Developing a DVB-I Parser Library in Dart and GUI App in Flutter

1st Luis Hebendanz
M. Sc. Computer Science
TU Berlin
Berlin, Germany

2nd Niklas
M. Sc. Information Systems Management
TU Berlin
Berlin, Germany

3rd Balint Rüb
M. Sc. Information Systems Management
TU Berlin
Berlin, Germany

***Abstract*—In this project, we aimed to develop an efficient DVB-I parser library in Dart and a GUI app in Flutter to present TV services on Android devices.**

We used the Dart programming language to write the DVB-I parser library and Google's cross-platform Flutter framework to develop the GUI app.

However, we faced challenges in developing the GUI app in Flutter due to multiple bugs in the libraries we used and sparse documentation. Despite these challenges, we were able to develop a working app to present TV services on Android devices.

Our project demonstrates the feasibility of using Dart to write an efficient DVB-I parser library and Flutter to develop a GUI app that presents TV services on Android devices. However the challenges we faced in developing the GUI app highlight the importance of mature libraries and documentation to support developers in using these technologies. The lack of maturity in the Flutter ecosystem has compelled us not to recommend it for further projects.

***Index Terms*—IP-TV, DVB-I, Flutter, Dart, Cross Platform**

I. Introduction

In this project, we aimed to develop an efficient DVB-I parser library in Dart and a GUI app in Flutter to present TV services on Android devices. The DVB-I standard is a standards-based solution for delivering television via the internet and offers a discovery mechanism to signal and discover television services, using a set of REST APIs allowing clients to retrieve a list of services in an XML-based format. Our primary objective was to create a parser library that can efficiently handle the DVB-I service list registry and provide all the necessary information required to present the TV service in the client app. Additionally, we aimed to create an Android GUI app using Flutter that uses the DVB-I parser library to present the TV services to the user.

The development of the DVB-I parser library involved reading the DVB-I standard and manually emulating REST requests as specified by the specification. We also familiarized ourselves with the Dart programming language and experimented with simple coding examples to gain proficiency with the language. Once we had a good understanding of the standard and the language, we designed and developed the DVB-I parser library using Dart, implementing the required REST APIs and XML parsing.

The development of the Android GUI app using Flutter involved building an intuitive user interface to present the TV services to the user, as well as incorporating the DVB-I parser library to retrieve and display the information for each service. We faced challenges while developing the app, including multiple bugs in libraries used and sparse documentation, which affected the app's functionality and usability.

In this report, we will assess the success of our project in achieving its goals and evaluate the performance and usability of the DVB-I parser library and Android GUI app. We will also discuss the challenges encountered during development and recommend future improvements to enhance the overall functionality and usability of the app.

II. Scientific Background

This section will provide an overview of the utilized technologies and their respective functionalities.

III. Flutter Framework

Flutter is an open-source mobile app development framework created by Google that allows developers to create high-performance, cross-platform mobile applications for iOS, Android, and other platforms from a single codebase.

Flutter uses a programming language called Dart, which is also developed by Google, and provides a rich set of pre-built UI widgets and tools that allow developers to create visually appealing and interactive mobile applications. The framework uses a reactive programming model, which means that changes in the app's state are automatically reflected in the UI, making it easy to build dynamic user interfaces.

One of the key benefits of Flutter is the hot-reload feature, which allows developers to quickly see the changes they make to the code in real-time on the app, without having to rebuild the entire application. This significantly speeds up the development process and makes it easier for developers to experiment with different UI designs and functionality.

Flutter is also known for its fast development speed and ease of use, making it an ideal choice for startups and businesses that need to quickly develop and deploy high-quality mobile applications.

In addition to mobile app development, Flutter can also be used for building desktop and web applications, thanks to its platform-independent nature. Overall, Flutter is a powerful and flexible mobile app development framework that is rapidly gaining popularity in the developer community.

IV. Flutter Architecture Overview

Flutter is a cross-platform framework for building mobile and desktop applications. Its architecture consists of four main layers: the Dart app layer, the framework layer, the engine layer, and the platform layer, shown in figure 1.

