KODI Concrete

Architecture Report

CISC 322

Sunday, November 19th, 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 concrete architecture of Kodi, a free open-source media player and entertainment platform. Starting with the derivation process where the Understand tool was used to analyze the source code's file structure and dependencies, enhancing understanding of the system's modularity. The top-level architecture follows a publish/subscribe style, emphasizing modularity and scalability, with key components like the Content Manager, Messaging module, and Controller serving specific functional roles. A detailed examination of the Player Core subsystem is provided, highlighting its role in multimedia content management and the interdependencies of its components such as the playercorefactory, retro player, and audio engine. Notably, the report identifies significant discrepancies between the conceptual and concrete architectures, including the unexpected complexity of module dependencies and the unique role of the Messaging module in the system's communication framework. Also, two sequence diagrams are observed, one for user requesting content, and the other for editing user information. This comprehensive analysis showcases the intricacies of Kodi's architecture, underscoring the depth and complexity not fully captured in the initial conceptual model.

INTRODUCTION & OVERVIEW

This report presents a detailed exploration of the concrete architecture of the Kodi application, a popular open-source media player and entertainment hub. The objective of this analysis is to provide an in-depth understanding of Kodi's architectural design, focusing on its core components, subsystems, and the intricate relationships between them. This investigation is pivotal to comprehend the structural nuances that enable Kodi's functionality and scalability.

The initial section of the report outlines the methodology employed to derive Kodi's concrete architecture. This process involved a meticulous examination of the app's source code using the Understand tool, which was instrumental in visualizing and analyzing the code structure and interdependencies. The focus was on assessing the system's organization, identifying key components, and understanding how they align with the previously established conceptual architecture, especially regarding modularity and layer interactions.

The report then delves into Kodi's top-level architecture, characterized by a publish/subscribe architectural style. This design choice facilitates modularity and scalability, two critical aspects of Kodi's architecture. Key components such as the Content Manager, Messaging module, Controller, Network module, and Utilities are discussed in detail, highlighting their roles, functionalities, and the nature of their interactions. The architecture's ability to handle various types of content and user requests efficiently is a focal point of this section.

A focused analysis of the Player Core subsystem is provided, underscoring its significance in Kodi's architecture. This subsystem is responsible for managing, processing, and playing multimedia content. Components like the playercorefactory, different media players (retro player, pa player, external player, and video player), data cache core, and audio engine are examined. The section emphasizes the dependencies and functionalities of these components, illustrating how they collectively contribute to Kodi's media handling capabilities.

The report also addresses discrepancies between the initial conceptual architecture and the observed concrete architecture of Kodi. This comparative analysis sheds light on aspects such as the unexpected complexity in module dependencies, the unique role of the Messaging module, and the unanticipated connections (or lack thereof) in the "Plugins" module. This section is crucial for understanding the differences between theoretical models and practical implementations.

Finally, two sequence diagrams are observed and examined, relating to common functions of the program, such as requesting content and editing user information. This allows for the comprehension of Kodi's execution, understanding exactly how functions work in order to achieve the desired result.

Thus, this report offers a comprehensive analysis of the Kodi application's concrete architecture, revealing its sophisticated structure and the dynamic interplay of its components. By dissecting the derivation process, examining the top-level architecture, and providing a detailed overview of the Player Core subsystem, the report not only presents a clear picture of Kodi's current architectural state but also identifies key areas where the conceptual and concrete architectures diverge. This analysis is essential for stakeholders seeking to enhance, maintain, or build upon Kodi's robust architectural framework.

DERIVATION PROCESS

Deriving the concrete architecture of Kodi consisted of multiple processes. Initially, using Understand proved complicated, as there are so many different functions and ways to analyze even the smallest dependencies. As such, we started off by going through the file structure of the source code. Understand helped us visualize how the actual source code was organized. One of the most useful parts of Understand, was the feature to see dependencies between directories. This allowed us to investigate our claims from the previous assignment. We were able to cross-reference these findings with our conceptual architecture, and see whether we were accurate as to what extent Kodi prioritized modularity in the architecture's design. This analysis helped us solidify our knowledge on how Kodi is structured. It also cleared up many questions we had regarding the interactions between the different layers of the system, which we will discuss in the discrepancies section of the report. In other words, we were able to see how well

our conceptual architecture mapped to the source code of Kodi. It was important to note which dependencies we lacked in our conceptual architecture that ended up being key in reality.

