Distant, Yet Present: Remote View Architecture on iOS & macOS

https://github.com/NSAntoine - Osama Al Hour

Introduction

As Apple's Operating Systems continue to get more interactive and complex, the need for a secure way of displaying content in one process from another process has become extremely important. Remote Views on iOS and macOS provide a flexible, secure, and performant mechanism for such a task, an implementation trusted to the point where Apple's willing to bet even your most sensitive data wouldn't be leaked by it.

In fact, you've definitely interacted with Remote Views before on a daily basis, when an application on macOS presents the file picker, that's a Remote View. When an application presents the photo picker on iOS, that's a Remote View. However, client applications of Remote Views, as well as users interacting with them, don't *know* they are Remote Views, this information is entirely abstracted away by the System, despite the fact that the View is rendered and managed by an entirely different process.

This writeup seeks to explore the how Remote Views are implemented, their model and archetecture, why they are so important from a security and privacy standpoint, as well as how the System interacts with them.

Hierarchy

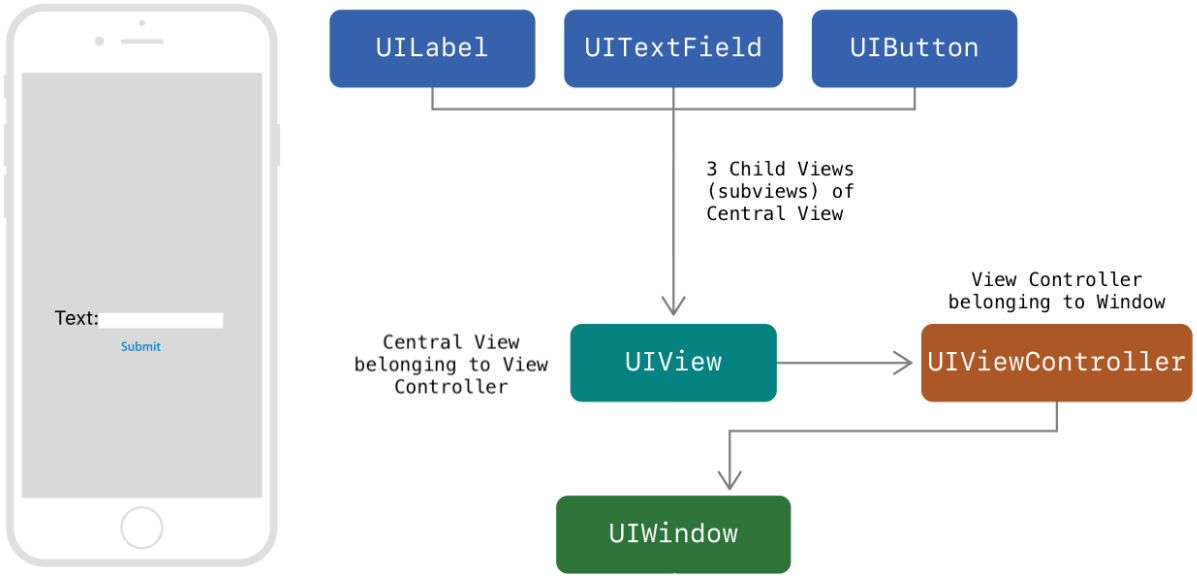
To understand the need and use of Remote Views, we must first understand the hierarchy of User Interface on iOS and macOS (if you are already an iOS / macOS developer, treat this as a refresher). Each User Interface element on both platforms, such as text, buttons, lists, images, etc. are made up of an object called a View (on iOS: `UIView,` on macOS: `NSView`), which is the basic building block of all user interface elements. Usually, specific elements will subclass the `UIView/NSView` base class and build on top of it, some notable examples include:

- `UIButton`: Provides a basic interface element of a button which performs an action when clicked. Subclass of `UIControl`, a subclass of `UIView` that allows views to respond to user interaction (i.e., buttons and toggles)
- `UILabel`: Provides a basic interface element which renders text. Subclass of `UIView`
- `UIImageView`: Provides a basic interface element which renders an image. Subclass of `UIView`
- `UITableView`: Provides a basic interface element which renders a simple list of items (i.e., the list used in the system Settings app). Subclass of `UIScrollView`, which is a subclass of `UIView` which provides scroll/zoom capabilities
- …and many, many, many more…

Views can also add child Views, called subviews (for example, the subviews of a View that represents a user's social media profile would be a label for the username, an image view of the user's banner, etc).

