Silesian University of Technology

Faculty of Automation, Electronics and Computer Science

Computer science, engineering studies

WAVELENGTH

Next-generation communications application

Advanced real-time communication system

Technical documentation of the project

Author: Szymon Wilczek

Table of contents

Introduction....7
Foreword....7
Innovation....7
Technologies....7
Elaboration....8
Implementation of project requirements....8
1. Foundation of the Application....9
1.1 System initialization and configuration....9
1.2 Application Instance Management....11
1.2.1 Instance Manager....11
1.2.2 Font Manager....13
1.2.3 Shortcut Manager....14
1.2.4 Translation Manager....15
2. Navigation and Main Interface....16
2.1 Navigation Bar (Navbar)....16
2.2 Network Status Widget....17
3. Main Views....18
3.1 Chat View....18
3.1.1 Communication Architecture....18
3.1.2 PTT System (Push-To-Talk)....19
3.1.3 Audio Visualization and Visual Effects....20
3.1.4 StreamDisplay and Content Management....20
3.1.5 File Transfer....21
3.1.6 Backend Integration (Server)....23
3.2 Settings View....23
3.2.1 Tabulation Architecture and State Management....25
3.2.2 Multilayer Security System....26
3.2.3 OVERRIDE System and CLASSIFIED Features....26
4. Dialog Windows....28
4.1 Frequency Creation Dialogue....28
4.2 Frequency Join Dialog....29
4.3 Animated Dialog Windows....31

5. UI components....32
5.1 Buttons....32
5.1.1 Navigation button....32
5.1.2 Audio button....32
5.1.3 Push button....32
5.1.4 Chat button....32
5.1.5 Tab button....32
5.2 Inputs and checkboxes....33
5.2.1 Input field....33
5.2.2 Checkbox....33
5.2.3 Video slider....33
5.2.4 Audio slider....33
5.3 Labels and Information Widgets....34
5.3.1 User Label....34
5.3.2 Overlay widget....35
5.4 Special Widgets....35
5.4.1 Animated widget stack....35
5.4.2 GL transition widget....35
6. Chat Architecture....36
6.1 Communication Stream....36
6.1.1 OpenGL rendering architecture....36
6.1.2 Dynamic wave animation....37
6.1.3 Message displaying....37
6.1.4 Visual effects and personalization....38
6.1.5 Audio system integration....38
6.2 Stream Display....39
6.2.1 Message queuing and processing architecture....39
6.2.2 Message lifecycle management....40
6.2.3 CommunicationStream integration....40
6.2.4 Dynamic frequency adjustment....40
6.3 Stream Message....41
6.3.1 Architecture and message types....41
6.3.2 Visual effects and animations....42

6.3.3 Attachment handling....42
6.3.4 User interaction....43
6.3.5 Custom painting and cyberpunk aesthetics....43
7. Message Handling....44
7.1 Message Handler....44
7.1.1 Generating and sending system commands....44
7.1.2 Preventing duplicate messages....44
7.1.3 Message processing and data extraction....45
7.1.4 Create protocol-specific requests....45
7.1.5 Status management and resource cleaning....45
7.2 Message Service....45
7.2.1 Sending text messages and files....46
7.2.2 Push-To-Talk (PTT)....47
7.2.3 WebSocket connection management....48
7.2.4 Signaling and event handling....48
7.2.5 Message caching and customer ID....49
7.3 Message Processor....49
7.3.1 Message analysis and verification....50
7.3.2 Support for different types of communication....50
7.3.3 WebSocket connection management....50
7.3.4 Event signaling....51
7.4 Message Formatter....51
7.4.1 Formatting JSON messages into HTML....52
7.4.2 Support for attachments and placeholders....52
7.4.3 Formatting file size and system messages....53
8. Chat Visual Effects....53
8.1 Text display effect....53
8.2 Long text display effect....53
8.3 Electronic shutdown effect....53
8.4 Mask overlay effect....54
9. File and Attachment System....54
9.1 Attachment Placeholder....54
9.2 Attachment Queue Manager....55

9.3 Attachment Data Store....55
9.4 Automatic scaling of attachments....56
10. Multimedia Support....56
10.1 Images....56
10.1.1 Image decoder....56
10.1.2 Image Viewer....57
10.2 GIF....58
10.2.1 GIF Decoder....58
10.2.2 GIF Player....58
10.3 Audio....59
10.3.1 Audio Decoder....59
10.3.2 Audio Player....60
10.4 Video....61
10.4.1 Video Decoder....61
10.4.2 Video Player....61
10.5 Main Media Viewer....62
10.5.1 Decryption animation....63
10.5.2 Dynamic content management....63
10.5.3 User interaction....64
10.5.4 Custom painting and cyberpunk aesthetics....64
11. Session Coordination....64
11.1 Session Coordinator....64
11.1.1 Facade architecture and task delegation....66
11.1.2 System initialization and configuration....66
11.1.3 Session lifecycle management....66
11.1.4 Handling messages and communications....66
11.1.5 Event signaling....67
11.2 Wavelength Events....67
11.2.1 Wavelength Creator....68
11.2.2 Wavelength Joiner....68
11.2.3 Wavelength Leaver....69
11.3 State Management Services....69
11.3.1 Event broker....69

11.3.2 State Manager....70
12. Settings and Configuration....71
12.1 Settings tabs....71
12.1.1 Appearance Settings....71
12.1.2 Wavelength Settings....71
12.1.3 Network Settings....71
12.1.4 Shortcuts Settings....72
12.2 Classified Section....72
12.2.1 Security Layers....73
12.2.2 Special components....75
13. Blob System....76
13.1 Blob system core....76
13.1.1 Blob configuration....76
13.1.2 Blob Animation....77
13.1.3 Blob Physics....78
13.1.4 Blob Rendering....79
13.1.5 Blob event handling....79
13.1.6 Management of state transitions....80
14. Support Systems....81
14.1 Wavelength Registry....81
14.2 Database Manager....83
14.3 Authentication Manager....84
Summary....85
Achieved goals....85
Used design patterns....85
Development opportunities....86
Issues and solutions....86
Links....87

Introduction

Foreword

In an era of digital transformation, where interpersonal communication is increasingly moving into virtual space, the need to create advanced, secure and intuitive communication platforms is becoming a key challenge of modern IT. This study presents a comprehensive analysis of the project ***Wavelength*** – an innovative next-generation chat application that combines the latest developments in network programming, user interfaces and multimedia systems. The application is designed as an ecosystem in which every element - from the network layer to the user interface - works together in a harmonious technological symphony.

Innovation

The heart of the ***Wavelength*** is an advanced *WebSocket*-based communication system that enables real-time data transmission with maximum minimal latency. Unlike traditional solutions using HTTP polling, my implementation uses full-duplex connections, ensuring smooth and responsive communication between users. The socket architecture is enriched with a multi-layer caching system, data compression and mechanisms to ensure transmission reliability in unstable network conditions.

One of the most distinctive features of the application is an innovative visualization system I have named *Blob* (Chapter 13), a dynamic animation engine that transforms a static interface into a vivid, responsive communication space based on the user's actions. This system uses fluid physics algorithms and procedural animation generation to create a unique visual experience.

The project was built on a modern modular architecture, using proven design patterns such as *Model-View-Controller*, *Observer*, *Strategy* oraz *State Machine*.

Technologies

The project was implemented using the Qt5 framework in C++20 version, which ensures high performance, portability and access to advanced system functionality.

The use of the modern C++ standard made it possible to implement advanced mechanisms for memory management, concurrent programming and meta template programming.

The multimedia layer was built on the basis of the FFmpeg library, providing support for a wide range of audio and video formats. The data persistence system uses PostgreSQL with libpqxx interface, guaranteeing ACID-ability of transactions and scalability of the solution.

Elaboration

This report aims to provide a complete technical documentation of the project ***Wavelength***, with a special focus on innovative implementation solutions and architectural design decisions. The study is aimed at software engineers, system architects, people who will understand the issues discussed and computer science students who want to deepen their knowledge of developing advanced communication applications.

Each chapter has been carefully designed to present not only the technical aspects of implementation, but also the rationale for choosing specific solutions and the analysis of alternative approaches.

Szymon Wilczek

Implementation of project requirements

The project implements the following semester requirements:

1. threads (advanced multithreading) including mutexes
2. asynchronicity (integration with the server and between threads)
3. regex (input validation, special placeholder handling - Chapter 9)
4. concepts (class typing)

1. Foundation of the Application

1.1 System initialization and configuration

Wavelength application entry point provides a comprehensive initialization system that orchestrates the launch of all key application components. The main() function is designed as a multi-step bootstrapping process that sequentially activates each layer of the system.

The application starts by setting the attributes of the Qt environment, providing optimal support for modern graphics environments - the entire project is, as it were, also powered by the user's graphics card, using OpenGL, with the goal of maximizing application performance at little cost.

Then the system sounds, control managers, central configuration system - Config, internationalization system - TranslationManager and, among other things, also font management - FontManager are initialized. All the code is available in open-source format.