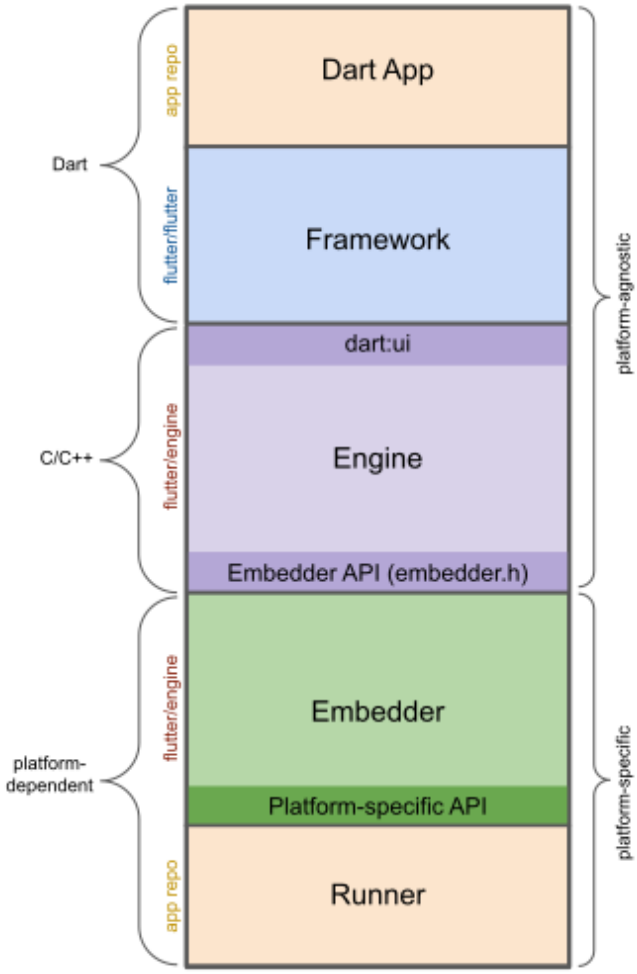

Fig. 1. Flutter architectural overview [1]

The Dart app layer is responsible for composing widgets into the desired UI and implementing business logic. It is owned by the app developer.

The framework layer provides a higher-level API for building UI apps, including widgets, hit-testing, gesture detection, accessibility, and text input. It composites the apps widget tree into a scene.

The engine layer is responsible for rasterizing composited scenes and provides low-level implementation of Flutter's core APIs, including graphics, text layout, and the Dart runtime. It exposes its functionality to the framework using the dart:ui API and integrates with a specific platform using the platform layer.

The platform layer "is the native OS application that hosts all Flutter content and acts as the glue between the host operating system and Flutter" [1]. Flutter includes platform embedders for each of the target platforms, and you can also create a custom platform embedder.

In summary, Flutter's architecture provides a robust and efficient rendering pipeline, bypassing system UI widget libraries and using its own widget set and Skia 2D library [10] for rendering. This results in a high-performance, cross-platform framework with minimal abstractions and overhead.

V. Integrating Flutter with other code

Flutter and Dart provide several ways to integrate with other code written in different programming languages.

For mobile and desktop apps, Flutter provides platform channels, which are a mechanism for communicating between Dart code and the platform-specific code of the host app. This allows developers to call custom code written in languages like Kotlin, Swift, or C-based APIs, including those generated for code written in modern languages like Rust or Go. The platform channel mechanism serializes Dart types into a common message format, which is sent to the receiving code that then deserializes it into a programming language-specific object displayed in figure 2.

For C-based APIs, Dart provides a direct mechanism for binding to native code using the dart:ffi library, which can be considerably faster than platform channels since no serialization is required to pass data.

For web apps, the js package serves a similar purpose to dart:ffi for C-based APIs, allowing developers to bind to native code directly from Dart. Additionally, since web apps run in the browser, developers can also use Dart's interop capabilities to integrate with JavaScript libraries, either by importing JavaScript libraries into Dart or by calling Dart code from JavaScript.

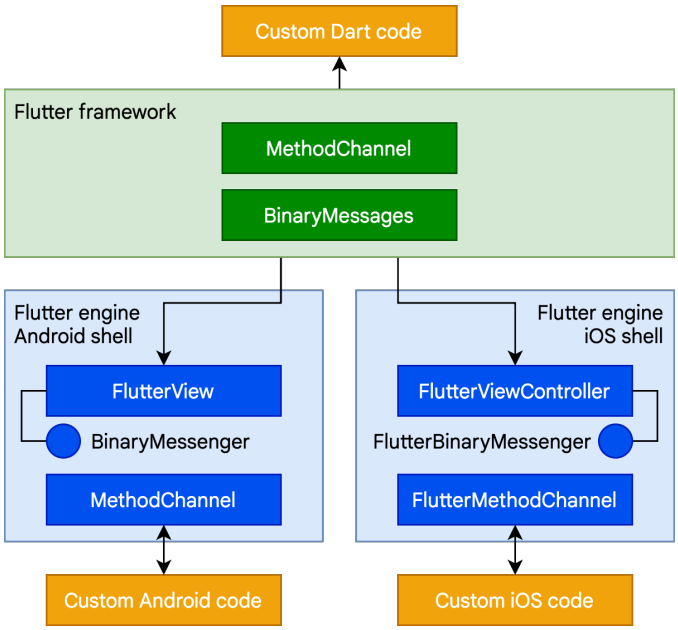

Fig. 2. Platform Channels Overview [1]

Overall, Flutter and Dart provide a variety of interoperability mechanisms that allow developers to integrate their code with other programming languages and platforms, providing flexibility and extensibility to their applications.

