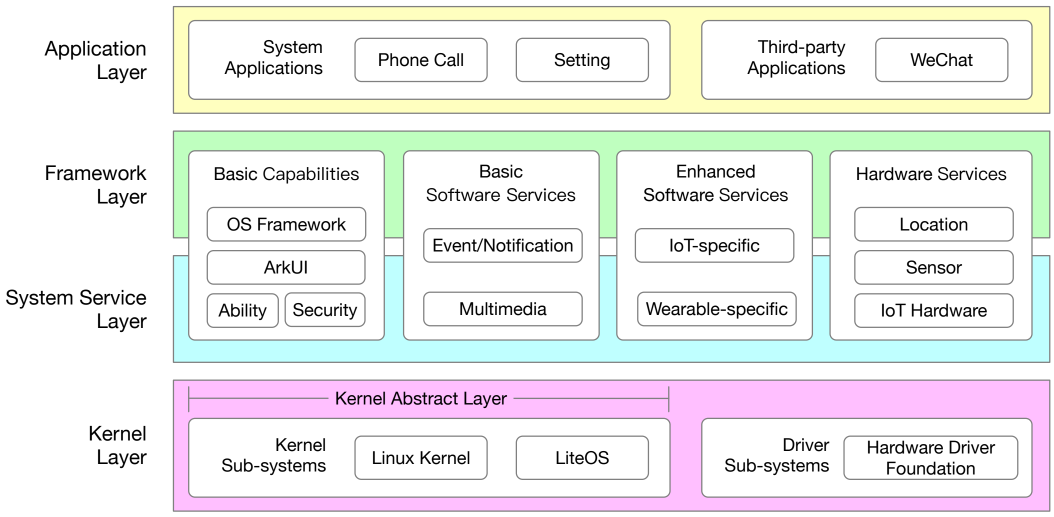

Fig. 1. The Software Architecture of OpenHarmony.

one based on Java program language and another based on ArkTS program language.[1] Since the Java version will be gradually replaced by the ArkTS version, in this work, we will only focus on the ArkTS version. Fig. 2 highlights the core components of the ArkTS-based OpenHarmony app development framework. OpenHarmony actually supports two ways of app logic (named Ability) developments, namely the FA (Feature Ability) model and the Stage model. The Stage model is newly introduced (since API version 9) to replace the FA model. Hence, in this work, we will only focus on the Stage model.

Stage-based Ability Framework. As shown in Fig. 2, in the Stage model, an OpenHarmony app is made up of *AbilityStage* components. Each *AbilityStage* should contain one or more *Ability* components. In OpenHarmony, there are two types of Ability components: UIAbility and ExtensionAbility. *UIAbility*, like *Activity* in Android, is responsible for implementing the app's visual parts (i.e., GUI pages). This is the reason why *UIAbility* component will include a *WindowStage* component that further contains a *Window* module with an ArkUI page attached to it.

For other functions that are not directly relevant to the app's UI pages, OpenHarmony has introduced the so-called *ExtensionAbility* component to support their implementation. Normally, in Android, such functions should be implemented in one of the following three types of components: *Service*, *Broadcast Receiver*, and *Content Provider*. In OpenHarmony, the *ExtensionAbility* mechanism provides a more fine-grained way to implement such functions. For example, *ServiceExtensionAbility*, a sub-class of *ExtensionAbility*, is designed to support background tasks, providing equivalent functions as that of *Service* in Android. Another sub-class of *ExtensionAbility*, namely *DataShareExtensionAbility*, is designed to support data sharing, providing equivalent functions as that of *Content Provider* in Android.

Like what has been designed in Android's components, there are lifecycle methods designed in OpenHarmony's ability components. Fig. 3 illustrates the lifecycle of OpenHarmony's UIAbility component, which by itself contains four states, namely Create, Foreground, Background, and Destroy. *Create* state is at the stage when a UIAbility is started. At that time, the system will call the

[1]ArkTS (also known as eTS) is the preferred programming language introduced by Huawei to develop OpenHarmony applications. It is extended from the famous TypeScript language.