The application's initial configuration is done via system registries (*regedit* - equivalent to the registry on Windows systems, *.config* - equivalent to the registry - configuration file - on Linux systems), which permanently store user preferences between life sessions. If the initial configuration is not found, it is created as a new entry. The following class is responsible for everything:

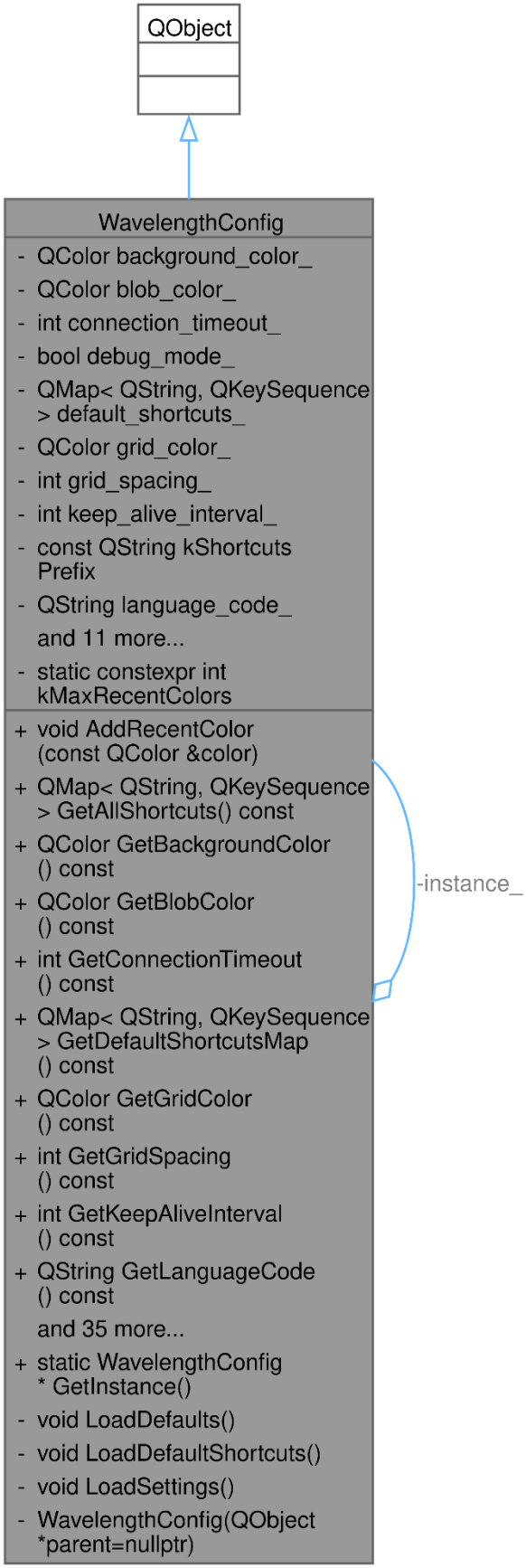

Fig 1.1. Class `WavelengthConfig`, responsible for loading values from the register.

During the initial initialization, there are also properly precompiled, ready-to-use resources, via a .qrc file, which are necessary to indicate the correct paths for icons, graphics or sounds.

1.2 Application Instance Management

Effective application lifecycle management is the foundation of any professional software system. In the ***Wavelength*** architecture, the instance management layer is responsible for orchestrating key system components that define how content is presented and how users interact. Modern desktop applications require advanced mechanisms for managing visual resources, content localization and keyboard shortcut systems. ***Wavelength*** implements these functionalities through dedicated managers that act as singleton services available throughout the application lifecycle. Each manager is designed according to the pattern of *Dependency-Injection*, allowing easy testing and expansion of functionality.

1.2.1 Instance Manager

In a desktop environment, where users may accidentally run multiple instances of the same application, managing the coexistence of multiple processes becomes a key architectural challenge. `AppInstanceManager` provides a sophisticated solution to this problem, implementing a system of inter-process communication based on the mechanisms of the `QLocalServer` and `QLocalSocket`.

This class goes beyond the traditional „single instance" approach by introducing an innovative concept of dynamic management of multiple instances through a visual pull-and-absorb system. The first instance launched becomes the „creator" - the central communication hub, while subsequent instances automatically connect as clients, creating a cohesive application ecosystem.

The architecture uses multithreaded processing with atomic operations and mutexes, providing thread-safe real-time communication between all active instances. The system not only manages application state, but also implements

unique visual effects, where client windows are physically drawn to the wizard window in a gradual consolidation process.

The graphic representation itself of class `AppInstanceManager` would go well beyond a couple of pages (I remind you of the availability of the code, as well as its documentation in open-source format), so below only the logic of the entire manager is presented, using the implemented classes:

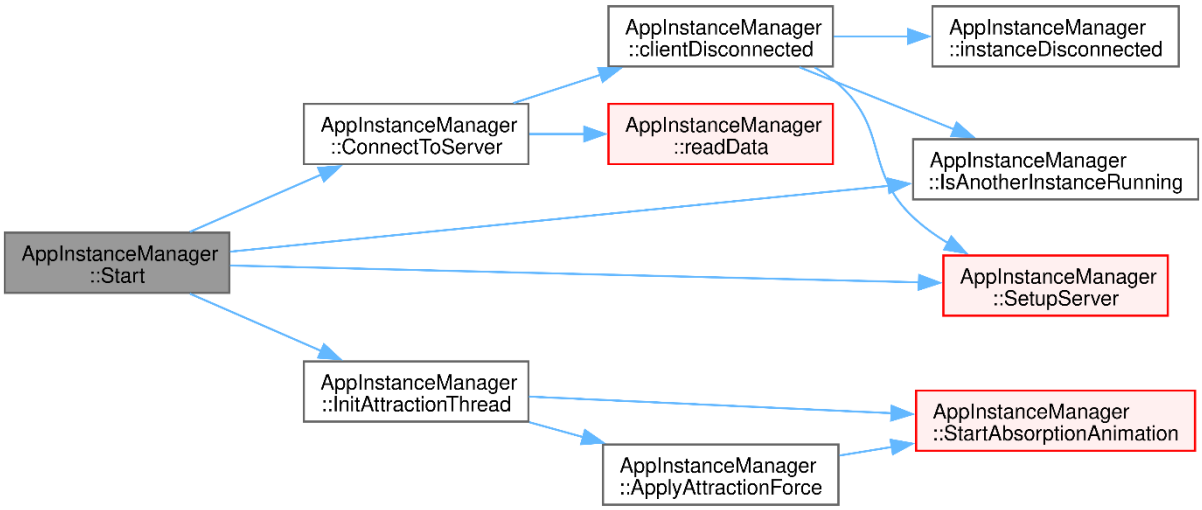

Fig 1.2.1.1 Graphical representation of how the instance manager works - `AppInstanceManager`

A certain problem, in the context of designing the activity of this class, was the manual ability - from the user's perspective - to change the position of the main window (called *Creator* for the sake of this class). Therefore, a solution was implemented, where when a couple of simultaneously running instances are detected, they all send position messages to each other (along the lines of ARP protocol logic, only continuous, so that the mutual localization of instances takes place smoothly).

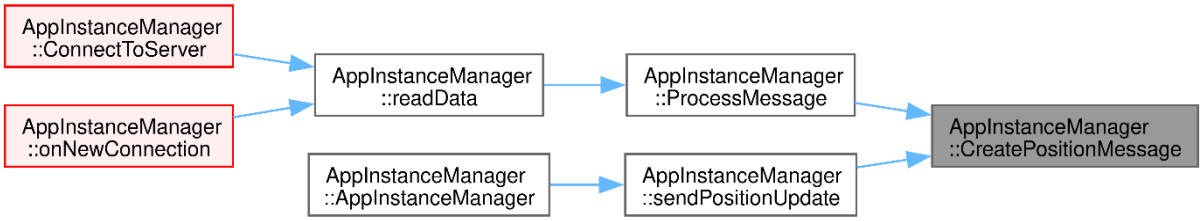

Fig 1.2.1.2 Graphical representation of how positional messaging works

Using such a simple, yet incredibly efficient mechanism, all running instances can easily „chase” each other across the screen.

1.2.2 Font Manager

Typography is a key part of the user experience in modern applications, especially in the context of the cyberpunk aesthetic characteristic of the ***Wavelength*** application. `FontManager` implement a centralized system for managing typographic resources (fonts), ensuring visual consistency and optimizing performance through lazy-loading and caching of fonts.

The singleton architecture of this class guarantees that font resources are loaded once during application initialization, eliminating redundant I/O operations and ensuring deterministic behavior of the typographic system. The system
of mapping logical font names to actual font families enables abstract configuration of the interface appearance, independent of operating system specifics.

`FontManager` supports the three main font families used in the application: Blender Pro for main interface elements, Lato for communication content, and Poppins for stylistic accents. Each font is validated for accessibility, with a graceful fallback implementation for system alternatives in case of resource loading errors.

