CISC 322/326 Assignment 1: Report

Kodi: A Conceptual Architecture Analysis

October 22, 2023

Team Torrent (Group 25)

Aselstyne, Alex (alex.aselstyne@queensu.ca)

Dinari, Daniel (20dd29@queensu.ca)

Nagel, Jake (20jn29@queensu.ca)

Peterson, Jack (21jrp10@queensu.ca)

Pleava, Ryan (20rcp5@queensu.ca)

Abstract

This paper analyzes the conceptual architecture of the software system *Kodi*. As a multimedia player application, Kodi is made of many interacting components which form its architecture. Though it started as a simple Xbox media player application, it has evolved into one of the most popular software media centers available. The application is available on a variety of platforms and allows the user to play back a wide variety of media content from their audio, video, and photo personal libraries.

Fundamentally, Kodi can be broken down into 9 broad subcomponents, which each rely and depend on many of the others. These subcomponents control aspects like the user interface, file loading, optional addons, and web request management, among other things. Several architectural styles can be applied to these components as well, which allow the software to be likened or compared to other applications. A couple of use cases are explored and explained further in the paper using sequence diagrams to give an overview of how components interact with each other over time.

Over the course of the project, the team faced a number of issues, and grew through these to produce a polished final product. Upon completion of the paper, the team felt we had achieved a realistic and complete model of the architecture underlying Kodi, while maintaining a focus on important components and avoiding minutia. Overall, we determined that Kodi's architecture is robust in nature and is well-able to support the wide variety of usages that the software provides to the user.

A few next steps are provided at the end of the report, including breaking the system down into smaller subcomponents for a holistic view, performing performance analysis and then refinement on the proposed architecture, and analyzing the security implications of all architectural decisions proposed. These next steps would allow for potential development of the Kodi software system to proceed with ease and would minimize errors discovered later in the development pipeline.

Introduction and Overview

Kodi is a free and open-source multimedia center designed to be user friendly and all encompassing. The software is managed by the Kodi Foundation, though the bulk of the development is done by Team Kodi, made up of members of its contributor community. It is available for a multitude of platforms, including iOS, Android, Windows, Mac OS and Linux, and supports a variety of input devices, including mouse and keyboard, game controllers, and touch screens. Kodi has support for a variety of video, audio, and image formats.

The Kodi project was originally developed for the Xbox (2001) as a piece of "homebrew software," or software that is not officially endorsed by Microsoft. Users required a modified console to install the software, which was called XBMC (**XB**ox **M**edia **C**entre) at the time of release in 2003. Over the initial 10 years following its release, it was ported to numerous other systems, including Linux, Windows, Mac OS, Android, and iOS. In 2014, the software was renamed Kodi, to dissociate the software from its Xbox roots.

As a media centre, Kodi does not come bundled with any media, rather the user must provide it themselves. This media can either come from a local disk, another data source on the local network, or a remote server. Add-ons can be used to add additional media sources. These include popular media services such as YouTube, Twitch, Plex, and Spotify. Kodi then handles the entire media playback pipeline, delivering the content to the user for viewing.

Given the software's complexity, Kodi has several different internal components. This includes a GUI Manager, Player Core, Renderers, Local File Manager, Request Manager, and a variety of Addons. These components have many dependencies on each other, and few components would be able to function standalone. Their interactions can be generally described using a couple of architectural styles: Pipe and Filter, Repository, and Client-Server.

This paper aims to describe the conceptual architecture behind Kodi abstractly, without exploring any of the concrete implementation. This will be covered in 5 sections: 1) a derivation process outlining how we arrived at our architecture, 2) an analysis of architectural styles used, 3) a deep dive into the conceptual architecture, 4) use case diagrams showing data flow, and 5) a summary of lessons learned in the project.