TOP LEVEL CONCRETE ARCHITECTURE

The architecture of Kodi follows a publish/subscribe architectural style. This means that by and large, components are split into publishers and subscribers. The publishers create and send messages, while the subscribers receive them. The advantage in the case of Kodi, is the potential for better scalability compared to a client-server architectural style. Furthermore, as we mentioned in our conceptual architecture report, Kodi prioritizes modularity. The publish/subscribe architectural style means that many components can operate independently of each other. This is what is known as loose coupling.

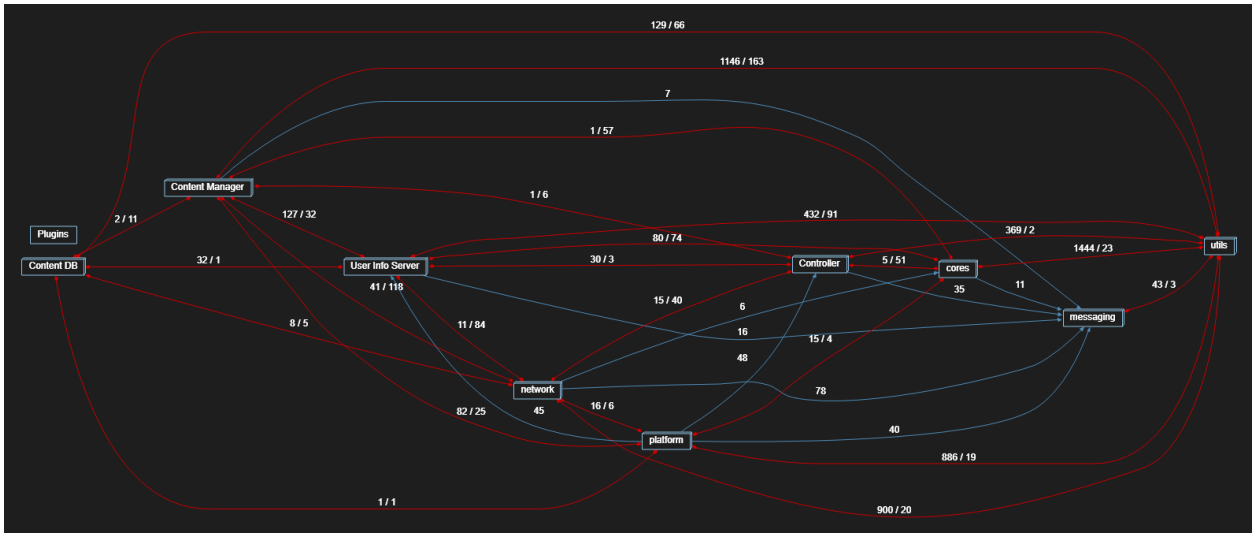

Figure 1: Kodi Architecture Diagram

In Figure 1, the Architecture of Kodi is represented through the nodes and relative dependencies. Following the Pub Sub architecture the Content Manager serves as the central node and interacts with the various other modules such as the User Info Server and Controller.

The responsibilities of the Content Manager are to manage the retrieval and storage of data in the Content Database as well as handling requests from the aforementioned modules. Kodi needs to be able to fetch whatever visual or audio entertainment the user owns. Content Manager has a large number of dependencies with other components, as it has to handle requests to access content, such is the job of a media player.

The Content Database stores pointers to the visual and audio playback. As such, it will receive requests from the Content Manager in order to access the form of entertainment the user desires.

The Messaging component is a node that has all but one two way communication, meaning that the role it provides is handling inter-service communication and message querying. The Messaging module is key for the publish subscribe architecture as one of the main advantages is loose coupling. With loose coupling, the components are weakly related causing changes to components to barely affect the performance of the program.

The Controller is connected with the Cores module, meaning that it is responsible for the processing of the core logic of the system. As the controller is connected to the Utilities component and the messaging component, it handles common platform functions and requests between many of the components.

The Network is the module related to routing the data through the internet. This comes with security responsibilities as the data needs to be safely trafficked to the endpoint. This could be between the user and the database when retrieving content to be played.

The Utilities node is connected with most modules handling common tasks shared between modules. This can be any helper functions or monitoring and logging information. By centralizing these utility services, the platform can promote code reusability, simplify maintenance, and ensure consistency in the execution of common tasks.

The User Info Server handles data specific to the user. This means that it is tasked with retrieving distinct data, like the user's preferences, location, and other information that varies between sessions.

The Platform links much of the network operations to the core processing. Kodi needs to be able to host its application on a solid foundation for the operations it uses.

The Cores component is tasked with any central processing Kodi requires. It includes the audio engine, and any specific Video Players Kodi has implemented.