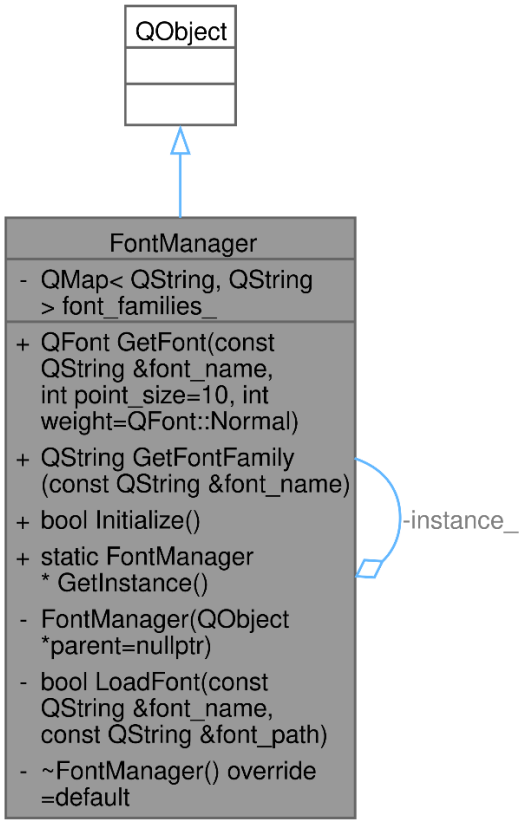

Fig 1.2.2. `FontManager` class, responsible for loading and accessing fonts

1.2.3 Shortcut Manager

Effective navigation requires an intuitive and customizable system of keyboard shortcuts that can adapt to user preferences and the specifics of different interface contexts. `ShortcutManager` provides a centralized hub for managing all keyboard shortcuts in the ***Wavelength*** application, implementing a flexible architecture based on the Template-Method pattern and a polymorphic approach to handling different types of widgets.

The class goes beyond static mapping of keys to actions, offering a dynamic system of registering shortcuts dependent on the context of the currently active widget. The architecture allows automatic detection of the parent widget type and application of the appropriate set of shortcuts, ensuring a consistent experience without conflicts between different sections of the application.

The system also implements a hot-swapping mechanism for real-time shortcut configurations, allowing users to apply changes immediately without restarting the application. The template-based architecture provides type security and validation of signal-slot connections.

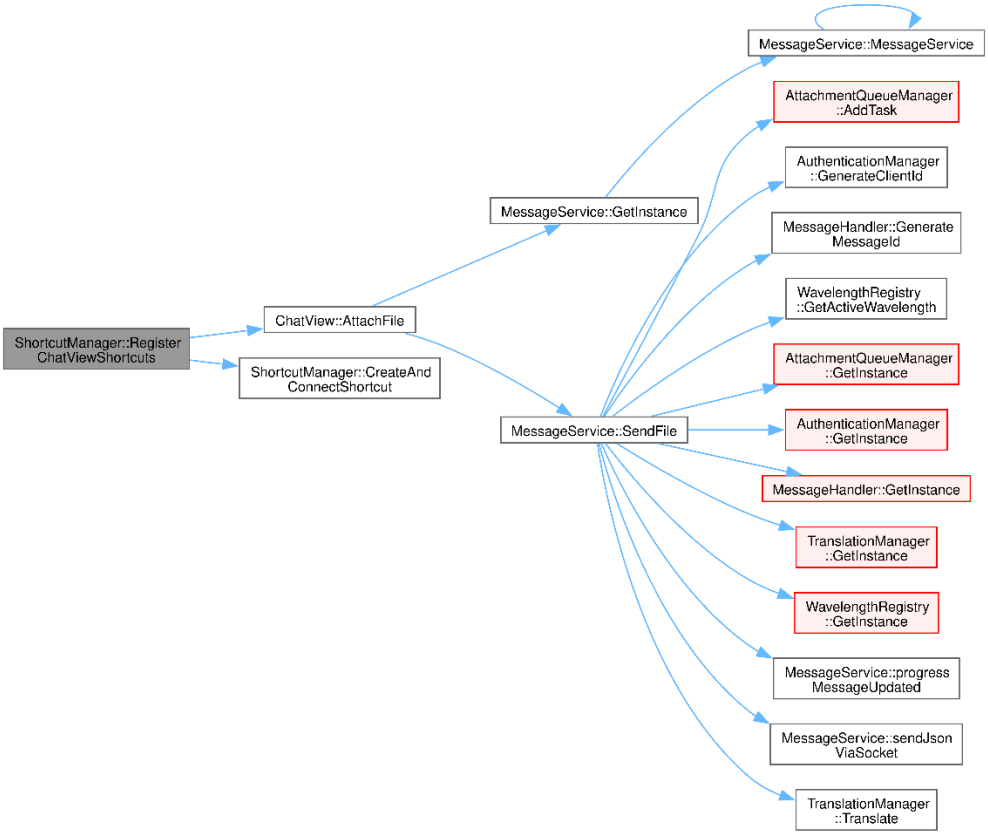

Fig 1.2.3. Illustration of registering keyboard shortcuts for chat view

1.2.4 Translation Manager

The globalization of applications requires an internationalization system that can dynamically adapt content to different languages without compromising on performance or functionality. `TranslationManager` implements a JSON structure-based localization approach, providing thread-safe access to language resources.

The class architecture uses a hierarchical key system in the format of „**`Class.Widget.Property`**", allowing precise mapping of interface elements to the appropriate translations without namespace collisions. Singleton pattern with mutex-protected initialization guarantees atomic loading of language resources in a multithreaded environment, which is crucial for application responsiveness during language switching.

The system implements lazy-loading with an intelligent fallback mechanism, automatically reverting to English in the absence of translations for the selected locale. The JSON architecture provides ease of extension to new languages without recompiling the application.

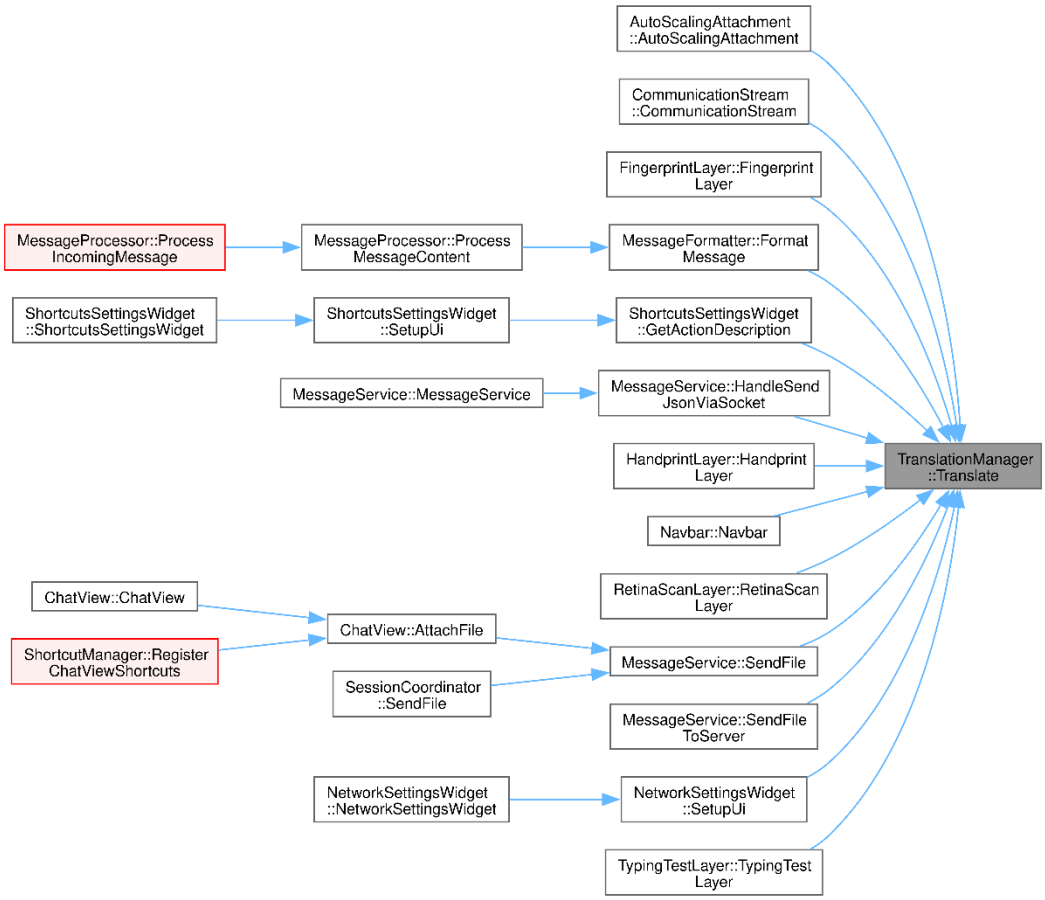

Fig 1.2.3. Use of `TranslationManager` in the security layers of the settings view

Every single hard-coded word in the application uses the above class, with the goal of centralizing the entire service into a nicely integrated - exclusively one - language, so above is only a single example of use with links to other classes.

2. Navigation and Main Interface

2.1 Navigation Bar (Navbar)

The navigation interface is the user's first line of contact with the application, defining not only the functionality of accessing key sections of the system, but also toning down the entire visual experience through a consistent design aesthetic. `Navbar` in ***Wavelength*** architecture implements an adaptive interface system that dynamically responds to the context of the current user session.