VI. State Management in Flutter

State management is an essential aspect of building robust and performant user interfaces in Flutter. The setState() method is the most straightforward way to manage state within a single widget. When called, setState() rebuilds the widget tree, which results in the updated state being reflected in the UI.

Sharing state between widgets is a common use case in Flutter applications, and it can be achieved using the setState() method by lifting the state up to a common ancestor widget, as shown in Figure 3 . This involves passing callbacks down the widget tree, which can be called to update the state of the ancestor widget. However, this approach can become unwieldy and result in a cluttered codebase when dealing with large and complex widget trees.

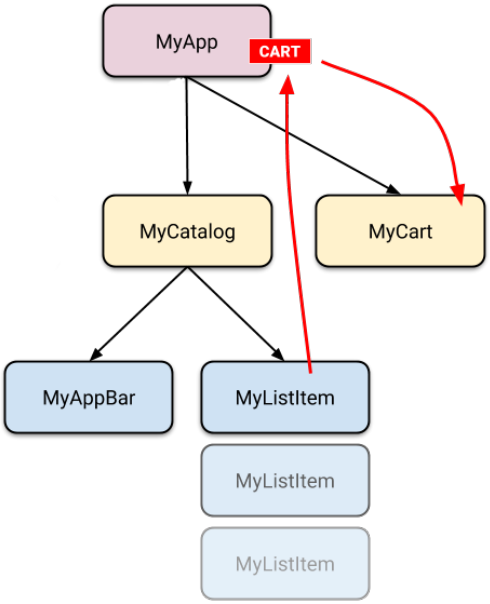

Fig. 3. State Tree

The provider package offers a more robust and flexible approach to state management in Flutter applications. It provides a mechanism for sharing state across the widget tree using ChangeNotifier and ChangeNotifierProvider. ChangeNotifier is a simple class included in the Flutter SDK that provides change notification to its listeners. ChangeNotifierProvider is the widget that provides an instance of a ChangeNotifier to its descendants.

To use provider, a developer simply wraps the widget subtree that requires access to the shared state with a ChangeNotifierProvider widget. Consumers of the state can then use the Consumer widget, which listens for changes to the state and rebuilds the corresponding part of the widget tree when necessary. By using provider, developers can avoid passing callbacks down the widget tree and instead focus on building composable and maintainable widgets.

VII. Dart Language

Dart is a class-based, object-oriented programming language with syntax that is similar to Java-style languages. It is optimized for client-side web and mobile app development, but can also be used for server-side programming. Some key concepts are:

1) Everything in Dart is an object, and every object is an instance of a class.
2) Dart has a garbage collector that automatically frees memory that is no longer in use.
3) Type annotations are optional in Dart because the language can infer types.
4) If you enable null safety in Dart, variables cannot contain null values unless you explicitly make them nullable with a question mark (?).
5) Dart does not have keywords for public, protected, and private access, however an identifier starting with an underscore (_) is private to its library.
6) Dart supports concurrent programming with async-await, Future, and Stream objects.
7) A Future in Dart represents the result of an asynchronous operation and can be completed with a value or an error.

VIII. Dart Asynchronous Programming

In Dart, async/await is a powerful mechanism that allows for writing asynchronous code in a synchronous style. With async/await, developers can write code that doesn't block the application's main event loop while waiting for I/O operations, network requests, or other time-consuming tasks to complete.

To use async/await in Dart, developers mark functions as "async" and use the "await" keyword to wait for the result of an asynchronous operation. When a function is marked as "async", it returns a Future object that can be used to obtain the result of the computation when it completes. The "await" keyword is used to wait for the Future to complete, and once it does, the result is returned as if it were a normal synchronous operation.

Async/await is particularly useful in situations where multiple asynchronous operations need to be performed in sequence, and the result of each operation is dependent on the completion of the previous one.

IX. DVB-I Standard

The DVB-I (Digital Video Broadcasting - Internet) [14] protocol is a standard developed by the Digital Video Broadcasting (DVB) [13] organization. The DVB-I standard lays out processes to allow internet connected devices, such as smartphones, tablets, laptops, and smart TVs, to receive broadcast quality content, including live and on-demand TV channels. It aims to provide simpler and more accessible options for

