KODI Conceptual Architecture Report

CISC 322/326 - Group 4

Kabeer Adil - 20ma101@queensu.ca
Kate Edgar - 20kie@queensu.ca
Nicholas Tillo - 20njt4@queensu.ca
Raksha Rehal - 20rsr3@queensu.ca
Yu Xuan Liu - 20yxl3@queensu.ca
Xiyun Cao - 18xc17@queensu.ca

TABLE OF CONTENTS

ABSTRACT

This report is built to highlight the multimedia, open-source entertainment platform known as KODI Media Player. Using both a modular and layered style architecture, KODI is unique in the sense that it is an open source project, meaning there are hundreds of current and potential contributors that work to constantly update the software and track bugs [3].

The KODI repository is available on GitHub and is largely coded in C++, so anyone with experience and knowledge can make effective changes to the software. While this isn't exactly common practice, examining KODI as a functioning entity makes it clear that the architecture in place is robust and can be used effectively as time goes on.

INTRODUCTION & OVERVIEW

Initially developed by the XBMC Foundation, KODI is a self-described "ultimate" entertainment center, with a hand in films, games, television shows and music. The source code was released to the public in June of 2012 and since then, has been constantly expanding and improving [2]. On the surface, KODI maintains a user-friendly and sleek design, allowing users to view pictures, films and music with relative ease. Its multi-faceted approach to entertainment can easily make it a staple in any household's living room. In order to supplement this multi-pronged strategy, the KMP utilizes C++ and Python to create plugins that support numerous add-ons and external, third-party softwares such as Spotify, Hulu, or YouTube [2].

Users can utilize these add-ons to customize and further supplement their streaming and download libraries. KODI originated as software built for the XBOX gaming system but the rising popularity of the software and its user-friendliness led the XBMC Foundation to decide on expanding it to other platforms and avenues. It is currently available on iOS, OSX, Android and Linux as a result [2]. In the backend of the program, hundreds of software engineers, testers and architects are constantly working with each other in order to continuously improve KODI's source code by fixing bugs that are sent in by its various users. By examining the prominent stakeholders, it becomes clear that there are only a few developers that have the right to ratify or deny code contributions and/or pull requests [2]. They earned this privilege due to a large amount of commits to the Git, signalling great dedication to KODI's continued improvement.

SOFTWARE ARCHITECTURE OF KODI

Functionality and Requirements

From studying the documentation and the current concrete architecture of KODI that is available for public use, we can infer what the original requirements of the system might have been to better understand its conceptual architecture. The KODI system's goal and primary functionality is to act as a media player that works on a wide variety of different media formats as opposed to conventional media players that are restricted to working with one specific media format. This means that one of the key requirements of KODI is to ensure that the platform

highly supports portability while maintaining its performance. These requirements can be categorized into two groups:

Media Players

- Includes all functions that go with the user's interaction regarding the browsing and access of content.
- E.g. Graphic interfaces, image displayer, music player, video player, streaming, customizable skins.

Libraries

- This includes all the functions related to gathering, storing, and maintaining the entertainment content.
- E.g. Storage, rendering, sorting, add-ons, web scrapers.

Quality Attributes

Scalability

The multimedia nature of KODI requires that it must be able to be scaled with an increasing amount of file types and file sizes. It must be able to support anything from small audio files to hours-long video files.

Integration

The core functionality is shaped by the ability to create custom add-ons for any file sharing needs. Hence, this design choice also affects how the architecture is laid out. The system must accommodate for any additional space required by third-party components, further encouraging a modular design in order for subsystems to be grouped, accessed and adjusted more easily. Hence, we can see that KODI enforces three main invariants within its system; (1) code documentation, (2) installation, and (3) modular design. The presence of these invariants allow for seamless integration of external software into the KODI system.

Portability

The KODI system is designed to be able to operate on a variety of different devices such as standard Windows or Linux PC, most of the hardware in the Apple environment, and many others. Additionally, as previously established, KODI must be able to support a variety of multimedia formats including containers, videos, audios, images, and subtitles.

Testability

As KODI is built by independent developers without much structure or large documentation, the architecture must assist in aiding the developers with locating and localizing issues in the code.