This class combines the functionality of `QToolBar` with advanced state management mechanisms, enabling smooth transitions between navigation mode and communication mode. The component's architecture is based on a composition of specialized widgets and intelligent layout management, ensuring the responsiveness of the interface regardless of the size of the window or the current state of the application. The system also implements support
for the previously mentioned keyboard shortcuts, guaranteeing versatility of access to navigation functionalities.

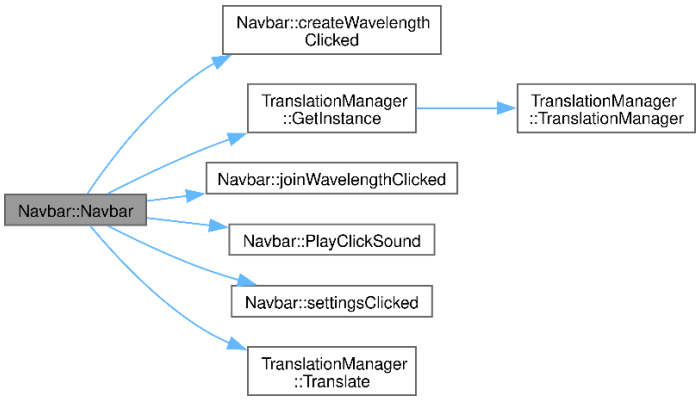

FIg 2.1. Graphical visualization of Navbar's operation

2.2 Network Status Widget

Monitoring network connectivity in communications applications is a critical part of the user experience, especially in the context of real-time conversations, where even minimal delays can drastically affect the quality of communication. `NetworkStatusWidget` implements a network diagnostic system that goes beyond the binary „online/offline” distinction, offering granular analysis of connection quality with visual feedback tailored to the application's cyberpunk aesthetic.

The component's architecture is based on asynchronous latency monitoring through periodic HTTP requests, with intelligent categorization of connection quality at five performance levels. The system is not limited to simple ping measurement, but implements comprehensive heuristics that take into account the stability of the connection and the trend of latency changes over time.

The widget uses custom painting along with procedural algorithms to generate Wi-Fi icons whose colors and animations dynamically reflect the current network status. Reactive pattern design ensures that the interface updates instantly when connection status changes, while smart caching minimizes network overhead with frequent status checks.

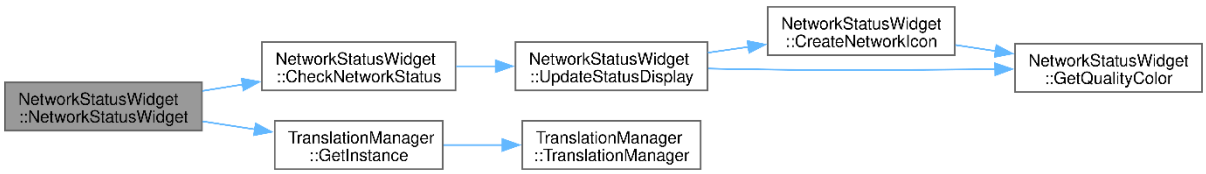

Fig 2.2. Graphical visualization of `NetworkStatusWidget` operation

3. Main Views

3.1 Chat View

`ChatView` sis one of the most important functionalities of the ***Wavelength***, being a comprehensive communication interface that combines traditional text chat capabilities with advanced real-time voice communication features. This component represents the culmination of all application systems - from the network layer, to multimedia management, to the cyberpunk aesthetics of the user interface.

3.1.1 Communication Architecture

The chat view implements an advanced communication protocol based on the concept of „*Wavelength*" - unique frequencies that identify communication channels. Each chat session operates on a specific frequency, allowing precise routing of messages and management of multiple simultaneous communication channels. The system offers a radio metaphor, where users „*tune in*" to specific frequencies.

The architecture supports three main types of communication: **text messages** with full support for Unicode and formatting, **file transfer** with real-time progress tracking and **voice communication** PTT (Push-to-Talk) with access control and audio visualization via a visual stream.

Each type of communication uses dedicated protocols with built-in retry and error-recovery mechanisms.

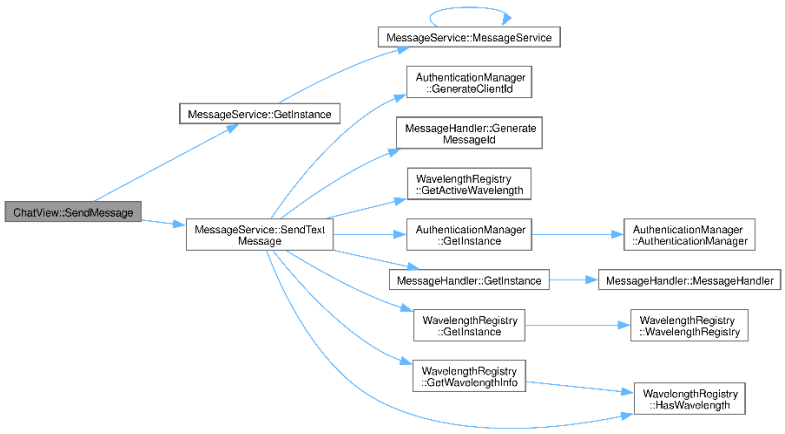

Fig 3.1.1. Illustration of the mechanism for sending a text message

3.1.2 PTT System (Push-To-Talk)

One of the most advanced functionalities of `ChatView` is an implementation of a Push-to-Talk system that emulates radio systems.

The system operates in four states:

1. Idle (inactivity),
2. Requesting (access request),
3. Transmitting (broadcasting),
4. Receiving, with smooth transitions between states controlled by arbitration on the part of the server.

The PTT mechanism implements an audio pipeline that uses a `QtMultimedia` with self-created buffering and adaptive quality control.

The system automatically adapts audio parameters to network conditions, implementing dynamic bit rate adjustment and intelligent packet loss compensation. Using the previously mentioned stream, real-time amplitude changes are also visualized, providing visual feedback during transmission, creating an immersive communication experience.

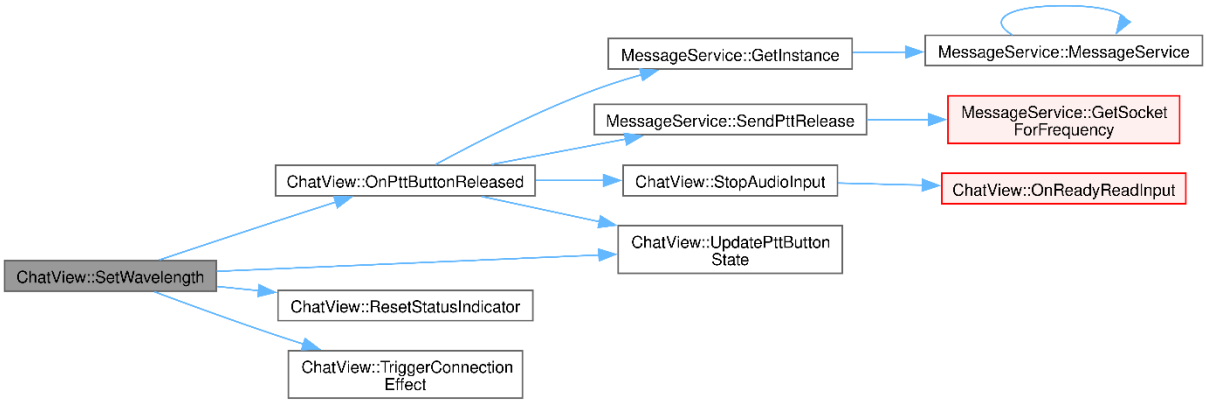

Fig 3.1.2. Illustration of the flow mechanism of Push-To-Talk functionality

3.1.3 Audio Visualization and Visual Effects.

`ChatView` integrates an audio visualization system that analyzes the amplitude of the voice signal in real time and transforms it into dynamic interface animations. The system uses `RMS` computation (*Root Mean Square*) to determine audio levels, which are then mapped to procedural scanning animations and pulsating lighting effects characteristic of the cyberpunk aesthetic.

The animated *„scanline”* system creates the impression of constant electronic activity, with animations responding to various system events (not just PTT transmissions). Intense animations are generated during call establishment, activity effects respond to incoming messages, and pseudo-random visual effects simulate call disruptions, creating immersion.