Derivation Process

Initially, we discussed possible architectural styles for Kodi. From the architecture outlined on the Kodi wiki, we quickly came to an agreement about a repository style for accessing local data and a client-server for downloading addons and related data. With some prior knowledge about video players, we decided on a pipe and filter system for rendering. We then decided on two use cases that would utilize most of the components of the Kodi app. Our first use case pertained to playing a video from a local file, and our second use case involved downloading and using a plug-in from the Kodi server. These two use cases were selected due to involving many components, and the separate use spaces occupied - the first use case uses local files, and the other use case, a server.

Two primary strategies were used to derive the sequence diagrams and conceptual architecture components. Performing the use case in the Kodi application and cross-checking the process with the architecture from the Kodi Wiki. Walking through the use case sequence diagrams provided an outline of GUI interactions with other components and the operating system, and the use case components were matched or added from the Kodi Wiki architecture.

The Kodi Wiki presents the application architecture in four primary layers, the client layer, the presentation layer, the 'business' layer, and the data layer. Each layer contains modules, and from those modules the necessary components for the sequence diagrams and component diagrams were made. The interactions were hypothesized by utilizing the Kodi app to simulate the actions being carried out in each use case.

The draw the sequence diagrams, we used one primary writer for each use case, and the other members discussed. Collectively we employed the two strategies outlined above to create each component and determine their interactions. Once the group agreed and finalized the

existence of a particular component or interactions, we would add or remove them from the paper copy. Designating one writer allowed the entire group to think analytically about the sequence diagram and bounced ideas around, while components were consistently recorded without logical integrity issues.

The components for the "Conceptual Architecture" section and box-and-arrows diagrams were then derived from the sequence diagrams. This allowed us to have concrete examples of how the components interacted. By choosing two use cases that covered nearly all areas of the Kodi system, we were able to capture nearly all interactions using this method alone.

Architectural Style(s)

Below we outline three architectural styles used in Kodi's conceptual architecture. The pipe and filter style is used for audio/video rendering, the repository style is used for file gathering, and the client-server architecture is used in the addon framework. Each architectural style and its application to Kodi is explored in depth below.

Pipe and Filter

The Pipe and Filter architectural style involves a series of "filters" which individually process an input stream of data and pass the data down the line using "pipes." The pipes are simply streams for data to flow through in small chunks, allowing inter-filter data passage. These filters are not reliant on the previous filters in the series, and therefore may be placed in any order. Additionally, this also allows additional filters to be added with ease and allows filters to be removed without dependency issues.

Video/audio decoding is a task usually tackled using the Pipe and Filter architectural style. The data is read from the hard disk (in Kodi's case, this is through the "Local File Manager" component) and then must be decoded before it can be rendered on screen. This process involves several independent steps, including raw decoding, inverse quantization, motion compensation, and finally frame rendering. Additional effects can also be applied to the video using filters, including rotation, cropping, or resolution scaling. The pipes then act as a buffer of upcoming video data which, once through the pipeline, will be displayed to the user. Kodi's video renderer component contains this Pipe and Filter architecture internally and uses it to transform the raw data streamed into visual data for the Player Core to output. The Audio renderer contains a similar internal pipe and filter architecture, with subcomponents that extract the audio from the container file, decode it, and apply effects like volume adjustment.

Repository

In the Repository style, one component is responsible for managing/delivering data to a collection of other components. This component then becomes a "repository" for data, as it must be accessed by other components for them to get the data they need for execution.

The Repository style is employed when accessing files for media consumption, either from a local store or one over the internet. When accessing data locally, the Local File Manager component acts as a repository for the data. All other components must send their requests

through the local file manager for both reads and writes. The Local File Manager is therefore the host of all data, and as such a repository.

Client-Server

In the Client-Server style, there are multiple computers in the system which execute different sets of tasks. Many client machines can connect to each server, which act as resource hubs (for computing or data) and serve the resources back to the clients.

The Client-Server architectural style is used in a couple different ways in Kodi's architecture. When addons are installed, the system must contact an external main Kodi server, which stores a list of all available addons. This list must be requested and transferred from the server to the client for the user to select the addon they want. Once this addon is selected, the client must connect to the server once again to download the addon data to be installed. Finally, any installed addons likely also must connect to remote servers. Most addons give the user access to content over the internet, from services like YouTube and Soundcloud, and as such must connect to those company's servers.