Views don't just exist by themselves, however, you can't have an app with just Views and nothing else. When an application is initialized, it creates a window, that window contains what is called a *View Controller*. A View Controller has a singular, central View, and any other views which the application wishes to display to the user are to be added as subviews of that central View. This flow of a window, view controller(s), and views is called the "View Hierarchy" of an application.

The image below demonstrates the View Hierarchy of a simple application with a label, textfield, and button



The View Hierarchy is easily accessible within the application, meaning that it's possible to find the instance of just about any view in memory and manipulate it in just about any way (hiding/showing it, changing it's size, animating it, etc.) and capture a snapshot of the View as an image that could be saved later.

You may be wondering, how is this important for Remote Views? Why would you supply a View Controller rendered through another process? We'll be covering that now with the following section.

What even is a Remote View in the first place? And why?

When an application presents a View Controller which it has implemented itself or imported from a library/framework, that View Controller is initialized *in the app*, managed *by the app*, and rendered *by the app*. In other words, the View Controller is completely controlled by the application, where it's free to manipulate it'd like. For the majority of User Interface, this is fine and well (i.e., it makes complete sense for a social media app to be in complete control over the View Controller which displays the user's timeline).

However, there are a variety of use-cases where the system would like to provide an application a View Controller to present to the user for a specific use-case, but control and manipulation of that View Controller by the application (i.e., an application editing the Views of the View Controller, or reading the data from a list view in the View Controller) would not be acceptable.

This is where Remote Views come in: A Remote View Controller[1] is a View Controller which the user application can show and present to the user just like a normal View Controller, however the View Controller is entirely managed and rendered by another process, meaning that the user application has *no control* over the Remote View Controller. The user application cannot read the data in the Remote View Controller (i.e, items in a list shown in the View Controller), it cannot change the Views in the Remote View Controller (nor can it change properties of Views in the Remote View Controller), it cannot screenshot the Remote View Controller, nor can it make any modifications to the Remote View Controller.

[1] Though this writeup uses the term "Remote View", user applications are *almost always* going to be interacting with Remote View Controllers (always some subclass of _UIRemoteViewController) rather than Remote Views directly. However, there is virtually no distinction for the sake of this writeup.

System Use of Remote Views

As a result of how Remote View Controllers are designed (see page above), user applications can prompt the user to select potentially sensitive information in a Remote View Controller without that information being exposed to the application besides information that the user has decided to explicitly share with the app.

For example, starting with iOS 14, the System provides a View Controller to applications which allows the user to select certain photos / videos to share with a user application without granting the application access to the entire photos library. Both the user and the application benefit, as the user can grant the application access to only the photos / videos they choose, and the application doesn't have to request permission from the user to access the photos library.

Since the picker is a Remote View Controller, the user application cannot modify the picker *arbitrarily* (there's still some customizability, to be discussed later in this writeup, see "Communication" section), nor can it snapshot the picker to view the photos / videos that the user can select, the only thing that the user application can do is present the picker, send the configuration it wants (such as how many items it wants), get access to photos / videos that the user has decided to share with the application, and dismiss the picker.

Below is a table of notable Remote View Controllers on iOS, note that this is ***not an exhuastive list***.