It should be noted that the Plugins component of the concrete architecture has no dependencies. This is due to Kodi core not having any extra third party features by default. This will be further explained in the discrepancies section of the report.

PLAYER CORE SUBSYSTEM ARCHITECTURE

The subsystem we've chosen to focus on is the player core. The player core manages, processes controls and plays multimedia content. It acts as an interface between multimedia files

and the hardware or software required for their playback. In order to handle the multiple types of media that Kodi can play, playercorefactory exists in order to decide what type of player is best for the specific media, and will create instances of that player. This component depends on most of the functionality within this subsystem.

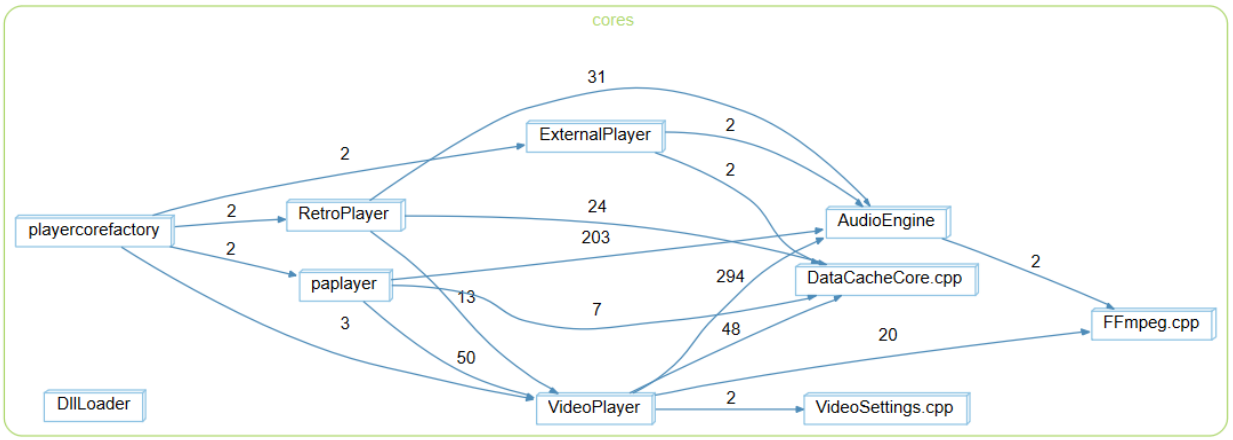

Figure 2: Player Core Architecture Diagram

The retro player, pa player, external player, and video player act as specific players for different types of media. All types of video playback get defaulted to the video player aside from special add-on cases. The retro player is used for handling game emulation. The paplayer is responsible for acting as an audio media player. The external player component is for when users want you to use an outside player for their content but want to use Kodi for organizing their content.

The four player components are dependent on the data cache core and audio engine, the data cache core is responsible for data caching and handling video and audio. It has no dependencies. The audio engine is responsible for processing and encoding audio. Many components are directly or indirectly dependent on the FFmpeg.cpp file which incorporates ffmpeg functionalities into Kodi.

This subsystem architecture was only given a short, high level analysis in our last report. We described it as “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”. While this was a short description, our conceptual architecture for the subsystem does not contradict our concrete architecture. The architecture styles of this subsystem follow a pipeline and modular architecture style as different forms of media are often processed, decoded/encoded, and funneled through different modules before getting sent to the presentation layer of the system.

DISCREPANCIES

There were many notable differences between our proposed conceptual architecture and the observed concrete architecture shown above. One of the largest discrepancies was the use of the Messages module to pass messages in a pub-sub style throughout the code.

In our conceptual architecture, one of the use cases we studied was playing content from a local source. In this conceptual use case, we predicted that the Controller connected directly to the Content Manager, which then connected directly to the Directory Searcher, ending at the Local Directories. However, looking at the concrete architecture above, we see that Kodi more accurately receives commands from the Controller which are then sent to the Messaging module. Messages in this module are held in the m_vecMessages data structure, where they wait to be processed via calls to ProcessMessages() From here, the commands are dispatched to the appropriate module to retrieve the data. In this case, the message will be dispatched to the Content Manager, which will then use the Platform module to get platform-specific file data. Then, a response message is created, going in the opposite direction. For this case, the Content Manager creates a message and sends it to the Messenger module. This message is then dispatched back to the Controller, where the content is displayed to the user.

Another large discrepancy is that the general “Plugins” module, which represents any third-party plugins made for Kodi, does not have any connections. This is because there are no plugins of this kind in the Kodi core, meaning that there are technically no dependencies. This of course would change when plugins are loaded, as there will likely be dependencies with the Messenger module at the very least to allow the plugin to communicate with the other modules in Kodi.

