KODI Conceptual Architecture Report

CISC 322

Sunday, October 22rd, 2023

Alexander Muglia - 19am107@queensu.ca - 20208942

Ben Falkner - 21bmf2@queensu.ca - 20306606

Colin Cockburn - 20clc1@queensu.ca - 20238677

Daniel lister - 20dl51@queensu.ca - 20290719

George Salib - 20gs36@queensu.ca - 20281662

Julian Brickman - 19jmb10@queensu.ca - 20206218

TABLE OF CONTENTS

ABSTRACT

This report outlines and examines the conceptual architecture of Kodi, a free open-source media player and entertainment platform. The study investigates how Kodi is structured, examining its various layers, including the client, presentation, business, and data layers. It also discusses the interactions between these layers, shedding light on how users interact with Kodi and how it manages content, both locally and from third-party sources. We explore the control and data flow within Kodi, highlighting how user input, content retrieval, and playback are orchestrated. Additionally, we delve into the system's concurrent operation, which ensures efficient multitasking and performance optimization. Finally, this report delves into the evolution of the Kodi platform, its community-driven development model, and the division of responsibilities among developers, emphasizing the significance of plugins and community contributions. Three use cases further illustrate the system's functionality in terms of content retrieval and user interactions.

INTRODUCTION & OVERVIEW

Over the past few years, Kodi has emerged as a prominent player in the realm of digital media management and entertainment. As an open-source project, initially known as XBMC (Xbox Media Center), Kodi stands as a testament to the power of community-driven development. The open-source nature of Kodi not only means it is freely accessible but also that it thrives on collaborative contributions from a global community. Starting from a group of developers who sought to transform the Xbox into a media center, Kodi has evolved into a comprehensive, cross-platform media player capable of running on numerous devices and operating systems.

Before delving into Kodi’s architecture, it is essential to understand its core functionality. Kodi serves as a media player and an entertainment hub, allowing users to manage and access their diverse media libraries. It provides a unified platform for organizing, streaming, and enjoying various types of content, including movies, music, TV shows, photos, and more. Kodi’s extensibility and versatility make it a popular choice for those looking to create a personalized multimedia experience.

This report aims to provide an in-depth exploration of Kodi’s conceptual architecture, shedding light on the underlying components that form the backbone of this media center. Key elements, such as the media library, plug-ins framework, user interface, and playback engine, play pivotal roles in Kodi’s functionality. Additionally, we will analyze how these components interact with each other, examine the date flow, and understand the concurrency between elements, which collectively ensure the smooth operation of the system.

To aid in this exploration, we will utilize various diagrams and models to illustrate the system's flow, including sequence diagrams and various use cases. These visual representations will help display the decision-making processes and pathways the system follows. Furthermore, we will explore Kodi's external interfaces, offering a comprehensive analysis of the system's overall architecture and its relevance in today's digital media landscape.

SYSTEM FUNCTIONALITY AND INTERACTING PARTS

Kodi is made up of different layers that serve as an abstraction for the various components of the system. Kodi includes the following layers:

Client Layer

The client layer is how the user accesses the application. Since Kodi is a multi-platform app, the same functionality has to be implemented across mobile (IOS and Android), desktop (Windows, MacOS, Linux), as well as potentially lower end devices (Raspberry Pi).

Presentation Layer

The presentation layer handles dynamic generation of the user interface and other data that gets sent to the client. This consists of components to manage different windows, GUI elements, and audio. In addition, the presentation layer is where user input is managed. User input can be done through standard operations such as with a mouse and keyboard, or through Kodi's official remote software.