Conceptual Architecture

Main Architectural Styles Utilized

Upon studying the way KODI functions, and while extrapolating from the concrete architecture, we quickly are able to notice that the system is a mixture of a **modular** and a **layered** style. KODI has multiple separate modules that all deal with different categories of operations, and communicate outside of their own module only on the topmost level [3]. This ensures that the components are able to be replaced easily, added onto at a high-level, and can be utilized on their own. This design choice satisfies the required integration and testability requirements established above. Each one of these modules are implemented in a layered design, in order to clearly distinguish interactions and reduce dependencies that may impact the desired modular design. This layered style gives the structure to help with the required scalability and portability previously mentioned.

Top-Level System

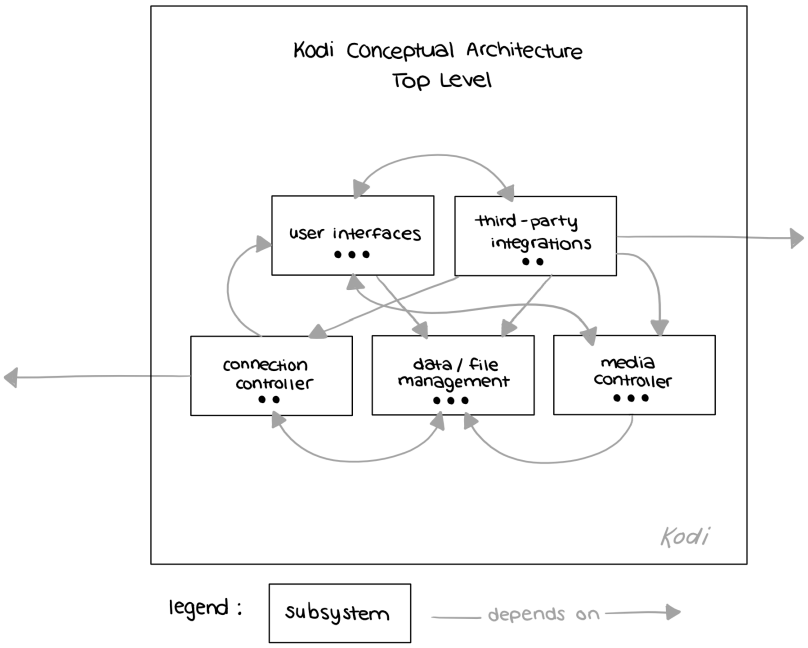

Figure 1: Conceptual architecture of KODI at the highest level of abstraction

When focusing on the conceptual architecture of KODI's system itself, we can see that KODI's functions can be grouped according to the following five major components:

1. **User Interfaces**: Refers to the interfaces that make up the parts of the system that the user sees and interacts with (related to KODI's appearance, and any input).
    - *Dependencies*: Third-Party Integrator, Media Controller
2. **Third Party Integrator**: Deals with third-party events that impact/work with the features of the application.

    - *Dependencies*: User Interfaces, External Means (outside of KODI), Connection Manager, Media Manager, Data/File Management
3. **Media Controller**: Is responsible for displaying the differing media types that are available in KODI.
    - *Dependencies*: User Interfaces, Data/File Management.

4. **Data/File Management**: Facilitates the collection and storage of the media files and other related data. This includes aspects like the files themselves, any metadata, and external libraries.
    - *Dependencies*: Connection Controller
5. **Connection Controller**: Supports streaming and connection services that are able to be provided through KODI via external sources/web connections.
    - *Dependencies*: User Interfaces, Data/File Management, External Means (outside of KODI)

Major Components

These 5 groups are individually broken down into separate layers that deal with their implementation. Each component showcases the lower layer, discussing the parts of each module within a few words. Each group has a differing amount of layers that is relative to the complexity of the component. The higher layers use methods and functions from the lower layers without relying on its implementation. Computations, data, and results are then returned to the layer above. These groups encapsulate the functional specifications of all requirements of the main KODI system.