Conceptual Architecture

The Kodi software consists of numerous components, providing abstraction, reusability, and portability. Each of these components are subsystems, keeping much of the component-specific functionality abstracted away. Here, we are going to examine the broad subsystems underlying the system.

Components

GUI Manager

When Kodi is initialized, the user can navigate through the features of the software, eventually selecting what they want. The GUI manager is responsible for displaying this interactive interface and registering user input. Within this component, there are subcomponents responsible for user input and displaying content. The user input subcomponent will register user input from a variety of controllers (Xbox, PlayStation, Touch screen, keyboard), and provide one interface for the GUI Manager. The display subcomponent will take content from the GUI manager and display it to the screen. This subcomponent will be responsible for handling different operating system specific display methods and different screen sizes. The GUI manager will interact with any components responsible for the features it displays. In our test cases, this includes the library manager, requests manager, addon installer, any addon, and the player core. One of the main features of Kodi is that it can play video files from your local machine. In this case, the GUI manager would interact with the library manager to get local files and use the player core to play them. There are also times where the GUI manager needs data from the internet. For example, when loading the list of available addons to the user. In these cases, the GUI manager will interact with the requests manager. The last great feature of Kodi that we cover is the addon functionality. In this case, the GUI manager would use the addon installer to install an addon, then interact with the installed addon for additional usage.

Local File Manager

Almost every personal software will need the functionality to interact with the local file system. Kodi is no exception. As mentioned earlier, one of Kodi's features is to play video or sound files from the local machine. To accomplish this, Kodi has a component whose sole purpose is reading and writing to the local file system. Since different operating systems have different file system methods, the local file manager will abstract the specific implementation and provide a single interface to the rest of the software. This component provides services to any component that needs access to the file system. In our case, this includes the library manager and the addon installer. More specific information on how the local file manager is used will be detailed in the client components.

Library Manager

As mentioned above, one of the main use cases of Kodi is to play media files. From a user's perspective, Kodi can play media from local files, from a network file system, or from another Kodi application running on the same network. Regardless of where the files come from, the process of playing the file, using the player core, will be the same. To abstract the origin of the files, Kodi has a library manager. This component creates a virtual library of media, made up of files from different locations. This component acts as a client to the local file manager, which allows it to read local video and audio files. While the library manager would also be a client to the network manager and web app manager, we are only focusing on local media in our use cases. The library manager provides services to any component that needs to read media files. This includes the GUI manager and the player core.

Player Core

The main purpose of Kodi is to play media files. To ensure that these files are played accurately, Kodi has a player core component. This component's purpose is to play any media file given to it. It will keep track of important metrics like elapsed time and volume. It will also ensure that different parts of the media, such as audio and video, play synchronously. Within this component, there are a few subcomponents used to render the media. For video files with audio and subtitles, there are video, audio, and subtitle renderers. Other files may use one or a few of these renderers. Regardless of the media being played though, there will always be a transport renderer, used to show the transport overlay. The player core component will provide services to any component that needs media to be played. In our case, the library manager and specific addons will provide media files to be played.

Renderers

While the exact render pipeline may differ, each of the specific media renderers will act the same relative to the player core. Each renderer will have a method to accept their specific

encoded data. They will each have playback control methods, such as play and pause. Each specific renderer will be responsible for interacting with the operating system and hardware specific technologies. Within each renderer will be a pipe and filter style processing line to convert encoded data into playable media. The only Kodi component the renderers will interact with is the player core, which controls the playback of these components.