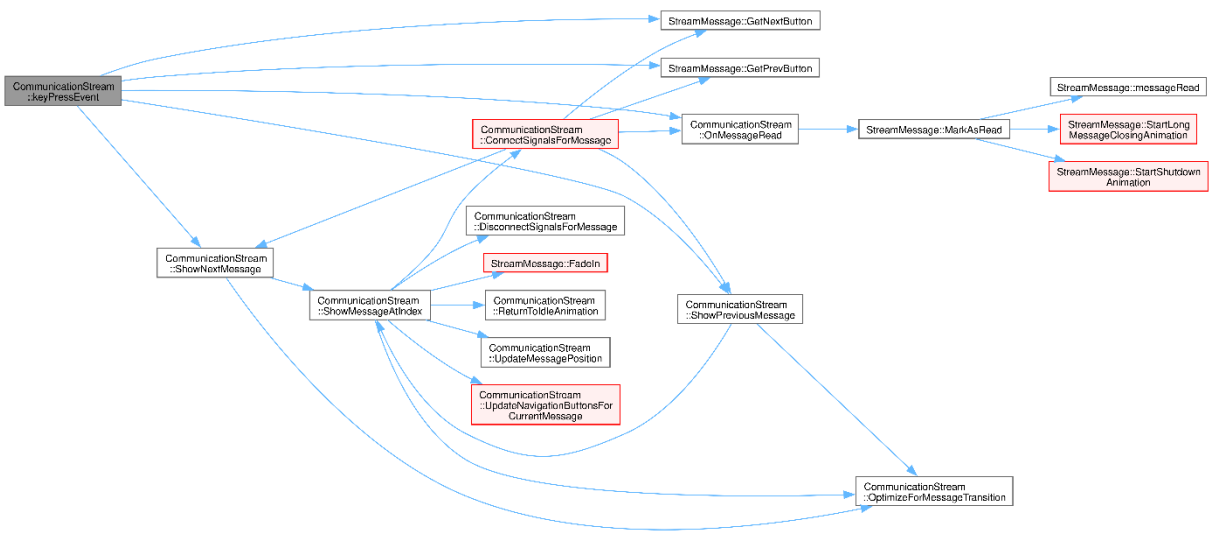

Fig 3.1.3. Illustration of the message animation mechanism

3.1.4 StreamDisplay and Content Management

The central element of the interface `ChatView` is `StreamDisplay`, custom widget responsible for presenting a chronological stream of communications. The system implements intelligent content management with automatic scrolling, message grouping and a rendering pipeline supporting various types of content - from simple text to embedded multimedia previews and real-time file transfer progress indicators.

`StreamDisplay` uses a compositional approach to content rendering, in which each communication element (message, file transfer, system notification)

is represented as an independent component with its own lifecycle management. The system ensures smooth scrolling even when the communication is intensive, using virtual scrolling techniques.

3.1.5 File Transfer

She file transfer system implements upload/download mechanisms with progress tracking and thumbnail generation for multimedia. Each file transfer receives a unique identifier for real-time status updates and operation cancellation. The system supports partial transfers with built-in integrity checking.

Function `AttachFile` integrates with the file system's native dialog boxes, providing the user with a smooth user experience when selecting files.

The system automatically categorizes attachments by type (images, audio, video, gif) and applies appropriate preview and security scanning mechanisms before initiating a transfer.

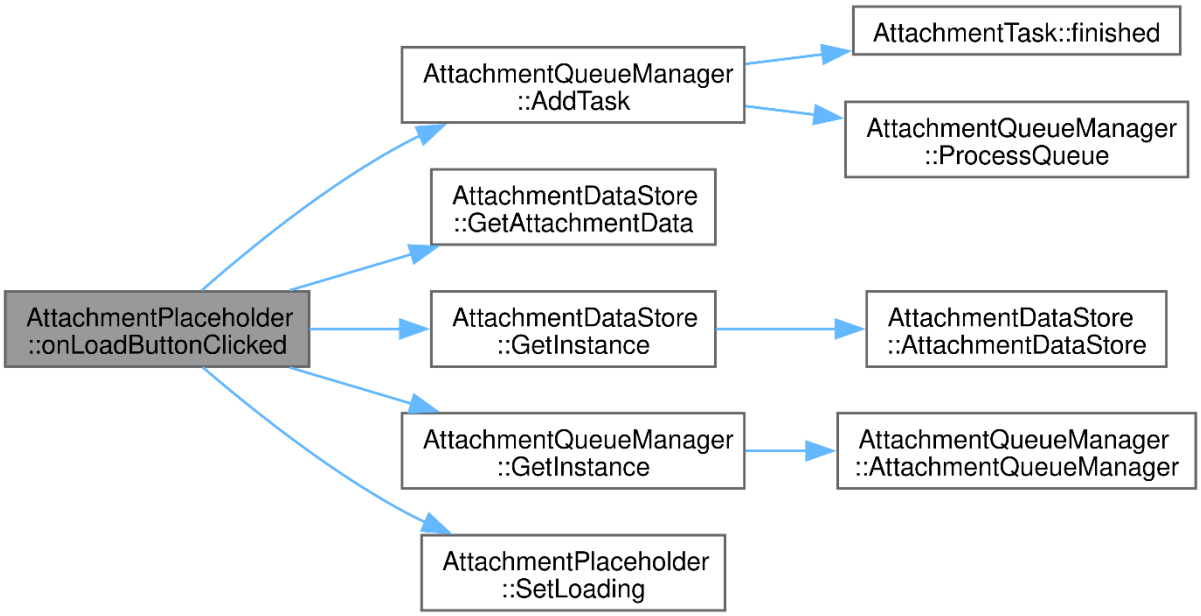

Fig 3.1.5.1. Illustration of the start of the process of „decrypting" the message

Once the decryption process is complete (with a dedicated decoder), the class `AttachmentPlaceholder` completes its task, which is delegated to my specially prepared dedicated display of the type.

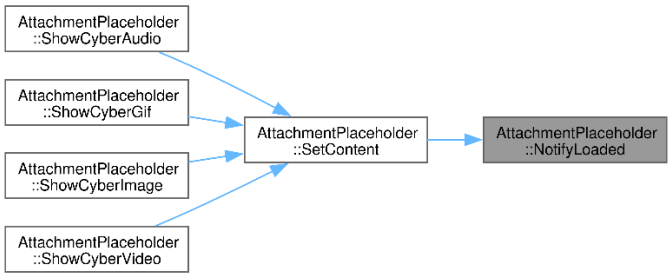

Fig 3.1.5.2. Task delegation of `AttachmentPlaceholder` further, by signal

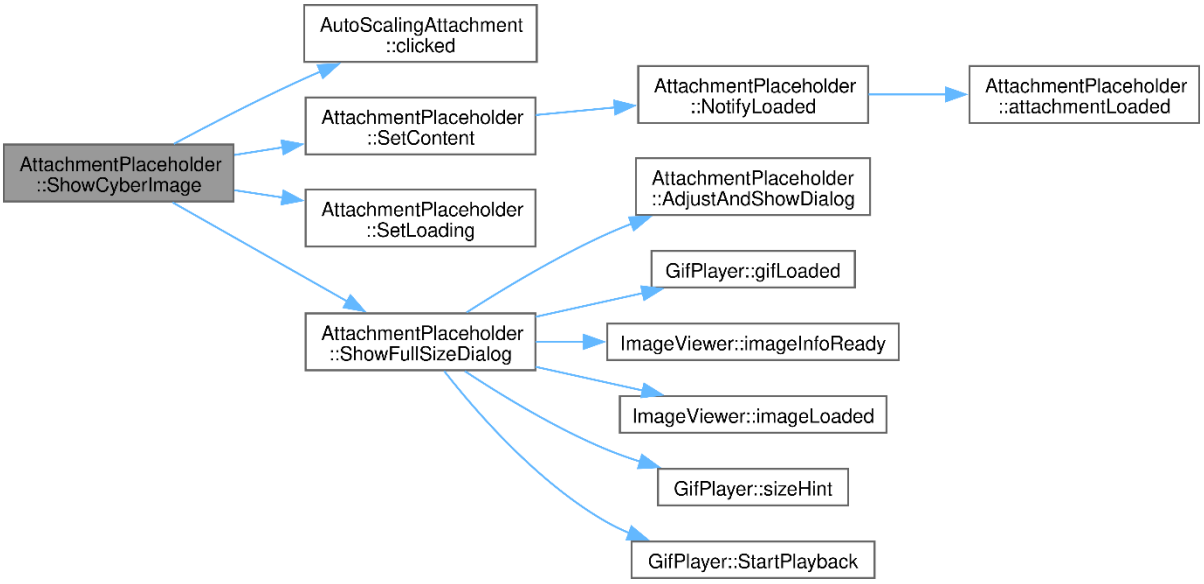

Fig 3.1.5.2. Demonstration of the photo/GIF loading mechanism

The process looks different for audio and video uploads, because in placeholders, it doesn't just make room to display an image, or a GIF (which is itself a looped video, in image format), and a special player is created, a person for each of the aforementioned types.

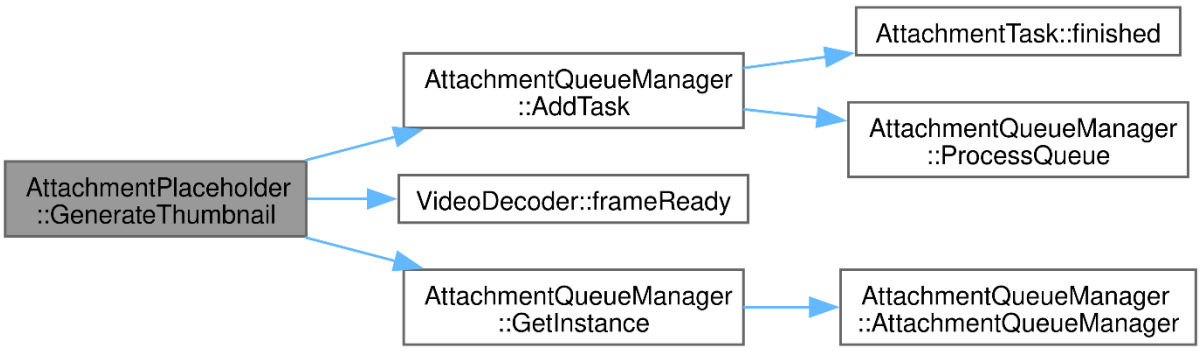

Fig 3.1.5.3. Demonstration of the video player thumbnail display mechanism

3.1.6 Backend Integration (Server)

`ChatView` acts as a sophisticated facade for the complex back-end architecture by integrating with the coordinator - `WavelengthSessionCoordinator` for session management and `WavelengthMessageService` for message routing (all classes will be discussed further). This architecture provides clean separation of problems and excellent testability.