1. User Interfaces

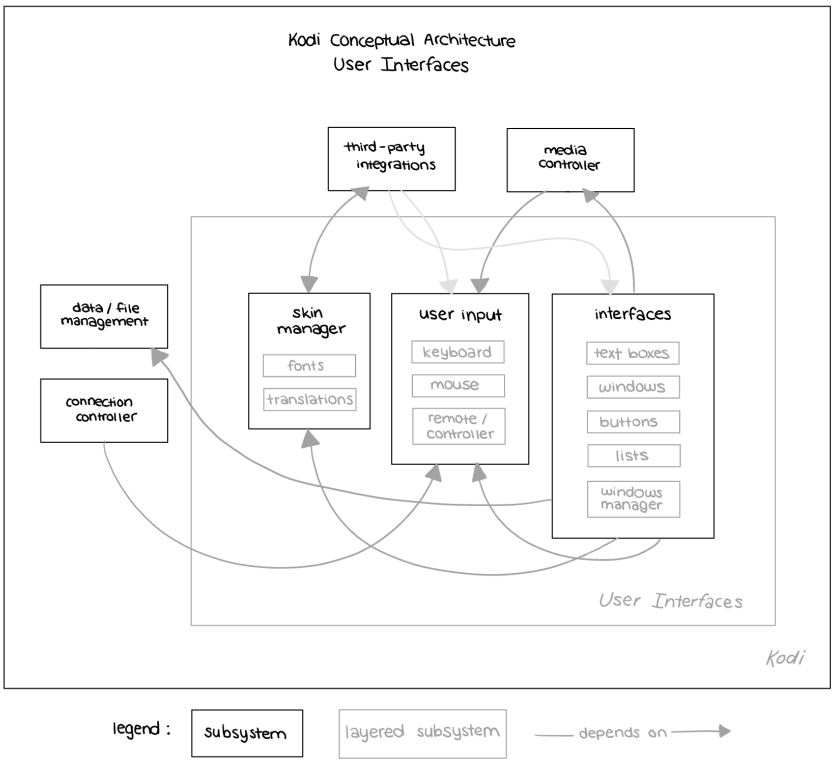

Figure 2: Conceptual architecture of the user interface component

- **Skin Manager** - Manages the visual appearance of KODI via various customizations associated with different skins. This includes fonts, language translations, and other aspects to the GUI.
    - *Dependencies*:
        - (1) Skin manager may depend on add-ons to provide additional skins or customization features that will impact the way these interfaces appear to the user
        - (2) Skin manager depends on interfaces to have windows and objects that it can change the appearance of.
- **Interfaces** - The functionality of the Graphic User Interface including displaying text boxes, windows, buttons, and lists. This also includes Windows Managers that manage different interfaces for different OSs.
    - *Dependencies*:
        - (1) Interfaces depend on the media controller providing the rendered photos, audio, or videos such that they can be displayed by the appropriate interface.
- **User Input** - Gathers all the user data that is given by the peripherals and calls the corresponding module. This includes Remote/Controller Manager, Keyboard Manager, Mouse Manager.