Central Kodi Server (Kodi API)

While Kodi could run without an internet connection, there are a few features that require it. To update the Kodi application and install new plugins, Kodi needs access to a central repository. Kodi has a central service, accessible from anyone on the internet, used to share updated application information. This central service will have access to a repository of Kodi software and plugins. This allows older Kodi installations to have up to date resources. This API will also give developers the functionality to publish plugins, making them available to all Kodi users. This component would be the sole component interacting with the Kodi software repository. The Kodi server would also act as a server to any service that wants access to up to date Kodi software and plugins.

Request Manager

Some Kodi functionality depends on data from the centralized repository, mentioned above. To interact with the central API, the local Kodi software has an internet data access component. The request manager operates as a middleman between the Kodi components and the Kodi central server. It provides an interface to other components to get updated software. To connect to the central server, the request manager uses the http protocol. This component acts as a client to the central Kodi server and provides services to any components that need internet access.

Addon Installer

Not all features of Kodi ship with its installation. Plugins/addons are available to expand the functionality of the software and enable new features. To get these new features on local installations of Kodi, the user must install addons. The addon installer works by downloading addons from the central Kodi server and installing them to the local machine. It uses the request manager to get necessary addon files from the central repository and uses the local file manager to store them locally. The GUI manager uses the addon manager by specifying which addon needs to be installed. This component downloads and installs plugins asynchronously, so the user can use Kodi during installation.

Addons

While each addon can have vastly different functionality, they each operate with the system in a similar way. After an addon is installed by the addon installer, it operates like a new

component, embedded in the system. It has access to other components like the requests manager, the file manager, and the player core. In our explanation, we go over a music streaming addon. Once installed, this addon would use the request manager to stream music data from a server. It would input this encoded data into the player core, which would play it for the user. While not every addon would work identically to this one, the music streaming addon shows a good demonstration of the reuse of components.

Data Flow

The primary data flow in Kodi is turning information in a supported file format into a displayed video that the user can view. When requested by the user, the data, in the form of video format, is loaded from a disk or network, and passes through rendering pipelines. The audio and video data are sent into separate renderers, which convert the file into a playable video and listenable audio, they are then synced and verified by the Player Core. The file, once fully rendered, is delivered to the user.

Control Flow

Control flow in Kodi is primarily managed by the GUI, based on the user input in the GUI. The GUI will send out requests to the required components for the task, e.g., send a request to grab a file and send it to the renderer.

Concurrency

To provide a quality user experience, some components need to run concurrently. Some concurrent processes improve the user experience, and some are necessary for the software. The main purpose of Kodi is to play media. Since most media consists of multiple data feeds, such as audio, video, and subtitles, it is necessary for these pieces to be rendered concurrently. The video, audio, subtitle, and transport renderers must not only run concurrently, but also in sync. Since the user can interact with the transport overlay while the media is playing, the transport controller must also run concurrently. Without the concurrent execution of these components, the Kodi software would be essentially useless.

While the other concurrent processes are not necessary for the application to run, they do improve the user experience. A few examples of these are with respect to the addon installer, the renderers, and the Kodi central server.

First, when installing new addons, the user should be able to use the rest of the system as usual. It can take many minutes, if not hours, depending on internet connection speeds, to install addons. If this installation was done non-concurrently, the user would have to wait for the installation to finish before using the rest of the app. Since the addon installer runs concurrently, the plugins can be installed in the background, while the user uses the rest of the software.

Next, decoding encoded media data can take many steps and a lot of time. Within each of the renderers, there is a sequence of functions to decode the data. Instead of waiting for the last function to complete before the first function starts the next batch, it can start immediately. This way, the renderers can maximize the throughput, which may be necessary to play media in real

time. The pipe and filter architecture assumes data will be streamed between filters, as opposed to wholly fed, so that the user can see the output while the next portion is being rendered.

In our last example, we can examine the concurrent nature of the central Kodi server. Each request made to the server can require several steps to complete, to eventually return a response. In most cases, the server would have to query the repository for more data, which could take even longer. If you multiply this by the millions of local Kodi installations, it seems almost impossible for the requests to be processed in series. Instead, requests can be added to a ready queue. While the server waits for some queried data for one request, it can continue executing another one. To improve the performance even more, the system can use multiple instances of the server service, so multiple requests can be executed at the same time.