Business Layer

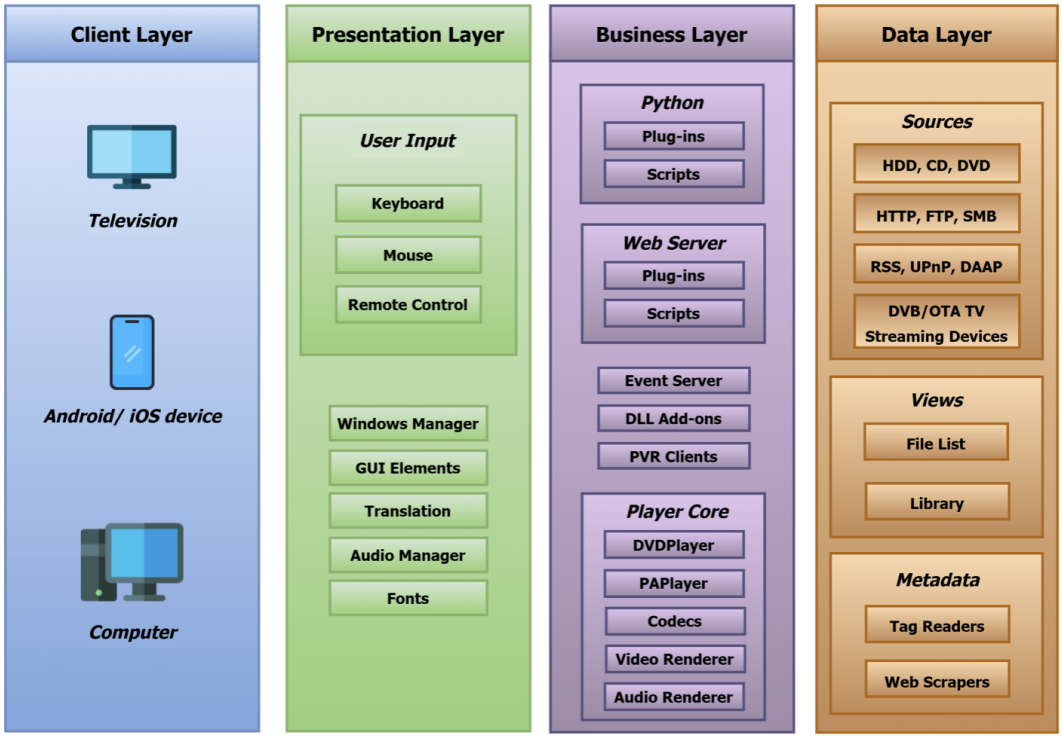
The business layer is an intermediary between the presentation layer and the data layer. It encapsulates the business logic of the app. Before data is used by the presentation layer to generate UI, or before data is stored in any of the databases present in the data layer, it has to pass through the business layer. This ensures that data is validated across any constraints, rules, or security measures that the app needs to impose. For Kodi, the business layer manages codecs, and handles rendering of any audio and video it receives. It also holds any python scripts used for the data, as well as the web server to communicate.

Data Layer

As previously mentioned, the data layer is where the data is managed before extra processing might happen in the business layer. It handles generating queries to receive

and store data. For Kodi, any errors or exceptions that might occur when accessing video or audio from their sources are handled within the data layer. Any web scraping that Kodi does is also part of the data layer.

The following diagram from the Kodi Wiki is used to visualize these layers:



Kodi is structured with the following components

Viewed functionally, the different elements that make up Kodi interact with each other to pass data back and forth between the user and the application. For instance, if a user navigates the UI to watch a movie, any language and accessibility features are sending outputs and receiving inputs from the user, while simultaneously communicating with other components to update the GUI elements. Then, when the user has settled on the content that they want to view, Kodi's player component has to request the required data from the video library. Similarly, if the user wants to play music, the music player component must request data from the music library. These libraries would be where the content is stored. Once the data is properly rendered and processed then it can be sent to the respective player that the user wants to use to stream their content.

CONTROL & DATA FLOW

Upon initialization, Kodi starts on the client layer and initializes settings based on the device's capabilities and user preferences. The client layer awaits user input, and the user input module captures commands from the peripherals. This input maps to specific functionalities, often invoking GUI elements to display options or prompt further actions. Users can find content organized by the views module in the data layer, which then sends the data back to the presentation layer. If the user accesses media, the business layer checks whether the media is local or streamed, fetching it from the data layer. In the case of local media access, the source elements might be involved in fetching content from HDD, CD, DVD, etc. If the access is for streaming, then protocols like HTTP or FTP within the business layer establish a data stream.

In the business layer, the data flows into the player control module, responsible for decoding incoming media data and translating compressed data. The player control module is also in charge of controlling playback flows such as pausing and playing media (using the user input module) and outputting the video and audio to the presentation layer. Audio data retrieval is directed to the audio manager module, which tailors its output based on available output choices such as HDMI or speakers. Metadata about the media in the data layer is managed by the metadata elements, which either use tag readers to extract data directly from the media or web scrapers to fetch additional details from online databases.