2. Third-Party Integrator

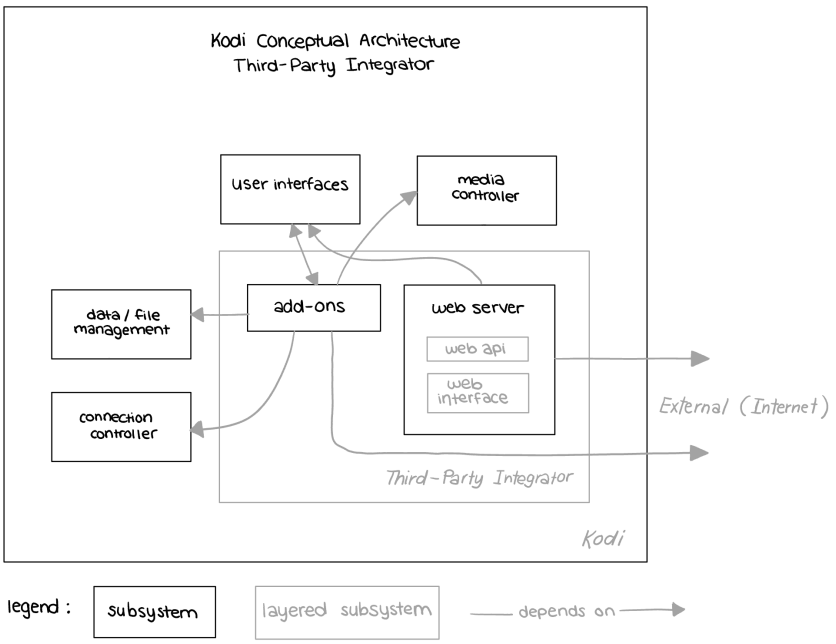

Figure 3: Conceptual architecture of the third-party integrator component

- **Web Server Access** - Allows the user to manage their personal instance of KODI remotely through their web browser or other applications. This would be facilitated through a web interface, as well as utilizing web API.
    - *Dependencies*:
        - (1) Web server access relies on the user interface to be able to represent a client's instance of KODI accurately.
- **Add-On Manager** - Allows the user to download and manage add-ons, plugins, and other widgets to customize their experience.
    - *Dependencies*:
        - (1) Add-on manager relies on user input, as to download and manipulate the addons, it requires the user to input a link.
        - (1) Add-ons can depend on any component as the add-ons it manages would rely on methods and functions given to by any of the other components. (This is demonstrated via a light gray arrow in each of the diagrams).

3. Media Controller

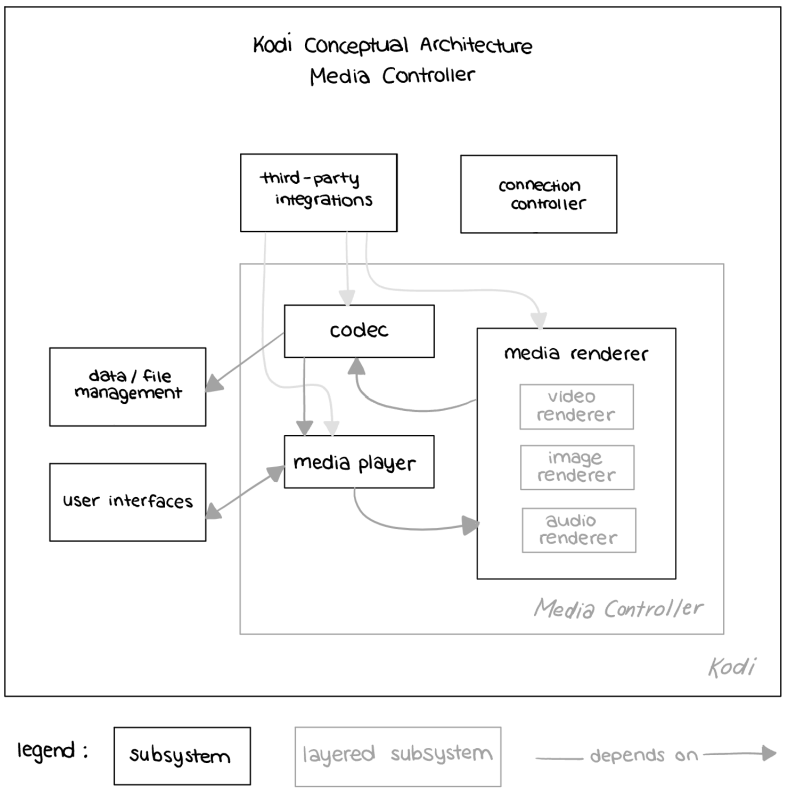

Figure 4: Conceptual architecture of the media controller component

- **Media Renderer** - Manages the rendering of every form of media stored within KODI, including preparing the videos to be played, images to be displayed, and audio to be output.
    - *Dependencies*:
        - (1) Depends on the decoded files and data from the codec in order to render.
- **Media Player** - Displays multi-media to the user and allows their selection via an interface.
    - *Dependencies*:
        - (1) Depends on the media renderer to give it the rendered files.
        - (2) The media player depends on the user interface to give it commands on which media piece to fetch.
- **Codec** - Decodes the data signal that comes from the data/management file into a usable form for the rest of the components.
    - *Dependencies*:
        - (1) Depends on the data/file management component in order to gather the raw coded files.

- (2) Codec depends on the media player to tell it which file it must decode and send to the media renderer.