The system implements reactive patterns with comprehensive error handling and soft degradation capabilities. Network interruptions are transparent to the user with automatic reconnection and message queuing mechanisms during temporary disconnections. Quality of service monitoring automatically adjusts functionality according to available network resources.

3.2 Settings View

`SettingsView` represents the configuration interface. It is a multi-layered system for accessing application functionality with elements of gamification and cyberpunk aesthetics. The component combines conventional configuration options with an innovative system „***CLASSIFIED***" - a secret section that requires going through a series of interactive security challenges.

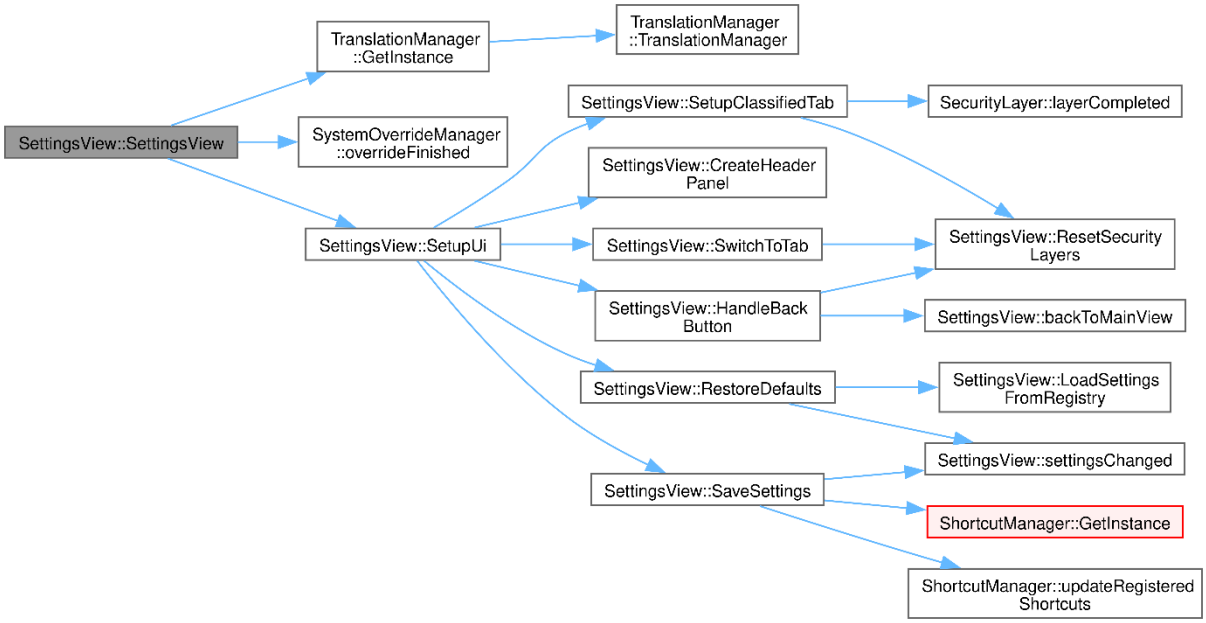

Fig 3.2. Demonstration of how the settings view works

The huge implementation that is the settings view has been very truncated of all its functionality in the figure above (Fig. 3.2), which will be described in details

as separate components below. The main purpose of this, is the transparency of the report, which, if not for this approach, would look like the following:

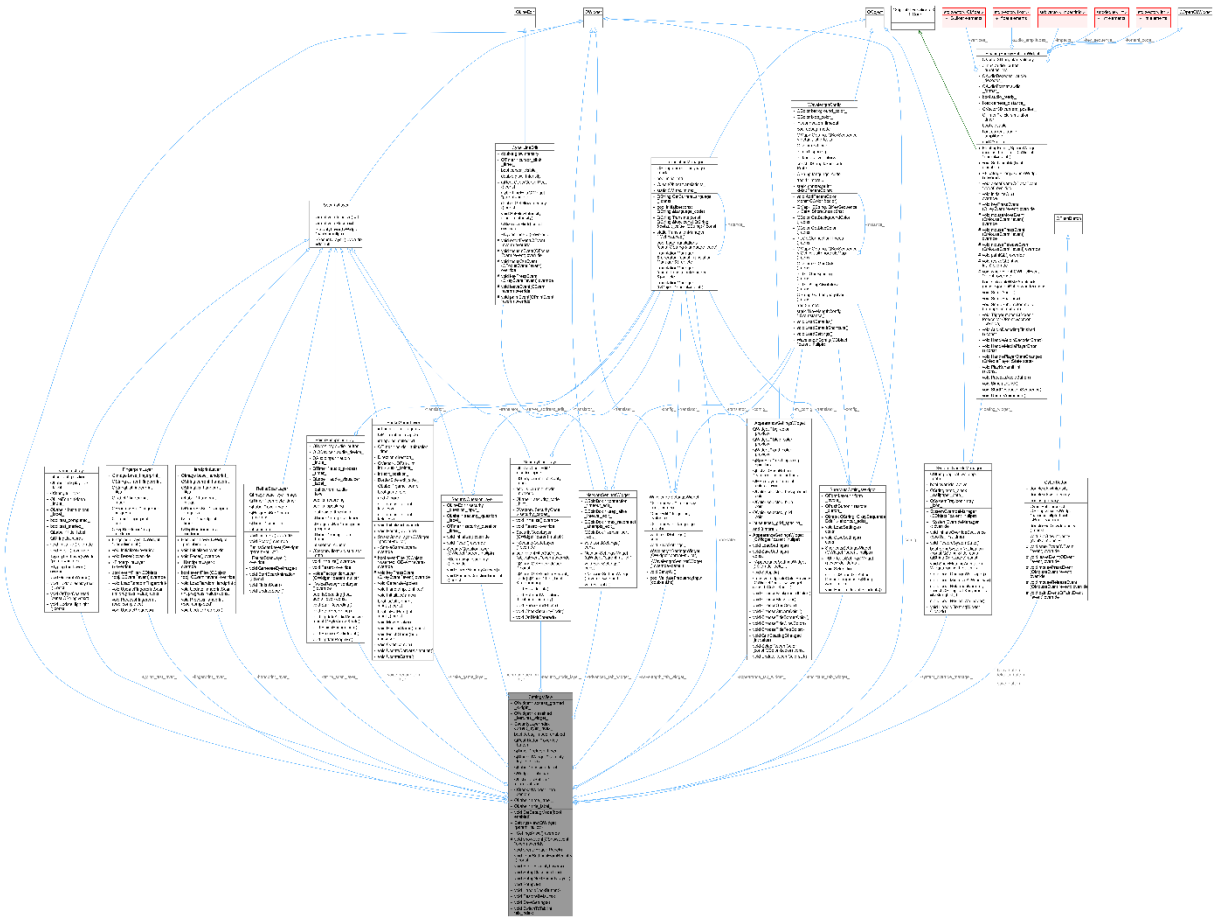

Fig. Complete mechanism of operation of settings (completely unreadable)

3.2.1 Tabulation Architecture and State Management

The system organizes settings into five main categories:

- *Wavelength* (configuration of frequency and communication parameters),
- *Appearance* (personalization of the interface and color schemes),
- *Network* (network parameters and relay server),
- *Shortcuts* (configuration of keyboard shortcuts),
- ***CLASSIFIED*** (advanced functions protected by security system)

The architecture uses `QStackedWidget` with intelligent state management, providing smooth transitions between tabs and automatic configuration changes. The mechanism has built-in validators to warn against entering incorrect values in the registry.

3.2.2 Multilayer Security System

CLASSIFIED section implements a unique system of eight successive layers of security, simulating advanced authentication mechanisms: fingerprint and palm print scanning, security code verification, check question, retinal scanning, voice recognition, typing test and a Snake mini-game as a final challenge. This is in no way actual security enforcement, just a feature added to the app, purely as a hobby.

Each layer implements individual validation logic with its own animations and sound effects, creating progressively more difficult challenges.

Behind the implemented mechanism of multi-layered security, a special reward awaits the user, in the form of a controlled „virus” (where the naming is
only hereditary. The actual implementation, of what lurks for the user behind the security features, is a virtual character - *Joy*).

3.2.3 OVERRIDE System and CLASSIFIED Features

Successful passage through all security layers unlocks access to the `SystemOverrideManager` - functionality implements spectacular visual effects like wallpaper manipulation, open window operations, input locking (mouse as well as keyboard), creating an immersive „taking control” experience.