If a user engages in custom scripts, the python module retrieves the scripts from the data layer and executes them. These scripts could fetch external data or modify internal data structures. Remote control and management are facilitated by the web server module, which establishes an interface using data structures from the data layer, allowing remote users to send commands. When a user exits the program, various modules relay their current state and settings to the data layer. This ensures that user preferences, playback positions, and other session-specific data are saved for future sessions.

CONCURRENCY

Kodi uses multi-threading to allow multiple operations to occur simultaneously. One example of this is running media playback while navigating a media library. Operations such as updating libraries, fetching metadata, or checking for add-on updates can run in the background without interrupting the user's primary tasks. To ensure smooth playback of media, Kodi must load portions of the media while playing back another portion. Kodi's support for third-party add-ons and its modular nature allow for multiple plugins and extensions to run concurrently,

potentially fetching data or providing functionalities. Kodi must also access its databases to perform tasks such as fetching metadata or saving user preferences without interrupting other functions. Additionally, Kodi manages its network operations concurrently, enabling it to perform multiple operations, such as streaming content from one source while downloading updates or metadata from another. In summary, concurrency in Kodi ensures the system is optimized for performance and responsiveness. This concurrency enables Kodi to process and display multimedia content efficiently while simultaneously handling user interactions and other background tasks.

SYSTEM EVOLUTION

Kodi, being an open source platform, uses version control software Github to allow members of the community to file, track, and fix bugs as well as implement new features. It is here that developers are able to address security issues, ensuring the safety and reliability of the software for all users. For example, SFTP (Secure File Transfer Protocol) was introduced in Kodi major version 10 with the introduction of libssh to the codebase. Furthermore, performance optimizations that occur in the Kodi core are crucial as the user base grows, allowing the platform to continue to deliver a smooth user experience. Engine and core library updates typically fall into this category of performance-boosting changes.

One of Kodi's greatest strengths is that it runs on a wide range of platforms. Originally developed to work on the Xbox, the Kodi community has continued to add platform support throughout the major releases, allowing the media player to stay competitive in the market of general media aggregators such as Apple TV. Kodi now supports Android, BSD, Linux, macOS, iOS, tvOS and Windows. One challenge with committing to the support of multiple platforms is that not only is the Kodi software constantly evolving, but the environment which Kodi has to run in is as well. One example of this is the increasingly widespread use of the ARM architecture, which requires developers to add support for platforms that previously already had support. Namely, in Kodi major version 10 ARM support was added for linux devices which were already supported on Intel x86 architecture. This is likely to continue to be an issue that the Kodi developer community faces as more technology such as Apple's M series chips are released which now use the ARM64 architecture.

The core of Kodi is mainly written in C++, however developers that wish to contribute to the community are not limited to core contributions. Due to Kodi's plugin-style architecture, developers well-versed in a wide range of languages are able to contribute by the creation or maintenance of Kodi plugins. The most popular choices for plugin developers are Python and Java. This system of user-driven plugins allows for quick deployment of new ideas and utility

without having to go through the slower, more rigorous scrutiny of code that is added to the Kodi core.

Since Kodi version 9, major releases to the platform have taken on an Android-style naming system, alphabetizing each new release with names such as Babylon, Camelot, all the way to the current versions of Nexus (production) and Omega (development).

DIVISION OF RESPONSIBILITY AMONG DEVELOPERS

As stated previously, Kodi is a community-driven open source product. Furthermore, Kodi has a plugin system that allows independent developers to add their own features as they wish. Due to these choices, the development responsibilities are not limited in the traditional style of teams of developers, testers, and quality assurance. In this system, anyone who wishes to contribute to the platform is welcome to select a known issue and do what they can to fix the problem. However, it is not the case that anyone can make any change they wish to the codebase of Kodi. Once the developer is satisfied with the changes they made, they will submit their code to be merged into the rest of the product. There is then a verification process conducted by trusted members of the community to vet the code for correctness, security issues, and overall quality. This is typical of the open source style in software development, allowing both community contribution and the ability to uphold standards of a small set of chosen developers.