4. Data/File Management

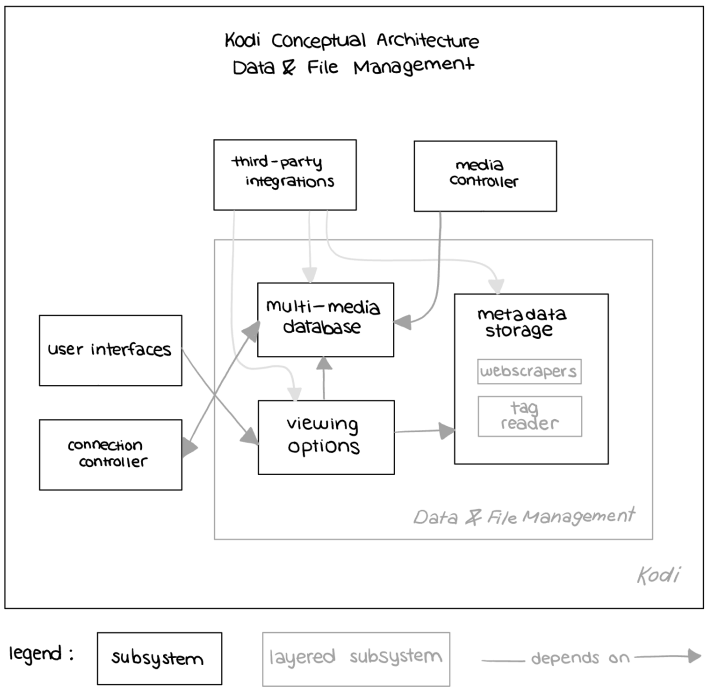

Figure 5: Conceptual architecture of the data/file management component

- **Multimedia Database** - Stores all the media files, including personal ones.
  - *Dependencies*:
    - (1) This database depends on the connection controller to gather files and data from outside resources.
- **Metadata Storage** - Contains information about pieces of media that is stored within the database. This includes webscrapers that gather metadata from online sources. Also contains a tag reader that is responsible for extracting metadata from the files directly.
- **Viewing Options** - Manages how the multimedia content can be displayed. These options give organization specifications to make the multimedia library more readable.
  - *Dependencies*:
    - (1) These options depend on the multimedia database in order to provide the data to organize and manage.
    - (2) Relies on the metadata storage for some sorting and displaying options, e.g. sorting by media type would rely on the metadata.

5. Connection Controller

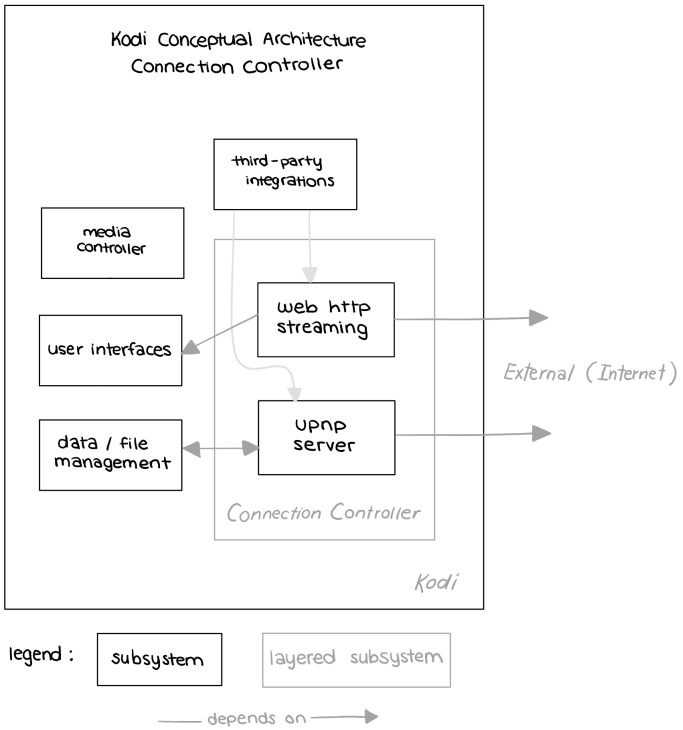

Figure 6: Conceptual architecture of the connection controller component

- **Web HTTP Streaming** - Manages the ability for KODI to use online HTTP servers to stream video or audio online.
    - *Dependencies*:
        - (1) HTTP streaming gathers video data from the Internet (externally) and displays via the KODI interfaces.
- **UPnP server** - Manages the ability for KODI to be either a client or a node for UPnP servers to send or receive data with other KODI instances on the same network.
    - *Dependencies*:
        - (1) The UPnP Server relies on the data/file management in order to share data with other users on the network.
        - (2) the UPnP Server relies on external sources on the UPnP network that allow for downloading of files and data.