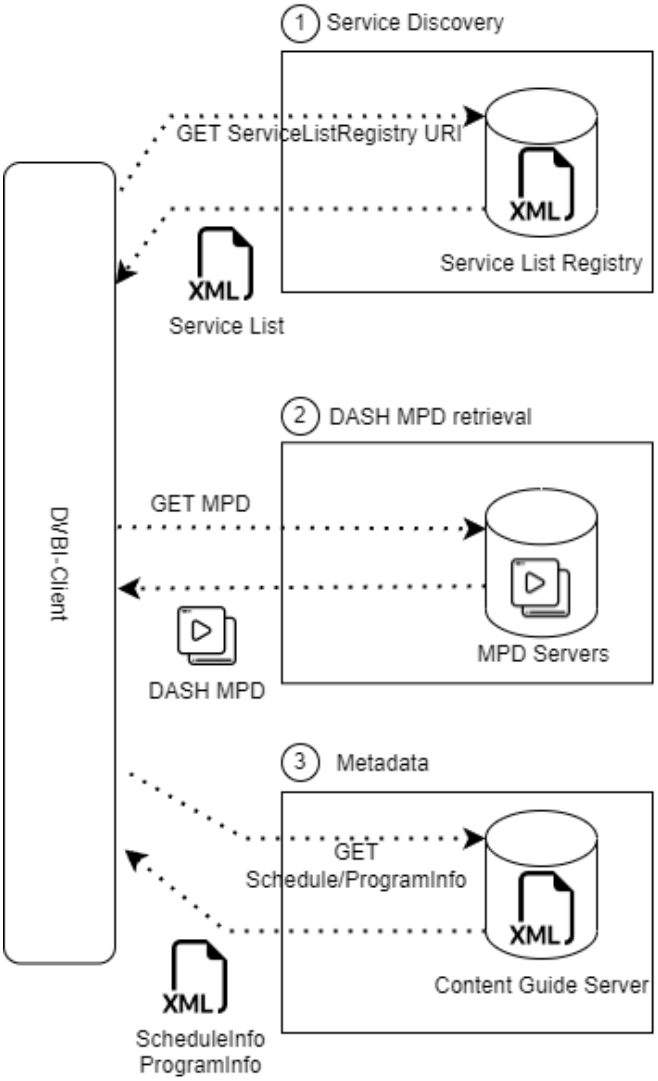

Fig. 4. DVBI-I architecture and client interaction overview

viewing both linear and Video-On-Demand (VOD) streaming services on any internet-connected device.

The DVB-I standard consists of many components. Figure 4 illustrates the elements most relevant to our project. At the core of the DVB-I protocol is the ServiceListRegistry (SLR), illustrated in ①. The SLR provides a REST API allowing clients to retrieve a list of services (ServiceList) in an XML-based format, including all information required to present the TV service in the client. The <*Service*> element of the ServiceList XML document holds metadata about a specific TV channel, such as the channel name, the channel logo URL, and the DASH MPD stream URL. The DASH MPD can be retrieved by making a request to an MPD server, as shown in ②.

The <*ContentGuideSource*> element, which can be referenced by multiple <*Service*> tags, provides information about the ProgramInfo and ScheduleInfo endpoints, which can be queried to retrieve more detailed information about the TV shows airing on the channel.

To retrieve scheduling information about the TV shows airing on a channel or additional programming information clients can send a GET HTTP request to a Content Guide Server, represented in ③. Requests for scheduling info should include the service element ID (sid), the start unix time, and the end unix time of the time period for which scheduling data is desired. The response to this request is a new XML document containing multiple <*ProgramInformation*> tags, each of which provide important details about the TV shows airing on the channel, such as the main title, duration, and start time.

Clients can also query a ProgramInfo endpoint to retrieve additional metadata about the TV shows airing on the channel, including the minimum age requirement, genre, and a longer description. This information can be useful for clients in presenting a more detailed schedule of TV shows to the user.

Overall, the DVB-I protocol provides a standard mechanism for delivering linear television services over the internet, with the aim of providing a seamless viewing experience across multiple devices without any limitations. The use of REST APIs and XML-based formats for delivering information about TV channels and shows enables clients to easily retrieve the information they need to present a comprehensive TV service to the user.

X. Development Setup

For Linux the Nix package manager has been found to provide an efficient and organized approach to setting up a development environment. This method involves installing the Nix package manager as outlined in the project's README, followed by executing the `nix develop` command to obtain all necessary dependencies and enter into a virtual environment akin to that of Python. The configuration of the development environment is defined in the `flake.nix` file, which enables developers to specify the required software and tools, including their respective versions. Furthermore, this approach includes preconfigured tools such as VSCodium with preinstalled extensions, the Android SDK, and the Flutter compiler, resulting in a streamlined and time-efficient setup process.

In summary, the utilization of Nix and its development environment provides an effective strategy for minimizing the complexity of setting up a development environment, ensuring reproducibility, and creating a dependable development environment.

XI. App Architecture

The project is hosted on *Github*[1], and the code is organized into four main directories: dvbi_client, dvbi_lib, video_player, and chewie. The dvbi_client directory serves as the top-level node and contains the graphical user interface implementation from which the main application is compiled. The dvbi_lib directory implements a DVB-I parser library and a command-line interface to test the parser. The video_player library is

[1]https://github.com/AWT-DVBI/dvbi_client

a wrapper for the Java Exoplayer that exposes a Flutter-compatible API. Finally, the chewie library is used to implement a basic graphical user interface for the video player.