Conceptual Architecture Box-and-Arrows Diagrams

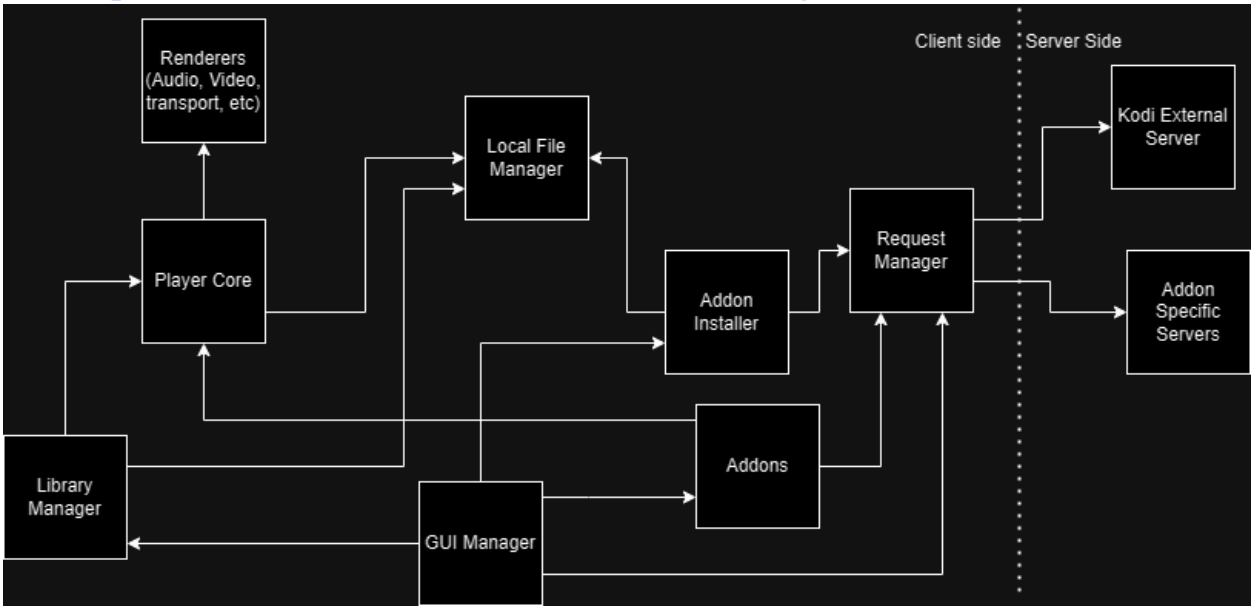

Figure 1: Overall Box-and-Arrows diagram with all outlined components and dependencies.

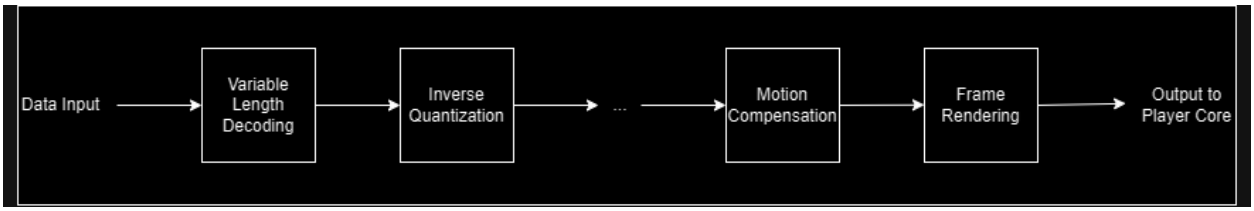

Figure 2: Box-and-Arrows diagram of the Pipe and Filter structure used in the renderers. Video renderer shown above. "…" used to show that further filters can be dropped in for extra effects.