Communication and Data Flow

These 5 groups are largely separate, especially on the lower layers of the implementation, where the main data flow is up and down within the same component. However the exception is on the uppermost layer, the top level, the groups are able to interact and call each other and even external sources. For example, the Video Renderer is not within the top level and thus can only interact with the layers below and above it, but the Multimedia Renderer is and thus can interact with other components.

Evolution

KODI's evolution as a system primarily comes from the fact that it is a community-driven development project. To be more specific, due to its status as a huge open-source project, developers from different places actively work on improving the software. As a **layered** style architecture, it lends itself to being easily modifiable, as changes to the function of one layer affects at most two other layers. Furthermore, different implementations (with identical interfaces) of the same layer can be used interchangeably. This is something that aligns perfectly with many parts of their development guideline, where they want:

- **Modular design:** KODI should be able to compile and run even if a non-essential module/library was disabled or removed due to its modules being made up of localized/isolated code libraries without dependencies.
- **Self-Containment:** KODI should be as little dependent as possible on operating-system and third-party services.

All of this amounts to KODI's architectural design as a layered architectural style and it leads to great adaptability and portability as different developers work on improving specific aspects of KODI as a system. With some examples of its evolution being its initial release onto Linux as XBMC Media Center software before eventually porting itself onto both Windows and Mac OS X in 2008, IOS in 2011 and finally Android in 2012 (before finally being renamed to KODI in 2014).

Concurrency

A primary feature that the guideline with which the development of KODI operates under is to have fast load and boot times for end-user perception. As such, there are many different aspects of KODI's software architecture that allows for multiple tasks/processes to still run/start in the background at the same time without the user's knowledge. An example of which would be the JSON-RPC API which allows the HTTP transport to handle response and downloading of files while the raw TCP transport allows response and notifications.

Architectural Styles Used

KODI uses many architectural styles throughout its large implementation. Each major component employs a layered style in which the layers of implementation are separated by layers of abstraction for ease of scalability and portability. Many components use an interpreter style, including the codec that translates encoded data into usable information, as well the interface in order to be manageable on multiple OSs. Pipe and filter is also employed with the renderer and file streaming to ensure continuous data transfer. A repository is used within the database component to allow for minimal duplication of data, and allowing all required modules to interact with the database. Because UPnP is employed within the system, KODI also utilizes a peer-to-peer style in this section.

EXTERNAL INTERFACES

KODI's primary external interface is its GUI which allows users to access and manage their media libraries. Further, to be able to provide an optimal user experience on any operating system and hardware environment, KODI utilizes the following interfaces to accept a variety of remote input control:

- **EventServer API:** Used to communicate and control KODI. Allows KODI to support a multitude of input devices across multiple hardware platforms, provided the event client (Keyboard, mouse, WiiRemote, etc) is able to communicate using UDP [7].
- **Web Interfaces:** Allow users to control and interact with their KODI installation through a web browser. This can be used for remote control, library management and visual feedback, etc [8].
- **JSON-RPC API:** A Hypertext Transfer Protocol (HTTP) and/or raw Transmission Control Protocol (TCP) is a socket-based interface for communicating with KODI. It offers a more secure and robust mechanism in the same format as the traditional HTTP API and is based upon the JSON-RPC 2.0 specification [9]. It allows for concurrency within the software architecture of KODI.

KODI also utilizes the File Transfer Protocol as its default protocol for sharing media files between a computer and KODI device [10]. In addition, KODI supports HTTP, NFS, RSS, SFTP, SMB and UPnP for file/data sharing [11].