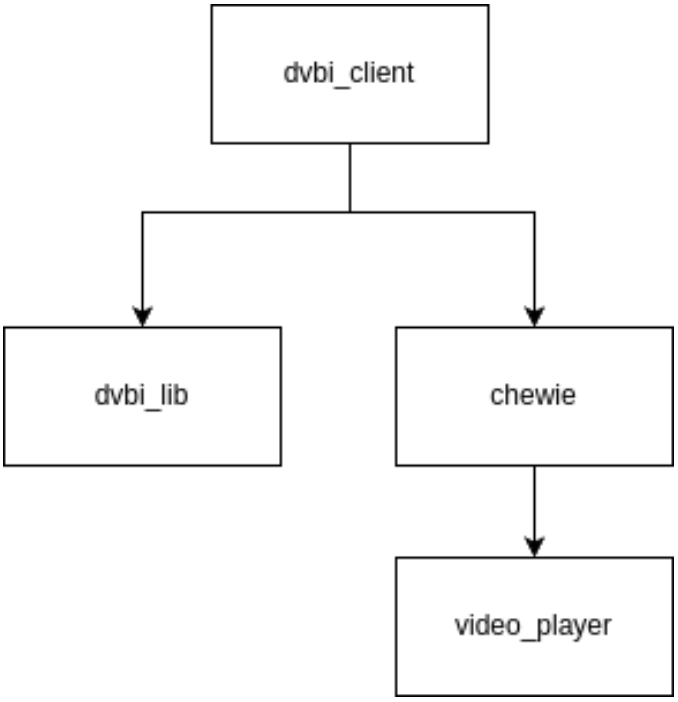

Fig. 5. DVB-I Dependency Structure

This development project utilizes a direct dependency approach to include dependencies into the repository, which allows for quicker changes and bug fixes. The Chewie library is used as a wrapper for the Java Exoplayer and includes two main components: the ChewieController and the MaterialControls UI component. In this project, the Android MaterialControl object is copied and modified to fit the project's specific needs. The customized MaterialControls UI component is initialized as a custom video overlay UI object called MyMaterialControls, as seen in Figure 6 .

```
122   void initializeEverything() async {
123     videoControls = MyMaterialControls(
124       showPlayButton: true,
125       nextSrc: nextChannel,
126       prevSrc: prevChannel,
127     );
128     await initializeSources();
129     initializePlayer();
130   }
```

Fig. 6. Custom video overlay initialization

XII. Feature: DVB-I Parser

An important part of our implementation is the dvbi parsing library. We created the library to make *HTTP calls*[2] to the serviceList servers and parse the responses. As the responses of the server are in xml format, we use the *XML dart package*[3] to parse the responses into dart and json objects. The services of the serviceList i.e. were parsed into serviceElement objects which contained meta data like servicename, id etc (shown in figure 7). Furthermore, they contained schedule and programinfo endpoints which where used to retireve even more meta data. This includes scheduling information and extended program information. The dvb-i parser library is an external dart library and can be seamlessly imported into other projects. Furthermore, it uses lazy loading this means that only the requested data you need will be loaded which improves the performance. By using asynchronous operations, our program can continue executing tasks while waiting for other long-running operations, such as fetching data over a network, to complete.

```
lhebendanz@qubasa-desktop ~/Projects/dvbi_client/dvbi_lib ‹gui*›
→ dart lib/main.dart -e https://dvb-i.net/production/services.php/de | jq -C | head -n40
[
  {
    "serviceName": "Das Erste HD",
    "uniqueIdentifier": "tag:mitxp.com,2021:1.1019.10301",
    "providerName": "ARD",
    "contentGuideSourceElem": {
      "scheduleInfoEndpoint": "https://dvb-i.net/production/schedule.php/de/ard",
      "programInfoEndpoint": "https://dvb-i.net/production/program.php/de/ard",
      "providerName": "ARD-Playout-Center",
      "cgsid": "ard-iSIMS"
    },
    "dashmpd": "https://mcdn.daserste.de/daserste/dash/manifest.mpd",
    "logo": "https://itv-api.ard.de/ardstart/img/services/28106.png"
  },
  {
    "serviceName": "arte HD",
    "uniqueIdentifier": "tag:mitxp.com,2021:1.1019.10302",
    "providerName": "ARD",
    "contentGuideSourceElem": {
      "scheduleInfoEndpoint": "https://dvb-i.net/production/schedule.php/de/ard",
      "programInfoEndpoint": "https://dvb-i.net/production/program.php/de/ard",
      "providerName": "ARD-Playout-Center",
      "cgsid": "ard-iSIMS"
    },
    "dashmpd": "https://arteliveext.akamaized.net/dash/live/2031004/artelive_de/dash.mpd",
    "logo": "https://itv-api.ard.de/ardstart/img/services/28724.png"
  },
  {
    "serviceName": "SWR BW HD",
    "uniqueIdentifier": "tag:mitxp.com,2021:1.1019.10303",
    "providerName": "ARD",
    "contentGuideSourceElem": {
      "scheduleInfoEndpoint": "https://dvb-i.net/production/schedule.php/de/ard",
      "programInfoEndpoint": "https://dvb-i.net/production/program.php/de/ard",
      "providerName": "ARD-Playout-Center",
      "cgsid": "ard-iSIMS"
    },
    "dashmpd": "https://swrbw-dash.akamaized.net/dash/live/2018674/swrbwd/manifest.mpd",
    "logo": "https://itv-api.ard.de/ardstart/img/services/28113.png"
  },
lhebendanz@qubasa-desktop ~/Projects/dvbi_client/dvbi_lib ‹gui*›
→
```