UI Element	Public View Controller	Remote View Controller Renderer Process
Photo / Video Picker	PHPickerViewController	PhotosPicker.appex (embedded in the bundle of the stock Photos app, MobileSlideShow.app)
Contact Picker	CNContactPickerViewController	ContactsViewService.appex (embedded in the bundle of ContactsUI.framework)
Share Sheet	UIActivityViewController	com.apple.UIKit.ShareUI.appex (embedded in the bundle of UIKit.framework)
Apple Pay payment sheet (when paying for something with Apple Pay in an application)	PKPaymentAuthorizationViewController	PassbookSecureUIService.app (Standalone Application, concept is to be discussed in this writeup)

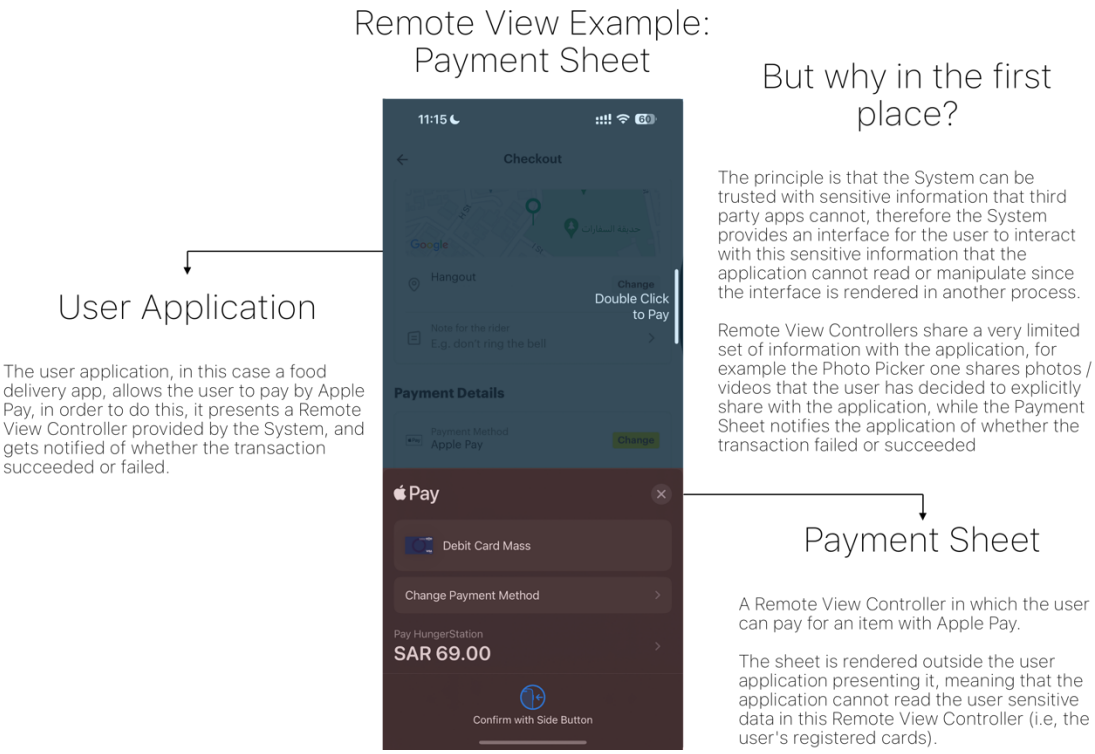
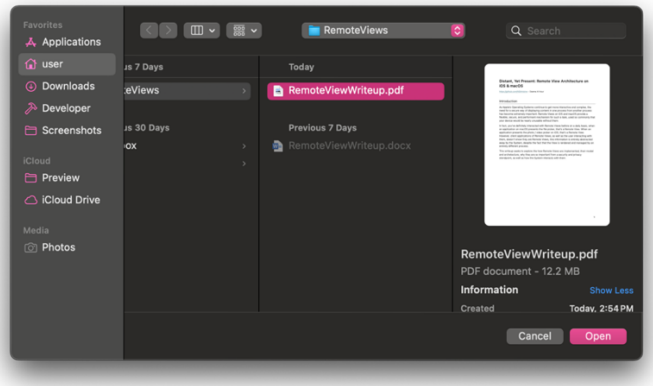


Figure 1: Payment Sheet Remote View Controller dissection

Remote View Controllers are also used extensively on macOS, such as with the file explorer panel:

File Selector Panel (NSOpenPanel, NSSavePanel)



When an application launches this panel, it is fully rendered and managed by a system process rather than the application.