Use Cases

Perhaps the most obvious – and critical – use case for Kodi is simply playing a video file from the user's local machine. The sequence diagram below outlines the component interactions

from the time the user clicks the "Movies" button on the home screen to see a list of available video files, until the video begins playing on the screen.

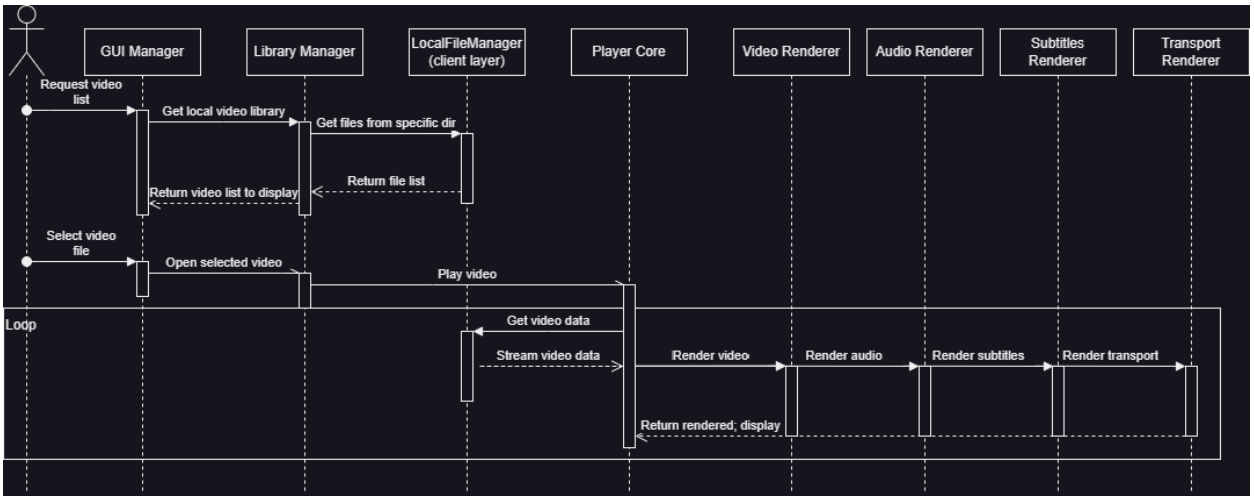

Figure 3: Sequence diagram detailing use case 1 - a user selecting and playing a video

The sequence begins with the user requesting to see a list of all video files, which is sent to the GUI Manager. From there, the request is forwarded to the Library Manager, which must communicate with the Local File Manager to get a list of all files available in the Library Manager's preset video directory. Had the user been looking for videos over the network, a "Network File Manager" component would have been used instead. The Library Manager then filters the response and only returns video file types to be displayed by the GUI Manager.

Once the user has the list, they can select a video file from the list to begin playing it. The GUI Manager receives the user input and sends a message to the Library Manager asking it to open the selected video file (which it already knows the location of from the earlier query to the Local File Manager). The Library Manager then passes this file path to the Player Core, which handles the video playback.

The Player Core then repeatedly asks the Local File Manager for the video data, and then passes the streamed data along to the various renderers. The renderers each return their data to the Player Core, which oversees displaying the results to the user and keeping the audio/video in sync for each frame of video.

For our 2nd use case diagram, we chose to look at Addons. In this diagram, the user pulls up the plugin list, selects to install Soundcloud, and then listens to a song from the plugin once it's been installed.