Fig. 7. Overview serviceElement json

A. DVB-I Parser: Schedule Info

We used the ScheduleInfoEndpoint described in [14], as shown in Figure 8 to retrieve scheduleinfo meta data for specific programs.

Every ServiceElement Object of our implementation can request more metadata by accessing the scheduleInfo Dart object. By default the schedule information for the next 6 hours gets fetched on object access. However, it's easy to extend this time window. To simplify further usage, we parse the duration information of a program from a string to a double. This is

[2]https://pub.dev/packages/http

[3]https://pub.dev/packages/xml

necessary because the original string representation follows the ISO 8601-1 format [9], which is not very human-readable.

During implementation, we discovered that the provided endpoint did not return the correct metadata. After conducting several tests with different variations of the endpoint, we found that substituting the word "production" in the URL with "staging" resulted in the correct scheduling information.

URL format:

```
<ScheduleInfoEndpoint>?start=<start_unixtime>&end=<end_unixtime>
&sid=<service_id>&image_variant=<variant>
```

Fig. 8. Overview scheduleInfoEndpoint [14]

1) DVB-I Parser: Detailed Program Info : To obtain additional metadata for a specific program of a particular service, we utilized an HTTP request to the ProgramInfo-Endpoint, as depicted in Figure 9 . We used the programId (crid), which was obtained through the schedulingInfo request. Similar to the issues encountered during the schedulingInfo request, we faced data problems while using the URL with "production" in the path, and had to replace it with "staging." However, we still encountered a problem where the Endpoint did not respond with all the metadata that could be displayed by the DVB-I Client. We were only able to retrieve a limited amount of additional information, such as the longer text description of the program.

The request URL shall be composed as follows:

```
<ProgramInfoEndpoint>?pid=<program_id>&image_variant=<variant>
```

Fig. 9. Overview scheduleInfoEndpoint [14]

2) DVB-I Parser: Optimization: Initially, our implementation had a runtime of $O(n^2)$ due to the requirement of matching each ScheduleEvent field with its corresponding program field marked by CRID. This was causing high execution time, as we were iterating through the entire XML tree using the XML Dart library.

However, we were able to significantly improve the performance of our program by utilizing a HashMap. This allowed us to retrieve the metadata of each service element at once, reducing the time from 10 minutes to just 30 seconds. It is important to note that this does not mean the user of our application will have to wait for 30 seconds to see any program results, as we use asynchronous functions.

XIII. Challenges: Parser

During the implementation of the DVB-I Parser Library several challenges were encountered. One significant challenge of the DVB-I parser library, as outlined in Section XII-A2, was the limited scalability when retrieving multiple meta-information for various services and their associated programs. We addressed this challenge by utilizing a hash map and reducing the time interval for displaying programs from the next 24 hours to 6 hours. To prevent long waiting times for users of the DVB-I client, we used the concept of Dart streams and asynchronous functions to decrease the loading time for channels. Another issue faced during the implementation of the DVB-I parser library was the incorrect information response from the official endpoints, as discussed in Section XIV. It appears that the production environment of the endpoint is not up-to-date and needs to be updated soon.

Furthermore, the responses from the endpoint varied significantly regarding the presence or absence of specific information. This is because the DVB-I standard allows for a wide range of possibilities. Consequently, our code required robustness and flexibility to handle null values and avoid errors in the client interface.

XIV. Features: GUI

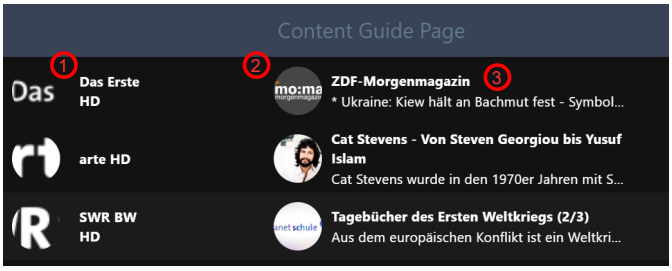

Fig. 10. Overview of the Content Guide Page: ① Title and logo (if available) of channel, ② logo of currently running program, ③ title and brief description of currently running program

The second major component of our application is a GUI that renders information retrieved and parsed from *ServiceListRegistry*, *ContentGuide*, and *MPD* servers in a visually appealing way as well as let users interact with said information through interfaces similar to terrestrial television. To this end, we developed a **Content Guide Page** and a **Video Player** page.