The final discrepancy found between our conceptual architecture and the discovered concrete architecture was a much larger number of dependencies between all modules. In our conceptual model, most modules had only a few dependencies on things like the Messenger, Utils, and Platform, and they used these to communicate. However, the concrete architecture is far more complex. There are many extra dependencies, both unidirectional and bidirectional, between modules that we did not expect. For example, the Platform module has a bidirectional dependency directly to the Content Manager. There are many other cases throughout the codebase that further illustrate this point. Overall, the complexity of the dependencies within Kodi were underestimated in our conceptual architecture.

SEQUENCE DIAGRAMS

Diagram One: Requesting content

Actors: Users

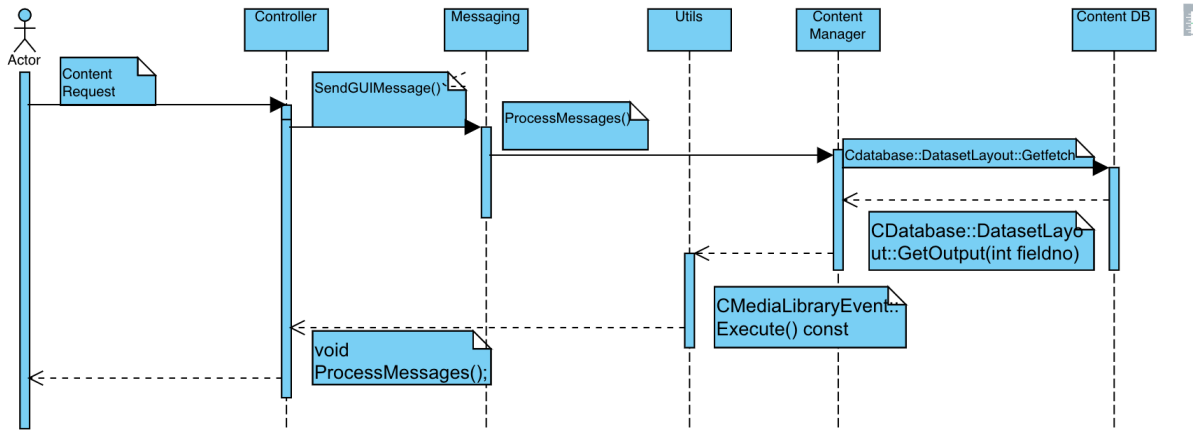

Figure 3: Sequence diagram for requesting content

This diagram illustrates how a user would request to view content. It starts by invoking the function SendGUIMessage(). This function is triggered by an input, likely the user clicking on the content he would like to view. The message is then set to the messenger to be processed and sent to the content manager. From the content manager the content database is then queried. The query results are then sent back to the content manager, where the content is processed with the function CMediaLibraryEvent. This function returns a boolean value of whether the processing was successful. This is sent back to the controller to be displayed to the user.

Diagram Two: Editing User information

Actors: Users

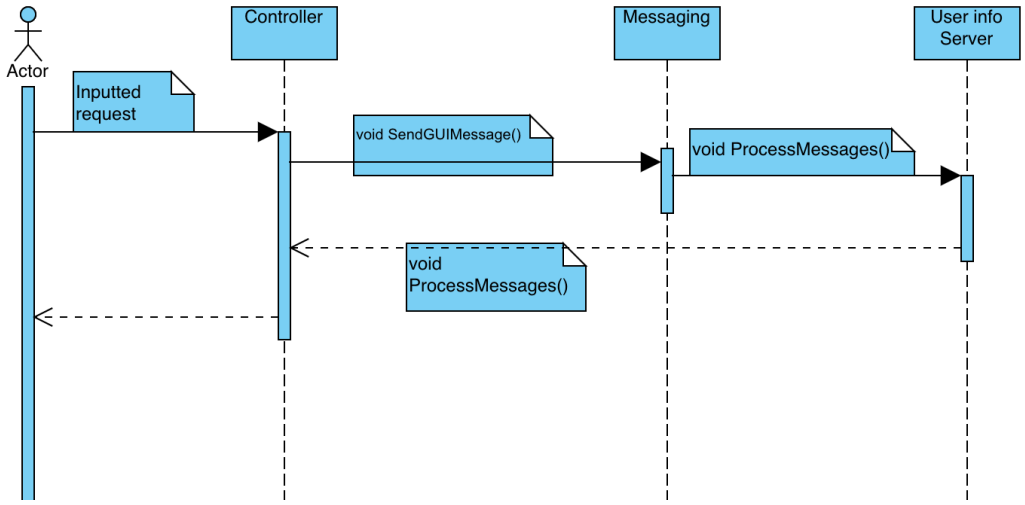

Figure 4: Sequence diagram for a user editing his information

This diagram illustrates the scenario where a user would like to edit his personal information. It starts from the user inputting the information that he would like to edit. The controller then sends this information to the messenger where it is processed. From there the processed information is then sent to the user info server. The server returns a message back on whether the edit was successful. This message is then displayed to the user from the controller.