If an individual wishes to circumvent the arduous process of core contribution, there is also the path of plugin development. Plugin developers, also members of the community, are able to create valuable additions to the software with much less scrutiny of their code. This is a fantastic way for passionate users to create a utility that they believe can help a subset of users (maybe just themselves), and allow the users to decide for themselves if they would like to install the plugin or not.

Overall, development of the Kodi platform is fully community-driven. Whether it be core contributions relating to performance, security, or compatibility, or a one-off plugin for a very specific use case, the system evolves through community contribution.

USE CASES

Use Case One: Playing network content from Kodi Application

Actors: Users

This illustrates how a user would access content from a network source. The user first selects which content he is requesting to access. The controller acts as the user interface layer, which processes the user's input and sends it to the content manager. For security purposes it's important to have multiple layers between the user's input and the database, performing validations at each layer. In the controller validations are typically less extensive (Verify the user's input is allowed Ex. inputting three tab characters would be invalid input and would not proceed to the next layer, or length of input). In the manager more extensive validations are performed (not sending a SQL injection to the database). Once the input is validated in the manager the request for that content is sent to a third party content server. Here we've abstracted and generalized the server layer to just content server rather than a specific third party application as there are many different add ons and options in Kodi (Youtube, SportsDevil, USAToday all have different servers for accessing their content). Once the request is sent there are two possibilities, the first being that the content exists and was successfully fetched from a third party server. If the content exists an encoded and compressed version is then sent back to the content manager. Here the content is sent to a decompressor, then a decoder, then the renderable content is sent back to the content manager. In the content manager the content is formatted then rendered and sent back to the controller to be given to the user to enjoy. On the contrary the second option exists for when the requested content cannot be successfully fetched from a third party server. In this scenario the decryption can be skipped as the content does not exist within the system. The failed request message is sent to the content manager, which then parlay's that information to the controller to be displayed to the user.

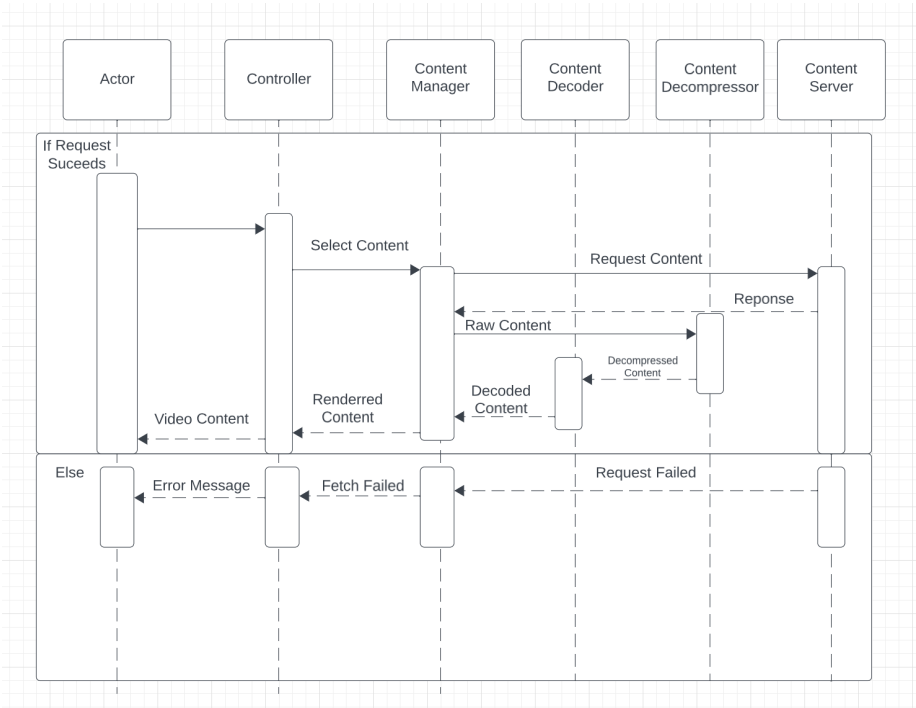


Figure 1: Conceptual architecture of requesting content from a third party.