USE CASES

1. Users should be able to control their KODI interface with a multitude of input devices.

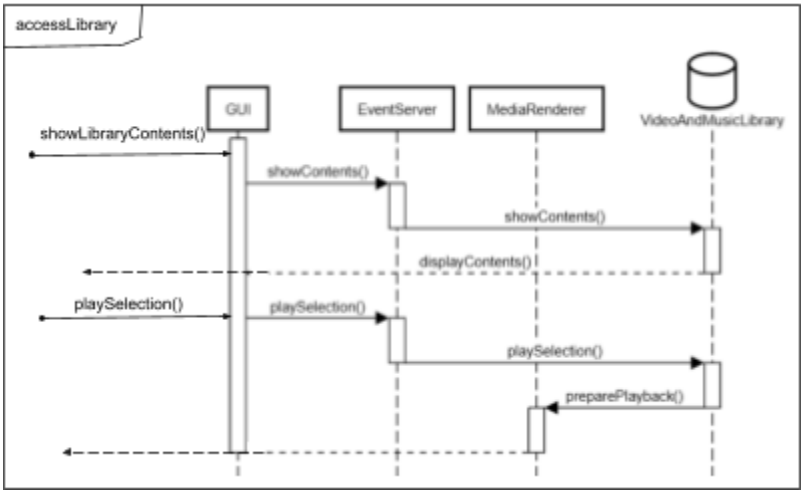

Figure 7: Sequence diagram illustrating how users can control their KODI interface

2. Users should be able to browse and play media from KODI servers on their network.

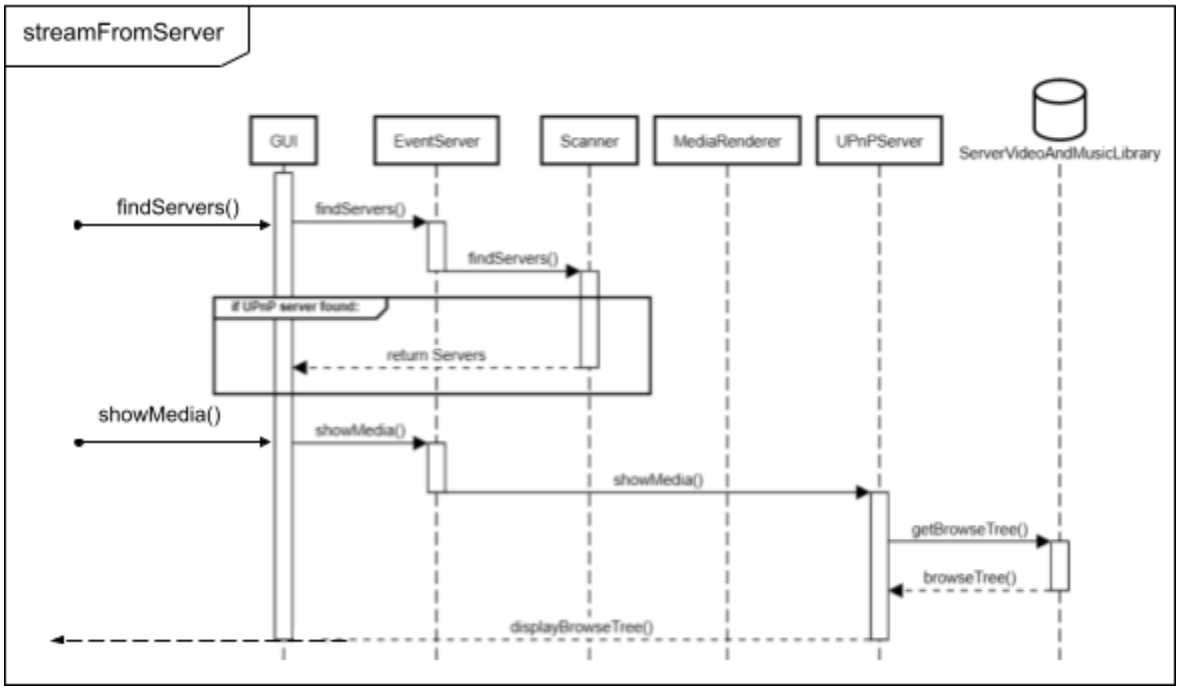

Figure 8: Sequence diagram illustrating how KODI can act as a UPnP server

- **Scanner:** Finds available clients on the network
- **UPnP:** Sends a browse tree to display files to the client player. The client player does not have direct access to the UPnP server files [12].

DATA DICTIONARY

Add-on

- A program/application that allows you to add a new feature, or to extend an existing feature, of KODI.