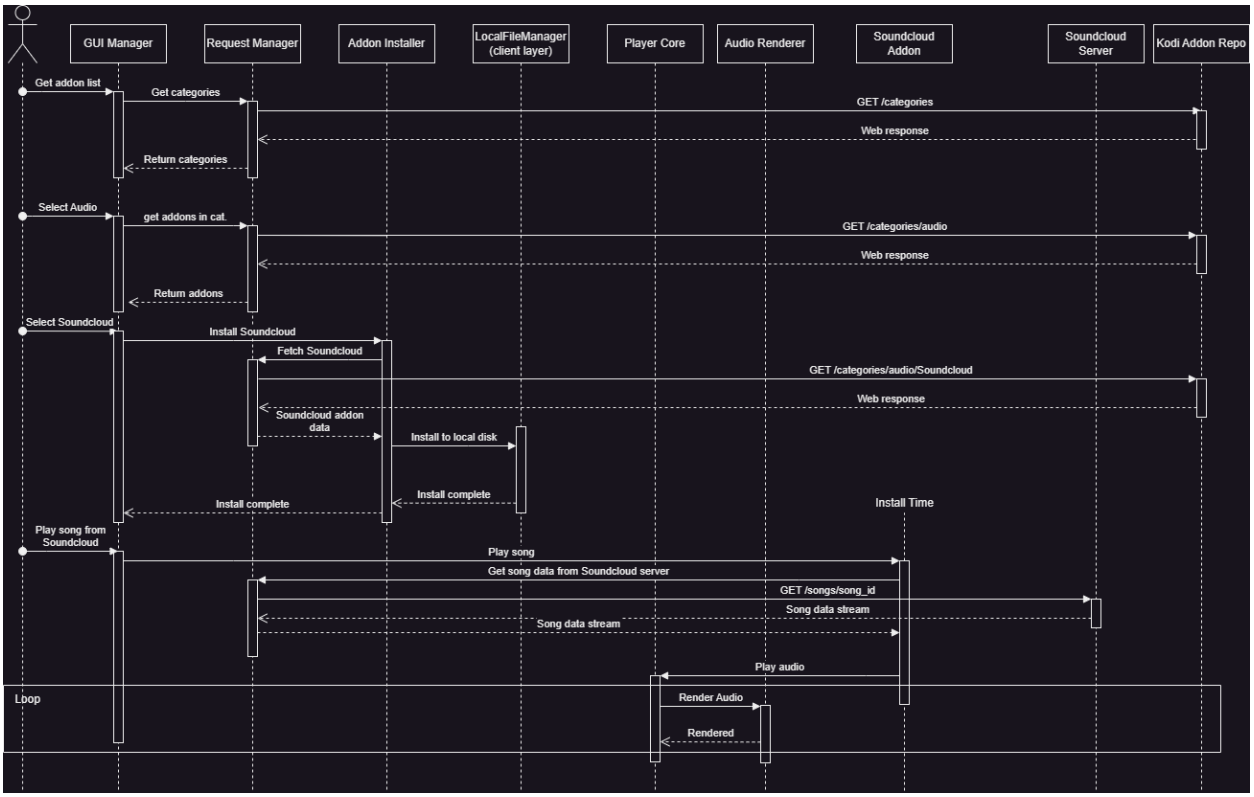

Figure 4: Sequence diagram of use case 2 – a user selects an addon from the available list, installs it, and then uses it to play a song from a remote server.

The user first clicks into the addon browser, to see a list of addons. This initially shows the user categories, which the GUI Manager must get from the remote Kodi Addon Repository. To do this, it asks the Request Manager component to connect to the Addon Repository and obtain the list of categories. The GUI Manager then displays this information to the user. The user then selects the "Audio" category, which prompts a similar chain from GUI Manager to Request Manager to Kodi Addon Repo, and then back to the GUI Manager.

Once the user can see the Audio addons, they select to install Soundcloud from the list. The GUI Manager sends this request through to the Addon Installer, which once again uses the Request Manager to connect to the Kodi Addon Repository. The data for the addon is then streamed back to the Addon Installer, which must use the Local File Manager to save the files as they're unpacked and installed.

When the installation is complete, the user then plays a song from Soundcloud through the GUI. The GUI then passes this request through to the newly installed Soundcloud addon, which uses the Request Manager to get the audio data from the Soundcloud server. This audio data is then streamed to the Player Core so it can be rendered by the Audio renderer and played for the user.

Data Dictionary

Rendering: The process of decoding and displaying a piece of media.

Addon: A small piece of software that can add additional functionality to Kodi.