This means that applications cannot read / write to files visible in this panel, except ones which the user explicitly decides to share with the application by pressing the "Open" button.

Figure 2 File Panel Remote View

Model & Architecture

Remote View Controllers follow a simple model where there's 2 processes:

- The process requesting and using the Remote View Controller (called the "Host application")
- The process rendering and providing the Remote View Controller (called the "View Service")

The 2 processes communicate through XPC, Apple's primary framework for IPC, A View Service is implemented as one of the 3 following types of bundles:

- Application
- App Extension
- XPC Service (only macOS uses XPC Services for View Services, from what I know)

XPC Services are just normal executables which receive requests to perform specific actions (in the case of XPC Services acting as View Services, the requests they receive are to provide a View which is rendered by the service). App Extensions are plugins which extend functionality of existing apps (for example, Widgets are implemented as app extensions), and are commonly used as View Services (Photo / Video Picker and Contacts Picker View Services are implemented as app extensions on iOS). The third and last way a View Service is implemented is as a standalone application, however, View Service applications are a lot more different than normal applications, as they're strictly meant to be binaries that provide View Controllers to requesting user applications, they're not meant to be launched through convential means, and are therefore hidden from the user (i.e., they don't show up on the user's Home Screen, or in Spotlight when searching for apps, etc.).

When a host application requests a Remote View Controller, the View Service is activated and provides the Remote View Controller asynchronously, the View Service is allowed to refuse the host application the Remote View Controller it asks for (hence why it's called a "request").

It is important to note that the hosting application does not directly use the instance of the Remote View Controller from the View Service, rather, the hosting application creates a parent View Controller (who's actual view contents are empty when initialized) which requests the Remote View Controller and adds the Remote View Controller to it's view hierarchy.

The parent View Controller which requests and displays the Remote View Controller is ***always*** implemented by a System framework / library, meaning that developers of hosting applications never have to know about Remote View Controllers, those are all implementation details for Apple themselves to implement.

The following diagram demonstrates how the Photos Picker View Controller, which relies on a Remote View Controller, is instantiated:

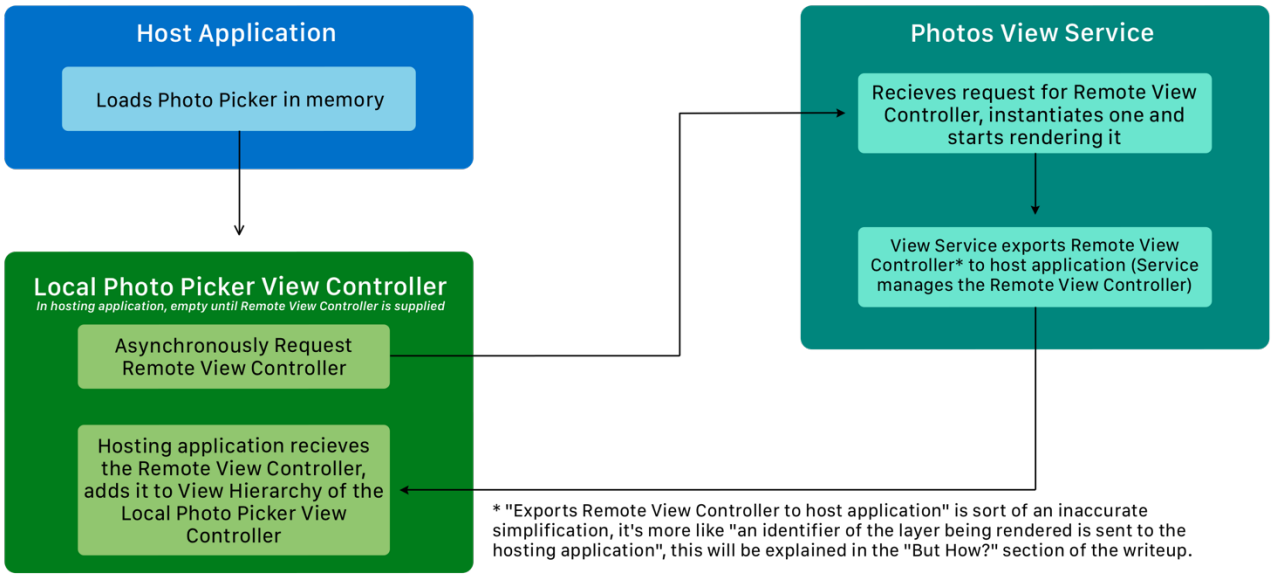


Figure 3 Diagram demonstrating the process of requesting the Remote View Controller for the Photo Picker

Communication

View Services almost always need to relay some information back to a host application using a Remote View Controller. For example, the Remote View Controller for the Payment Sheet needs to signal back to the host application that the transaction either succeeded or failed, and the Remote View Controller for the Photos Picker needs to provide the photos / videos that the user has explictily decided to share with the host application, so how is this achieved while still preserving the model of rendering content entirely in the View Service and preventing the host application from arbitrarly reading / writing user-sensitive content displayed in the Remote View Controller?

Recall earlier that View Services and Host Applications communicate through XPC, Apple's primary framework for IPC. In the case of View Services and Host Applications, The Objective-C API for XPC is used rather than the traditional C API, the traditional C API allows for 2 processes to communicate with each other by sending and receiving a hashmap (`xpc_dictionary_t` objects), while the Objective-C API for XPC works by having both processes supply "interfaces" (list of functions that the other side can call) to the other, each process gets access to the interface of the other process and can send a signal to the other process to perform a function from the obtained interface. This explanation can sound a bit confusing, so here's a diagram to explain it:

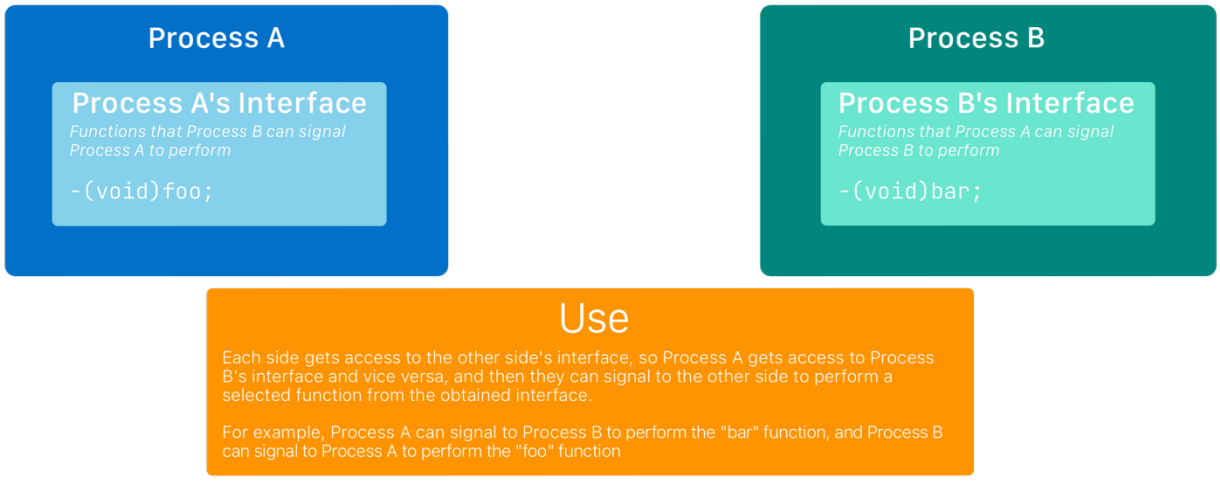


Figure 4 Diagram explaining Objective-C XPC API Interfaces Model

In the case of Remote View Controllers, both the Host Application and the View Service provide interfaces, the Host Application provides an interface as a delegate to get notified of changes by the View Service, and the View Service provides an

interface to allow the Host Application to make changes or configurations that the View Service allows.