Use Case Two: Playing local content from Kodi Application

Actors: Users

(Note that this is a distinct use case, on Kodi you select whether you're accessing third party, or local content prior to sending in a request for that content, so the content searching for each respectively have different conceptual architectures, opposed to one more generalized approach with two distinct pathways)

This scenario is very similar to requesting content from a third party server. Initially the user first selects which content he is requesting to access. Validations are performed in the controller and the content manager. Once the input is validated the controller the content manager formats a path to the content, then sends that to a content searcher. The content searcher navigates your local directory to find the content on your local device. The content generally would not be compressed or encrypted so the content can be sent back to the content manager directly, where it formats and renders the content appropriately. Once these tasks are performed the content is then sent to the controller to be accessible to the user. If the request fails, a message is sent to the content manager, which then parlay's that information to the controller to be displayed to the user.

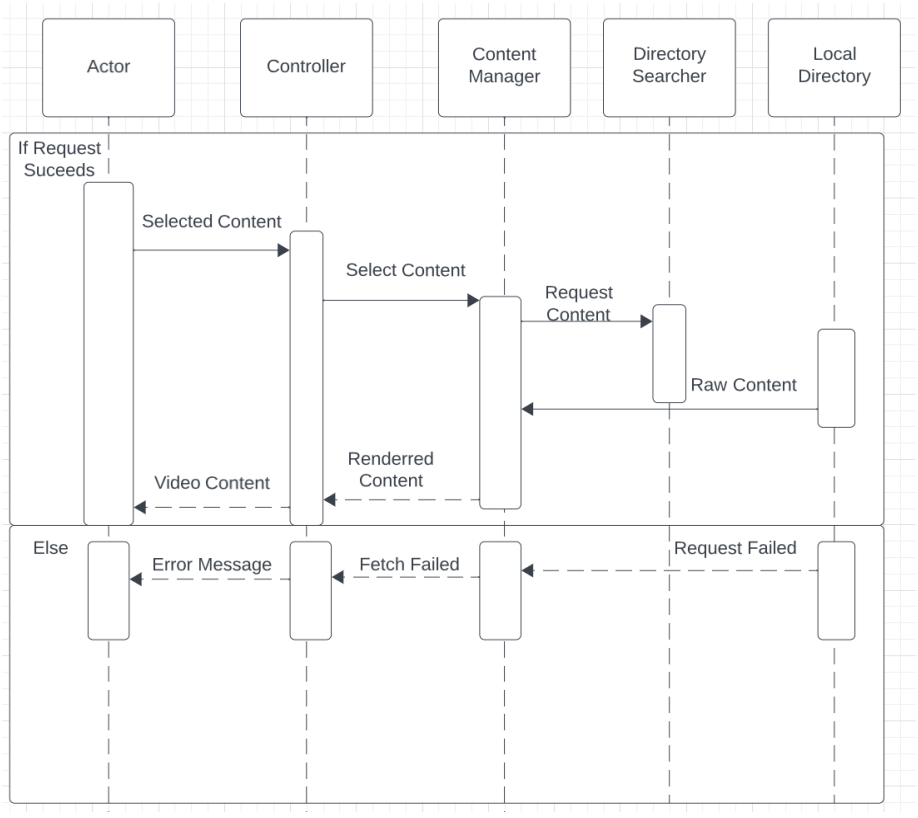


Figure 2: Conceptual architecture of requesting content from a local source.

Use Case Three/Four: Searching local content on Kodi, and selecting content to view

Actors: Users

Part one:

This case addresses the scenario where the user searches content he has added to his Kodi application, and then selects the searched content (to view not to load as the cases above address loading). First the user inputs a requested search, then basic validations are performed (as described above) in the controller. Then the inputted search is sent to the search engine. Here initial more extensive validations are performed, and then the input is passed to a local database of the content associated with this user. The database returns an array of the content titles to the search engine, Inside the search engine the content is looped through and sorted to be displayed in the correct order (I believe it's alphabetical, but I could be mistaken). Once the sorting is performed the sorted content is then passed back to the controller and displayed to the user.