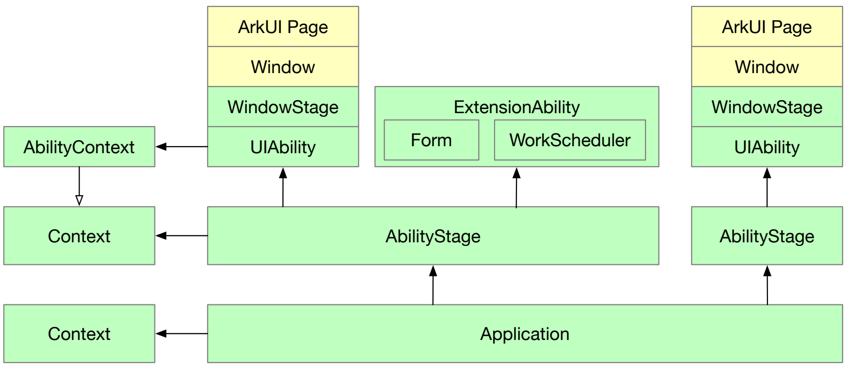

Fig. 2. The Architecture of ArkTS-based App Development Framework (Stage Model) of OpenHarmony.

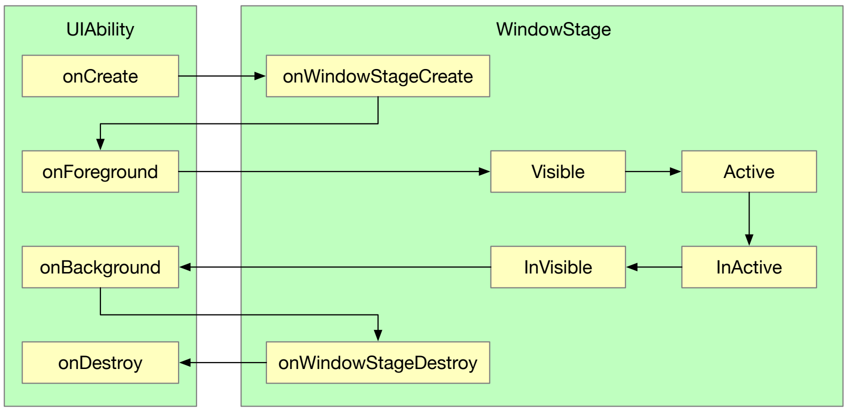

Fig. 3. The Lifecycle of OpenHarmony's *UIAbility* component.

corresponding *onCreate()* callback method, in which certain resources could be initiated. After the *onCreate()* method is called, the state moves to *Foreground* and the *onForeground()* callback method will be invoked. At this stage, the UI page of the UIAbility becomes visible and will be displayed to users. Once the UI page becomes invisible (e.g., other UI pages become visible), the state will be moved from *Foreground* to *Background*. At this time, the *onBackground* callback method will be called. When the UIAbility is going to be terminated, the *onDestroy()* callback method will be invoked and this is the place to store relevant data and free requested resources. Observant readers may have noticed that the lifecycle of UIAbility is associated with a *WindowStage* component, which per se has a sequence of lifecycle methods to be invoked as the UIAbility's state goes by.

ArkUI Module. As shown in Fig. 1 (with yellow background), the actual UI pages are implemented through the so-called ArkUI framework. ArkUI is a core module of ArkTS that is newly introduced to support UI developments. Fig. 4 illustrates the architecture of the ArkUI module. This module supports two ways of UI implementation. The first way is to leverage Web-based technicals (e.g., HTML, CSS, Javascript) and the other way is through the so-called declarative programming (specifically designed to support the implementation of OpenHarmony apps). This module also includes an *UI Engine* module to provide common UI-related functions and other modules to allow visual display of UI pages.

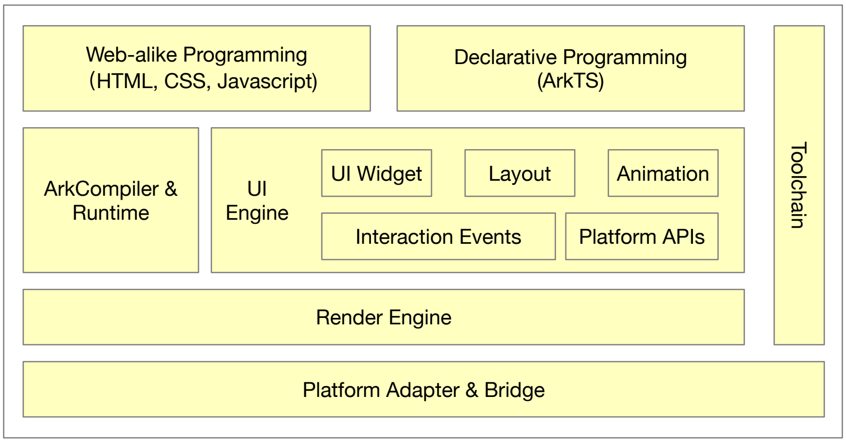

Fig. 4. The Architecture of OpenHarmony's ArkUI Module.