DATA DICTIONARY

Media Player: Software for playing multimedia computer files like audio and video files.
Understand: Software development tool that allows you to perform static code analysis, edit and refactor code, view dependency graphs, etc.
Publish/Subscribe Architecture: An architectural style where components (publishers) create and send messages, which are received by other components (subscribers).
Player Core: The central subsystem in Kodi responsible for managing, processing, and playing multimedia content.
Playercorefactory: A component in Kodi's Player Core that decides the most suitable player type for a given media and creates instances of that player.
Messaging Module: A component in Kodi that handles inter-service communication and message querying, playing a crucial role in the application's publish-subscribe architecture.
Content Manager: A key module in Kodi responsible for managing the retrieval and storage of data in the Content Database and handling requests from various modules.
Content Database: A storage system in Kodi that keeps pointers to audio and video playback files, facilitating access to media content.
Controller: A component in Kodi that processes the core logic of the system and handles requests between many other components.

Network Module: Responsible for routing data through the internet, including managing the security of data traffic.
Utilities Module: Centralizes common tasks shared between modules in Kodi, like helper functions, monitoring, and logging information.
User Info Server: Manages data specific to the user, such as preferences and session-based information.
Platform Module: Links network operations to core processing in Kodi and provides a foundation for the application's operations.
Cores Module: Handles central processing in Kodi, including audio engine and video players.
Plugins Module: Represents third-party plugins made for Kodi, capable of extending the application's functionality.

NAMING CONVENTIONS

GUI: Graphical User Interface

CONCLUSIONS

The intricate and modular structure of Kodi's system was the main focus of this in-depth examination of its concrete architecture. Kodi's publish/subscribe top-level architecture is examined in detail, and the report's systematic derivation methodology reveals the complexity and subtleties of the design. The Player Core subsystem is a prime example of the application's versatility and interdependence, as well as its ability to manage multimedia content effectively. Moreover, the discovery of significant differences between the conceptual and physical designs highlights the need for theoretical model implementation's evolution and practical obstacles. In addition, two sequence diagrams were observed and examined, allowing us to visualize the execution of functions that relate to content retrieval and user information changes. This thorough analysis not only improves knowledge of Kodi's architectural foundation but also offers insightful advice for future development and application optimization, guaranteeing the app's continuous applicability and efficacy in the ever-evolving field of media entertainment technology.

LESSONS LEARNED

One of the biggest challenges we faced in this report was processing the initial overload of information. While our conceptual architecture was largely similar to the concrete architecture, we thoroughly underestimated the sheer number of components and the dependencies between them. As such, when we first started using Understand to analyze much of the source code, we were faced with so much new information about Kodi that it seemed daunting. If we were able to redo this report, I think we would begin our analysis much more

methodically. We would go straight to noting the overall file structure, and then map the dependencies between components after we had a solid general understanding of what each section of the source code represented.

Once we had a good grasp on Kodi structure, we were then able to more clearly see what our conceptual architecture lacked. Namely, we underestimated how important the messaging module would be. The messaging module shapes how information flows throughout Kodi, and is a key part of why Kodi is designed in the way it is. Realizing this helped us understand the importance of figuring out the life cycle of information in your program early on. If we were to build an application now, we would invest a significant amount of time making sure that even in our planning and conceptual architecture phases we considered the flow of data between our major modules.

REFERENCES

"Add-on Development." Kodi Wiki, https://kodi.wiki/view/Add-on_development, Accessed 18 Nov. 2023.

Kodi Community Guidelines, https://kodi.wiki/view/Official:Kodi_Community_Guides. Accessed 18 Nov. 2023.

“Kodi Forum.” *Kodi Community Forum*, https://forum.kodi.tv/. Accessed 15 Nov. 2023.

Kodi Wiki. Official Kodi Wiki. (n.d.). https://kodi.wiki/view/Main_Page. Accessed 15 Nov. 2023.

Xbmc. “XBMC.” *GitHub*, https://github.com/xbmc/xbmc. Accessed 19 Nov. 2023.