Part two:

Once the list of content is displayed to the user, the user can then click on the content to view it (to view not to load as the cases above address loading). It's important to note in this case that content information is representative of general information of the selected content (title,thumbnails etc) not the full content, as there would be large computational overhead if everything in a folder would be rendered. Prior to displaying the content information Kodi verifies that the user has the proper permissions to access this content. This is performed by passing the selected content request to the permissions manager, which then navigates the local directories to verify that the user has the proper permissions to view this directory/file. If the permissions are in order the permissions manager receives a message, which prompts it to fetch the content information from the content database. This is sent back to the controller and displayed to the user. If the permissions are invalid, or If the request failed, a message is sent to the permissions manager, which then parlay's that information to the controller to be displayed to the user.

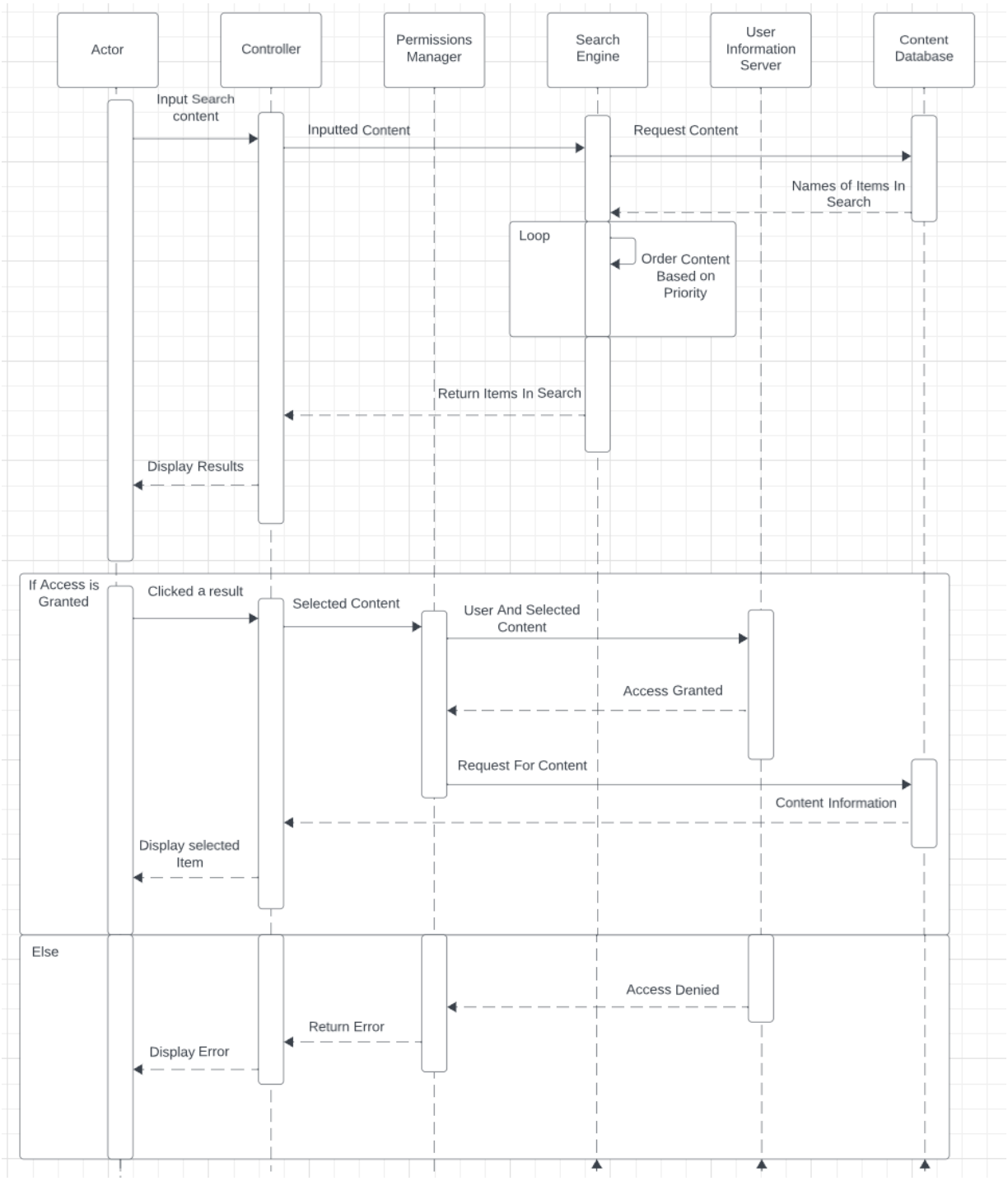


Figure 3: Conceptual architecture of searching local content on Kodi, and selecting content to view.