For example, for the photo picker, the View Service implements an interface where the Host Application can signal to zoom in / zoom out the picker view, as well as changing the configuration (i.e., are only photos allowed? How many photos / videos should the user be allowed to pick?), and the Host Application implements an interface where it gets notified of when the user has selected photos / videos that it wants to share with the application.

This model allows some interaction and customizability without gaining access to the underlying Remote View Controller and therefore guarantee that data displayed in the Remote View Controller is still not accessible by the Host Application.

But How?

So how do Remote View Controllers even do this? How can you host content that is rendered in an entirely different process? What magic is going on here?

Recall in the Hierarchy section that we discussed that each user interface element are made up of objects called Views. Views themselves are backed by an object called the *layer*, the layer manages the majority of a View's appearance on screen, such as:

- Background Color
- Animation
- Geometric properties (size, position, transform)
- Shadow
- Corner Radius
- Hidden / Visible status
- …and more properties…

Normal (non-remote) Views use the standard `CALayer` class on both macOS and iOS, which contains the properties listed above. Hosting Applications displaying Remote Views use a subclass of the `CALayer` class, called `CAHostLayer`. A host layer displays content rendered from another process. In the case of Remote Views, A host layer is used by the hosting application to display content which is rendered in the View Service. The way this works is that the View Service issues a 64-bit integer identifying the layer it's rendering, this identifier is called the Context ID.

The View Service sends the Context ID to the Hosting Application, and the Hosting Application initializes a `CAHostLayer` then sets the host layer's `contextId` property to the one sent by the View Service.

When a `CAHostLayer` registers a `contextId` once it receives one from the rendering process (in our case, when the hosting application receives the context ID from the View Service), a bridge is formed between the hosting application and the View Service, then the View Service starts sending objects to the hosting application called IOSurface objects, which are framebuffer objects (data in memory which represents pixels in a video frame), the hosting application's `CAHostLayer` then uses those IOSurface objects to display content rendered by the View Service.

IOSurface objects are actually created and managed by the *kernel*, not the View Service itself (the View Service just requests the kernel to create one). Since the IOSurface object is created and managed by the kernel, it means sending the

IOSurface object is almost guaranteed to be secure since the kernel prevents unauthorized access to memory between different processes.

In order to prevent fake touches by applications (i.e., a malicious application may present the Photo Picker View Controller then simulate a fake touch on the software side to fool the system into thinking the user explicitly selected a photo / video to share with the application), synthetic touch events created by the host application are contained to View Controllers of the host application itself so they never reach the Remote View Controller of the View Service, and real touch events are re-routed by the System (specifically, a System process called backboardd) so that touch events made to the Remote View Controller never reach the host application in the first place and instead reach the View Service.

Thanks to & Resources

- JxBrowser: Cross-Process Rendering using CALayer
  https://teamdev.com/jxbrowser/blog/cross-process-rendering-using-calayer/
- Chromium Source Code: (Demonstrates usage of CALayerHost)
  https://github.com/chromium/chromium
- Class dumps of _UIRemoteView, _UIRemoteViewController, and CALayerHost:
  https://headers.dfiore.xyz
- RemoteViewFun: Research - Making a custom _UIRemoteViewController
  https://github.com/pixelomer/RemoteViewFun
- Ole Begemann: Remote View Controllers in iOS 6
  https://oleb.net/blog/2012/10/remote-view-controllers-in-ios-6/