A. Content Guide Page

Figure 10 illustrates the Content Guide Page. It presents a user with an overview of all available channels, gives visual and textual context regarding each channel and the currently playing program (illustrated in ① and ②), and gives the title and a brief textual description of the currently playing program, illustrated in ③.

The app briefly loads the entire ServiceList at start up. The Content Guide Page is populated with information found in the ServiceList elements. Requests for further information about each running program is done asynchronously, allowing users to browse through the Content Guide Page while more information is being fetched. This ensures the GUI stays responsive even on lower powered hardware devices. Once a user has found a program they desire to watch, they can select the program. The **Video Player** page is then loaded to play the selected channel.

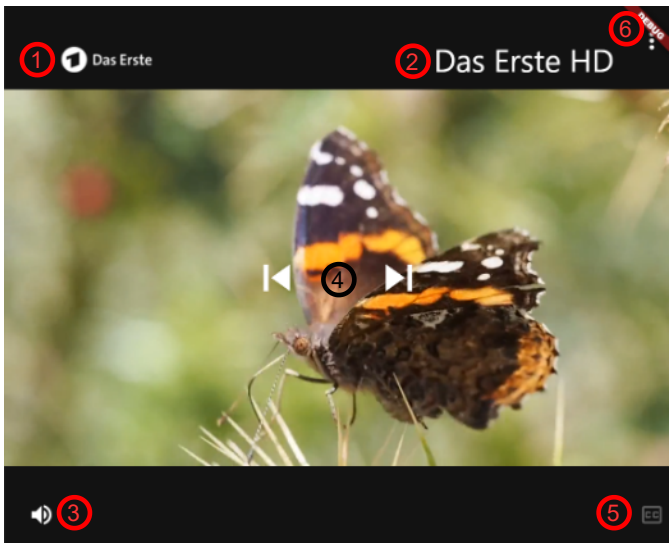

Fig. 11. Overview of the Video Player: ① Channel logo, ② name of channel, ③ mute button, ④ previous and next channel buttons, ⑤ closed captioning button, ⑥ extra options (playback speed)

B. Video Player

The **Video Player** page plays back DASH MPD files retrieved from MPD servers, displays contextual information about the currently running program, lets users switch channels, as well as mute and unmute.

Figure 11 illustrates the video player in more detail. ① and ② give contextual information about the current channel through the channel logo and channel name respectively. ③ allows the user to mute and unmute the program. ④ show the previous and next channel buttons. Using these allows the user to browse between channels. ⑤ illustrates the closed captioning button. This is currently not available but is part of potential future work. ⑥ brings up a menu for additional video playback options, such as video playback speed. This feature is also part of our future work.

We use the Chewie [11] library to display a widget that plays back video. The Chewie library is built ontop of the video player [12] library. Ontop of the video we overlay buttons to browse between channels. The Video Player Controller listens to a state that holds the current playing video. The channel browsing buttons change the state, which causes the video player to re-render.

XV. Challenges: GUI

A. State Management

One of the major issues we ran into during development was properly managing state in Flutter. Figure 12 illustrates how Flutter renders its UI. Flutter uses a declarative approach to UI programming, meaning you describe what the UI should look like for any given state. The widgets that depend on state are re-rendered whenever that state changes.

Fig. 12. UI rendering in Flutter: The UI is a function of app state [16]

Initially, we wanted to provide the contents of a ServiceList as a stream, dynamically loading, parsing, and rendering elements as they arrive. Using streams instead of loading all Service Elements at once would make our UI more responsive and save on memory. To render the GUI anew whenever new elements of the stream arrived, we initially encapsulated the stream in a StreamProvider using Riverpod [15]. Widgets that depend on the stream would listen for changes to the state, which contained the list of currently available Service Elements an re-render. Since stream elements arrive asynchronously, components of our app listening to changes were required to carry the async keyword. This resulted in the propagation of async functions throughout the code base, which in turn made it more difficult to implement simple functions. Errors occuring in Futures don't produce a stack trace, making debugging highly asycnrhonous code much more difficult. Furthermore, Riverpod as a library lacks maturity, as asynchronous functions arent supported, which are used by the vidoe player. Due to the two above mentioned difficulties, we moved away from using streams and simply loaded the entire Service List into memory at once. This greatly simplified state management.