DATA DICTIONARY

Media Player: Software for playing multimedia computer files like audio and video files.
Library: Collection of media content, either video or audio.
GitHub: Platform used to store the source code for a project and track all progress.
Metadata: Data providing information about one or more aspects of the data.
Multi-Threading: The ability of a central processing unit to provide multiple threads of execution concurrently.

NAMING CONVENTIONS

XBMC: Xbox Media Center
GUI: Graphical User Interface
SFTP: Secure File Transfer Protocol
ARM: Advanced RISC Machine
BSD: Berkeley Software Distribution
HDD: Hard Disk Drive
HTTP: Hypertext Transfer Protocol
FTP: File Transfer Protocol

CONCLUSIONS

Kodi's architectural analysis reveals the intricate design that makes it such a versatile media center. The four distinct layers: client, presentation, business, and data, collectively form a robust framework that enables users to access and manage their multimedia content effortlessly. These interactions between layers ensure that content is securely and seamlessly processed, whether it's sourced locally or from third-party servers. The control and data flow within Kodi highlights the orchestration of user input, content retrieval, and playback. Kodi's concurrent operation allows for efficient multitasking, making it a robust and responsive media center. Notably, Kodi's commitment to openness and extensibility allows for continuous open-development, and its support for a wide range of platforms, including the evolving ARM architecture, showcases its ability to adapt in the ever-changing technological landscape. Additionally, Kodi’s platform allows for a vibrant and community-driven development model, with an open-source approach that allows for contributions from developers and enthusiasts all around the globe.

In summary, Kodi's conceptual architecture serves as a foundation for its continued evolution as a leading media center. By dissecting its layers, understanding user interactions, and

appreciating the role of community contributions, we gain insight into how this open-source project has become a cornerstone of the digital media management and entertainment industry. Kodi's success is not only defined by its codebase, but by the dedicated and passionate community that fuels its development, ensuring its position in the world of media consumption.

LESSONS LEARNED

This assignment aims to research and outline the conceptual architecture of Kodi. In trying to accomplish this, we encountered many difficulties and setbacks, such as the ones mentioned below:

1. **Early Exploration of Architectural Concepts:** One of the key takeaways from researching Kodi's conceptual architecture is the importance of exploring the broad architectural view early in the project. This allows for a more streamlined process and prevents the need to edit or rewrite components and concurrencies to fit the desired architectural style after they have been completed. Future architectural investigations should aim to understand the system's high-level structure from the beginning.
2. **Diverse Sources and Documentation:** Kodi, is an open-source project with contributions from a global community. That being said, gathering information on Kodi's architecture is challenging due to the decentralized nature of the project. Architectural concepts and documentation are distributed across various sources on the internet. It is vital to organize these sources efficiently in order to create a comprehensive architectural report.
3. **User-Driven Plugin Architecture:** Allowing developers from different backgrounds to contribute valuable additions to the software quickly is crucial to Kodi’s success, however, it makes it difficult to understand/quantify the scope of the whole operation. Thus, a better understanding of the extensibility and openness of the platform, particularly regarding the plugin architecture, is essential for an accurate portrayal of the system's capabilities.

REFERENCES

"Add-on Development." Kodi Wiki, https://kodi.wiki/view/Add-on_development, Accessed 21 Oct. 2023.

Kodi Community Guidelines, https://kodi.wiki/view/Official:Kodi_Community_Guides. Accessed 20 Oct. 2023.

“Kodi Forum.” *Kodi Community Forum*, https://forum.kodi.tv/. Accessed 22 Oct. 2023.

Kodi Wiki. Official Kodi Wiki. (n.d.). https://kodi.wiki/view/Main_Page. Accessed 20 Oct. 2023.

Xbmc. “XBMC.” *GitHub*, https://github.com/xbmc/xbmc. Accessed 20 Oct. 2023.