GUI: Graphical User Interface.

API: Application Programming Interface, used to allow applications to communicate with backend software effectively.

Lessons Learned

Analyzing the Kodi architecture taught us important lessons in work collaboration, designation, and problem approach.

The first challenge faced was trying to gain an intuitive understanding of how the Kodi app works. A lot of time was spent reading over documentation and other reports, but without any experience using the app, it was hard to establish a 'feel' for how the application behaves. In hindsight, it would have been smarter to even start with using the app first, so that the first read-through of the underlying documentation could be better associated with other screens, modules, and even dependencies.

Throughout the compilation of this report, choosing the order in which tasks we done proved to be one of the most impactful decisions. For example, after using the app and reading some of the documentation, it was much more natural to develop the sequence diagram for given use cases, and from there create a more formal dependency graph.

Furthermore, as can be expected in hindsight, this report proved to be more laborious than originally expected, and it would have been wise to start working on it earlier. With more time to organize and communicate our work with each other, work would have been finished sooner before the deadline, and stress levels would have been lower while working on this.

With respect to the more course-related lessons learned, analyzing the Kodi architecture provided a wide range of heuristics and conventions associated with developing applications, such as typical ways of breaking components down into their smaller modules, as for many of us this was our first time properly exploring the design of a system this large.

Conclusion

In summary, our analysis of Kodi's architecture reveals a well-designed system that exhibits a high degree of modularity. The system's architecture effectively separates core functionalities such as media playback, the graphical user interface, and data management, from extensible features provided via add-ons and plugins. This makes Kodi a versatile media center solution capable of evolving with emerging technologies and user demands over time.

One notable architectural component is Kodi's utilization of a client-server model, which enables remote control and streaming functionalities. This broadens possible user interactions and enhances its portability across multiple platforms.

Furthermore, the dependency graphs and sequence diagrams created during our study create a deeper understanding of inter-module relationships. We observed that Kodi's

architecture tends to split up major functional components into many smaller components, each with their own purpose and minimal interdependency between them to allow for maintainability and scalability. Despite having at least 10 major architectural components, no component is directly depended upon by more than 3 other components.

The sequence diagrams reveal the systematic approach in process execution, where tasks are delegated to specialized modules, affirming the effectiveness of Kodi's modular architecture as there is a clear and structured flow of data when a process is initiated.

Looking ahead, there are several avenues for future research and improvement:

- Architectural Formalization: Developing a formal architectural model by further researching constituent components will produce a more rigorous analysis, making it easier to move forward with implementation.
- Performance Benchmarking: Comparative studies can be conducted against other media center software to ascertain Kodi's efficiency and identify areas for improvement.
- Security Measures: With the system's extensible nature through add-ons, enhancing security protocols within its architecture can make it more robust against vulnerabilities.

In conclusion, Kodi's architecture stands as a testament to efficient software design, blending modularity, adaptability, and systematic organization to provide a user-friendly and extensible media center solution. Future efforts should aim to build upon this strong architectural foundation to elevate Kodi's capabilities further.

References

[1] "About Kodi," Kodi.tv. [Online]. Available: https://kodi.tv/about/. [Accessed: 22-Oct-2023].

[2] Kodi.wiki. [Online]. Available: https://kodi.wiki/view/Architecture#Business_Layer. [Accessed: 22-Oct-2023].

[3] "Kodi," Github.io. [Online]. Available: http://delftswa.github.io/chapters/kodi/. [Accessed: 22-Oct-2023].

[4] "Kodi Foundation," Kodi.tv. [Online]. Available: https://kodi.tv/about/foundation/. [Accessed: 22-Oct-2023].

[5] Kodi.wiki. [Online]. Available: https://kodi.wiki/view/History_of_Kodi. [Accessed: 22-Oct-2023].

[6] "Pipe and filter," Berkeley.edu. [Online]. Available: https://patterns.eecs.berkeley.edu/?page_id=19. [Accessed: 22-Oct-2023].