Modular style architecture

- A type of architecture style that consists a central data structure and a collection of independent components which operate on the central data structure

Layered style architecture

- A type of architecture where the components of a system are organized into different layers, each responsible for a specific set of functionality, and each layer provides services to the layer above it while relying on the services from the layer below.

NAMING CONVENTIONS

API - Application Programming Interface
XBMC - XBOX Media Centre
UPnP - Universal Plug and Play
HTTP - Hypertext Transfer Protocol
TCP - Transmission Control Protocol
GUI - Graphical User Interface
NFS - Network File System
RSS - Really Simple Syndication
SFTP - Secure File Transfer Protocol
SMB - Server Message Block
JSON-RPC - JavaScript Object Notation-Remote Procedure Call
OS - Operating System
IOS - iPhone Operating System

CONCLUSIONS

In conclusion, KODI is a large scale open source project with the goal of creating a versatile media player system that works on a variety of different media formats and isn't constrained nor restricted by factors of hardware or platform portability. With a core software architecture that's composed of a modular architecture employing a layered style. With the exception of its highest layer, the various layers of KODI's structure are largely separate with the lower layers of the implementation being exceptionally so, as the data flow within those layers is up and down within the same block. As such, it successfully eases the flow of large data files to minimize system workload, while allowing for a structure in which modularity and parallel development are feasible due to its layered structure. Furthermore, KODI incorporates a variety of external interfaces, among which is the JSON-RPC API which enables concurrency within KODI. Ultimately, KODI serves as a great demonstration of an effective software architecture that promotes modifiability, in particular within its large open source project setting, through its utilization of a modular architecture implemented in a layered style. For future implementations of large scale open sourced or group work related projects, a software architectural style incorporating a mixture of both modular and layered styles is an invaluable choice.

LESSONS LEARNED

With KODI we can see how the constructed architectural style of any given system greatly benefits from choosing the correct architectural style. More specifically, there are two core lessons that we've learned throughout our time looking into the software architecture of KODI.

- KODI's architecture is incredibly modular, and this allows users to install various add-ons and plugins to further increase its functionality. In particular, an example would be the external interface JSON-RPC API which enables concurrency capabilities for KODI.
- KODI separates its different modules into different layers and because of this separation of tasks/responsibilities, the codebase is far more modifiable and allows KODI to compile and run even if a non-essential module/library is disabled or removed. This greatly helps as KODI's an open source project and this allows for simultaneous development without affecting other parts of the system.

REFERENCES

[1] XBMC. "Kodi Code Guidelines." GitHub. Available: https://github.com/xbmc/xbmc/blob/master/docs/CODE_GUIDELINES.md. Accessed: Oct. 11, 2023.
[2] "Kodi: The Free and Open Source Home Theater Software," Delft Student Web Archive. Available: https://delftswa.github.io/chapters/kodi/. Accessed: Oct. 11, 2023.
[3] "Kodi Development," Kodi Wiki. Available: https://kodi.wiki/view/Development. Accessed: Oct. 12, 2023.
[4] "Kodi File Sharing," Kodi Wiki. Available: https://kodi.wiki/view/File_sharing. Accessed: Oct. 12, 2023.
[5] "Kodi Architecture," Kodi Wiki. Available: https://kodi.wiki/view/Architecture. Accessed: Oct. 16, 2023.
[6] "Kodi Glossary," Kodi Wiki. Available: https://kodi.wiki/view/Glossary. Accessed: Oct. 16, 2023.
[7] "EventServer," Kodi Wiki. Available: https://kodi.wiki/view/EventServer. Accessed: Oct. 17, 2023.
[8] "Web Interface," Kodi Wiki. Available: https://kodi.wiki/view/Web_interface. Accessed: Oct. 17, 2023
[9] "JSON-RPC API," Kodi Wiki. Available: https://kodi.wiki/view/JSON-RPC_API. Accessed: Oct. 17, 2023.
[10] "FTP," Kodi Wiki. Available: https://kodi.wiki/view/FTP. Accessed: Oct. 16, 2023.
[11] "UPnP," Kodi Wiki. Available https://kodi.wiki/view/UPnP. Accessed: Oct. 17, 2023.