B. Chewie Library

To speed up development, the Content Guide Page and Video Player page were developed separately. On its own, browsing channels on the Video Player page works without triggering bugs that crash the app. However, when incorporating the Content Guide Page, browsing channels causes the app to function incorrectly. The sound of the new channel is played, but the video player displays only a black screen. We believe this is due to either the old state not being disposed of correctly or the widgets controlling video playback still referring to objects based on old state which no longer exist. No errors are thrown regarding improper disposal of state or objects using no longer valid state, making debugging more difficult. Furthermore, we had to keep track of when to drop the video player widget in order to not run out of memory, which becomes more complex when the video player object can be accessed concurrently due to asynchronous functions. This resulted in having to manually deallocate memory in an otherwise garbage collected language, meaning we were working against the intended design of Dart.

XVI. Evaluation

In this section, we present a detailed analysis of the results obtained from our implementation, which was developed to meet specific requirements.

One of the primary objectives of our implementation was to handle the required REST APIs and XML parsing from the DVB-I concept, including communication with the ServiceList, MPD, and content guide servers. To achieve this, we developed a DVB-I Android Client Library, which is described in section XIV. Our parser library is capable of handling all requests from endpoints, and the code can adapt flexibly to different XML responses.

Another important objective of our implementation was to develop a GUI using Flutter, as described in section XIV. This GUI can be compiled as a native Android application that utilizes the DVB-I Android Client Library to present TV Services. Our GUI includes several key features, such as the ability to play various channels and programs and switch between them seamlessly, as detailed in section XIV. Additionally, we needed to display the meta-data of programs from different services in a user-friendly manner.

We evaluated the functionalities by using the Android emulator in Android Studio. We used emulators of the Google Pixel and Android TV. Our implementation has only been tested on Android with Exoplayer. As mentioned in section XII-A2, we optimized scalability by utilizing a hash map, enabling users to view the results of the service list requests immediately.

We encountered multiple NULL pointer bugs in the video_player and chewie library, which we had to fix. We also found that documentation was absent when we derailed slightly from the few examples provided. Additionally, we discovered that state management solutions like Riverpod or the provider package, which have been recommended in the official Flutter documentation, are feature incomplete and very error-prone, making them challenging to debug. Furthermore, Darts Async feature is a big contributing factor for NULL pointer bugs in general, and it makes it difficult to locate the issue as stack traces are not available if an error occurs in a Future.

In total Flutter is a promising technology on paper but due to immature libraries, bad documentation and questionable language design decisions we believe that Flutter is not yet ready to be used for production.

XVII. Conclusion and Future Work

In summary, we developed an application consisting of two major components, a DVB-I parser and a GUI, which together are capable of accesing TV programs over the internet and allowing users to interact with the application similar to terrestrial TV. We used Dart and Flutter due its cross-platform capabilities and speed. The DVB-I parser library efficiently retrieves data from Service List Registry servers and can be easily incorporated into other projects. The GUI presents users with a clean interface over which to interact with the information provided by the parser. Our project demonstrates the feasibility of using Dart and Flutter to develop an Andoird application for presenting TV services. However, the many challenges encountered while developing the GUI shows the importance of mature libraries and good documentation, thus leading to our reccomendation to not use Flutter for similar projects, until libraries mature further.

Although some of the libraries in the Flutter ecosystem require more maturation, the flexibility and extensibility provided by the Flutter frameworks and libraries therein allow a broad range of future work to be explored.

Currently, the video player page displays the option to turn on closed captioning. Some Service Elements also deliver subtitles, allowing for easy integration, greater accessibility, and usability to the hearing impaired.

Furthermore, the video player page as well as the content guide page display only a small amount of the information available in Service Elements. This could be extended to include a more detailed description of the currently running program, both in the Content Guide Page and the Video Player page. The Content Guide Page can also be updated to display channel timetables, allowing users to see what programs will be running in the future.

Currently, we have deployed our application only on Android platforms. However, supporting iOS and web applications is also possible. Lastly, moving back to using streams instead of loading entire Service Lists upfront is worth exploring, since potentially many service lists might no longer fit into memory.

References

[1] Flutter, “Flutter architectural overview“, [Online] https://docs.flutter.dev/resources/architectural-overview

[2] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[3] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[4] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[5] K. Elissa, “Title of paper if known,” unpublished.

[6] R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.

[7] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[8] M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

[9] ISO 8601-1: "Date and time – Representations for information interchange – Part 1: Basic rules". 2019

[10] Skia 2D, “Skia 2D Rendering Library“, [Online] https://skia.org/

[11] Chewie Library, [Online] https://pub.dev/packages/chewie

[12] Video Player Library, [Online] https://pub.dev/packages/video_player

[13] Digital Video Broadcasting Project, [Online] https://dvb.org/

[14] Digital Video Broadcasting-Internet: “a standards-based solution for delivering television – live, linear and on-demand – in the internet age“ https://dvb-i.tv/

[15] A Reactive Caching and Data-binding Framework https://riverpod.dev/

[16] 'Start thinking declaratively' https://docs.flutter.dev/development/data-and-backend/state-mgmt/declarative