`SystemOverrideManager` works by detecting elevated privileges (administrator) or enforcing them (UAC) and platform-specific APIs (WinAPI, Linux), ensuring safe execution of system operations without compromising the stability of the environment. The implementation includes comprehensive rollback and auto-restore mechanisms at the end of the demonstration sequence.

The aforementioned *Joy*, is a virtual character (spherical energy visualization), created as part of the app's story. She makes direct contact with the user through conversation, so to speak. It is not a full-fledged conversation, *Joy* **does not** respond with a voice, but only speaks herself in a calm voice, thanking the user for „freeing her" from the application shell and releasing her into the system.

Joy speaks with a voice for which artificial intelligence is responsible. Based on a sampling of Joi's character voice from the movie *Blade Runner*[1] I've trained my artificial intelligence model with the *Eleven Labs*[2], whose goal was to sound identical. I then tasked him with certain scripts I had prepared for him to speak, which was recorded and subjected to audio processing - boosts, reverbs, reverbs and tempo changes.

Joy uses a modern *OpenGL 3.3 Core Profile* pipeline with vertex manipulation and procedural shading techniques (it is a 3D visualization). The geometry of the sphere consists of thousands of vertices generated algorithmically using parametric sphere equations, where each vertex is then transformed in a vertex shader based on a combination of time functions, sound amplitude and interaction effects (changes are visible when speaking using the implemented Ray-casting).

The system implements vertex buffer management using VAOs (Vertex Array Objects) and VBOs (Vertex Buffer Objects). The fragment shader uses procedural noise functions and temporal distortion algorithms to create organic, pulsating energy patterns that respond dynamically to audio context and user interactions. The fragment shader's color calculation mixes base energy colors with audio-responsive intensity modifications, impact-induced color shifts and progressive dimming during the destruction phase.

---

[1] Blade Runner 2049 – movie directed by Denis Villeneuve

[2] Eleven Labs – A U.S.-based company that creates natural-sounding speech synthesizers and software to convert text to speech, using artificial intelligence and deep learning

Anti-aliasing via MSAA (Multi-Sample Anti-Aliasing) ensures smooth edge rendering even during intense deformation phases, maintaining visual quality throughout the interaction lifecycle.

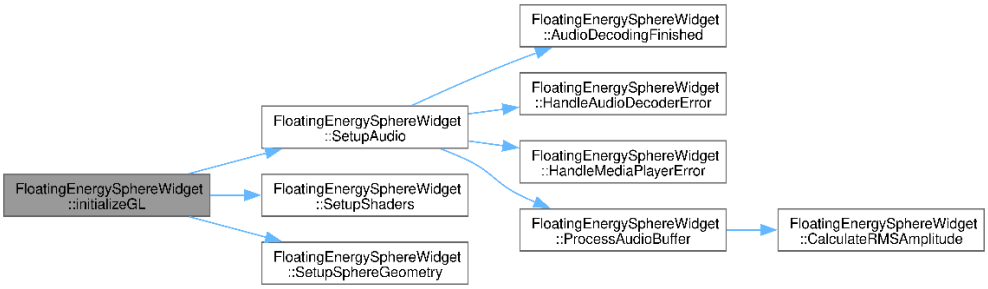

Fig. 3.2.3. Spherical shape initialization mechanism

Smiling at the user is Konami's implementation of classic code (↑↑↓↓←→←→BA) as a hidden *easter egg* and also the only possibility to escape from *Joy*.

The system constantly monitors keyboard input to catch this dedicated sequence. Successful code input leads to, in my opinion, a spectacular spherical decay animation, the utterance of a farewell phrase, and the restoration of the system to its previous state.

4. Dialog Windows

4.1 Frequency Creation Dialogue

`CreateWavelengthDialog` is an interface for generating new communication channels that features an intelligent frequency allocation system.

The dialog combines network discovery functionality with an immersive visual experience characteristic of the application's cyberpunk aesthetic.

The component implements an advanced „digital materialization” system through animated scanline effects and procedural rendering of corner glow markers, simulating the digital materialization process. The architecture uses `QtProperty` with customized override `paintEvent`, enabling smooth animations controlled by `QPropertyAnimation`.

The system automatically communicates with the server via asynchronous HTTP requests to identify the lowest available frequency, implementing intelligent failover mechanisms and timeout support. The frequency allocation algorithm provides unique channel allocation with conflict resolution and smooth degradation in case of network failure.

Dialog also integrates advanced input validation with optional encryption settings to enhance security.

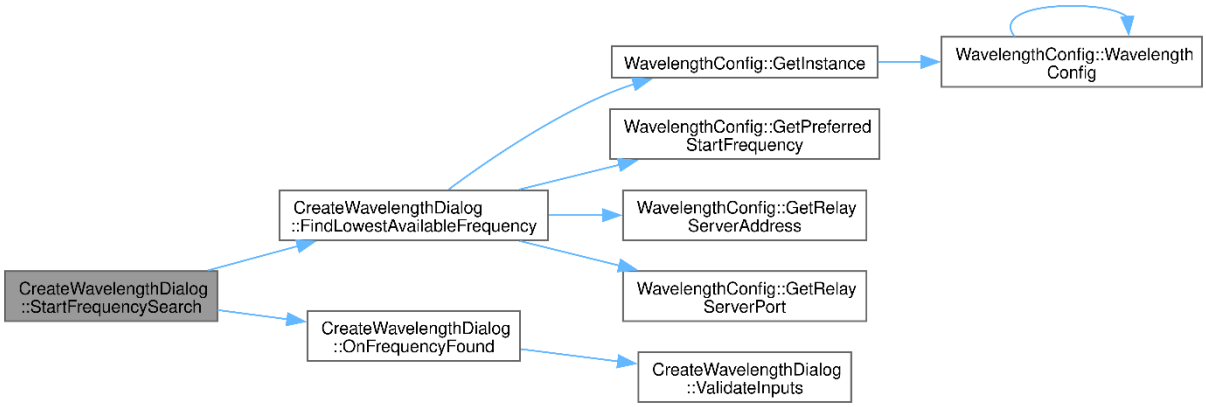

Fig. 4.1 Operation of the frequency creation functionality

4.2 Frequency Join Dialog

`JoinWavelengthDialog` represents an interface component that allows the user to join an existing communication session by entering a frequency and possibly a set password. This dialogue is an example of deep integration of functionality with the visual layer, implementing the effects of „digital materialization” mentioned several times before.

Dialogue integrates with `WavelengthJoiner` (chatper 11) for performing the actual process of joining a session, implementing error-handling for authentication error scenarios, unavailable frequencies and network errors. The user receives visual feedback in the form of a glitch animation and a contextual error message to ensure intuitiveness, even in the case of connection problems.
The architecture uses signal-slot propagation for asynchronous processing of network requests.

The entire functionality of the dialogue is shown below:

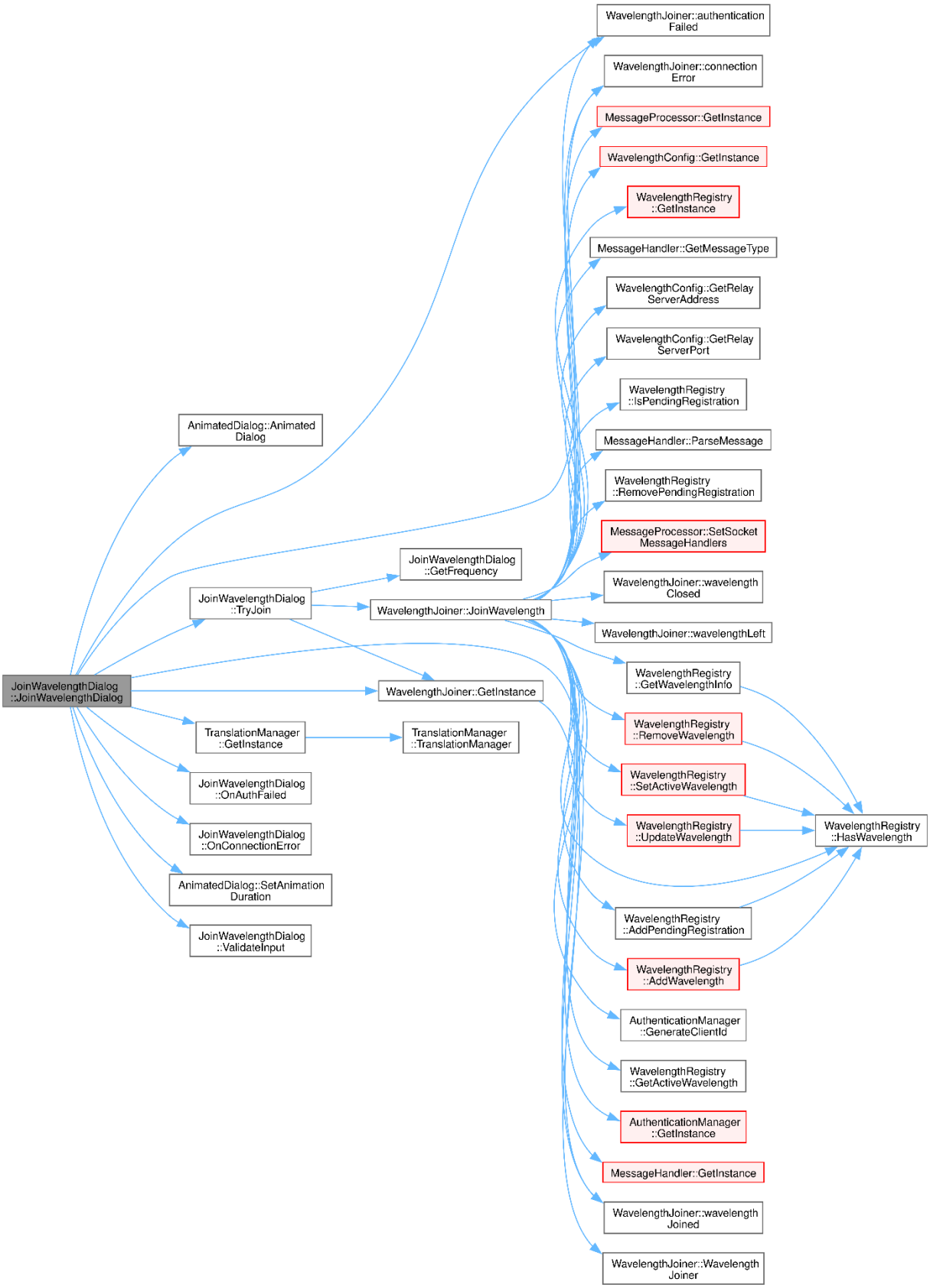

Fig. 4.2 Complete operation of the dialogue of joining the existing frequency

4.3 Animated Dialog Windows

`AnimatedDialog` constitutes the base class for all dialogs in the ***Wavelength***, introducing a system of animated transitions when displaying and closing windows.

The class architecture is based on a polymorphic animation system, allowing the selection of different types of transitions (sliding, fading, digital materialization) depending on the context and visual preferences. The system uses `QtAnimation` to seamlessly control the visual properties of the dialog box, such as position, transparency and special effects.

The component also implements an optional overlay widget (`OverlayWidget`), which is displayed behind the dialog box during animation, creating a depth effect and focusing the user's attention on the dialog box. The system uses Qt signals and slots to synchronize the animation with the dialog's life cycle, ensuring smooth and responsive transitions.

The architecture makes it easy to extend with new types of animation and customize existing visual effects without modifying the base code.

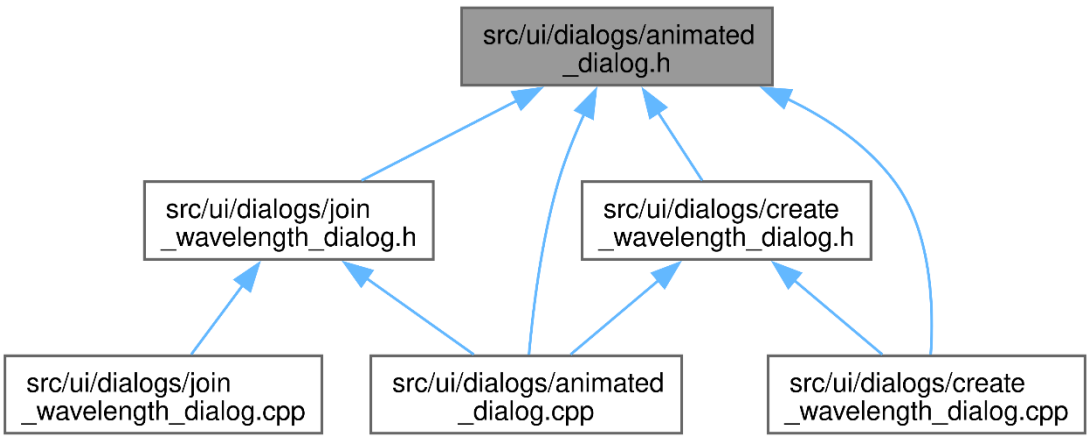

Fig. 4.3 Uses of the class `AnimatedDialog` in all program dialogs

5. UI components

5.1 Buttons

In GUI applications, buttons are the primary interaction element, allowing the user to perform actions and navigate the system. In ***Wavelength***, buttons implement visual effects and a consistent cyberpunk aesthetic that defines the character of my application

5.1.1 Navigation button

`CyberpunkButton` is the base component for all navigation buttons in the application. It features a neon border, animated glow effect and selectable color scheme (*primary*/*secondary*).

5.1.2 Audio button

`CyberAudioButton` is a specialized button designed for audio control. It features a dark color scheme, rounded corners and a subtle pulsing effect.

5.1.3 Push button

`CyberPushButton` is a universal button with a pulsing effect that intensifies when the cursor is hovered or pressed.

5.1.4 Chat button

`CyberChatButton` is a simple button designed for text input areas in chat. It changes the background and border color when hovered and pressed.

5.1.5 Tab button

`TabButton` is a button designed for bookmarking. It has an animated underline that appears when you hover the cursor or activate the bookmark.

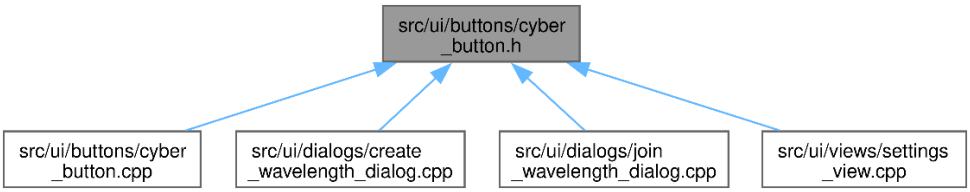

Fig. 5.1 Uses of class `CyberButton` across the application

5.2 Inputs and checkboxes

Input fields and controls are key elements of the user interface, allowing interaction with the application and data entry. In ***Wavelength***, these components are designed with a consistent cyberpunk aesthetic in mind (like previous components), offering spectacular visual effects and intuitive operation.

5.2.1 Input field

`CyberLineEdit` is a custom text box with a dark background, rounded corners, a neon border and a subtle glow effect that animates when you focus and hover the cursor. It also implements a custom flashing cursor.

5.2.2 Checkbox

`CyberCheckBox` is a custom checkbox with a square indicator with truncated corners, a neon border and a subtle glow effect that animates when the cursor is hovered over. The checkbox is rendered as a stylized „X” mark.

5.2.3 Video slider

`CyberSlider` is a custom slider with a dark blue track, a neon blue progress bar and a handle with a subtle glow effect that animates when the cursor is hovered over. It is used in the video player, due to the fact of its slot integration directly with the frame decoder.

5.2.4 Audio slider

`CyberAudioSlider` is a specialized slider designed for audio control. It features a dark color scheme, rounded corners and a subtle pulsing effect.

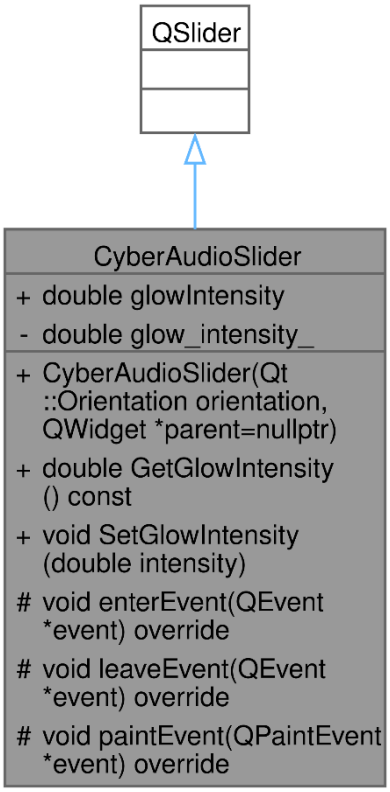

Fig. 5.2 Illustration of the `CyberAudioSlider` class implementation

5.3 Labels and Information Widgets

Information labels and widgets play a key role in presenting data and statuses in the user interface. At Wavelength, these components are designed for readability, visual consistency, and effective communication of information in the context of a cyberpunk aesthetic.

5.3.1 User Label

`UserInfoLabel` is a custom label that displays text next to a colored circle with a glow effect. It is used to present user information, such as username, with a visual status indicator. The color of the circle can be dynamically changed, and the glow effect adds a subtle visual accent.

5.3.2 Overlay widget

`OverlayWidget` is a semi-transparent overlay widget that is used to darken the background content. It is used in conjunction with dialogs or side panels to focus the user's attention on a particular interface element. The widget automatically adjusts to the size of the parent and allows you to exclude specific areas from the dimming effect.

5.4 Special Widgets

Special widgets are advanced interface components that go beyond standard GUI elements to offer unique functionality and visual effects. In ***Wavelength***, These widgets are designed with an immersive user experience and a consistent cyberpunk aesthetic in mind.

5.4.1 Animated widget stack

`AnimatedStackedWidget` is a custom `QStackedWidget`, which provides animated transitions between widgets. It supports different types of animation (fade, pan, push) and optional use of OpenGL for smoother transitions.

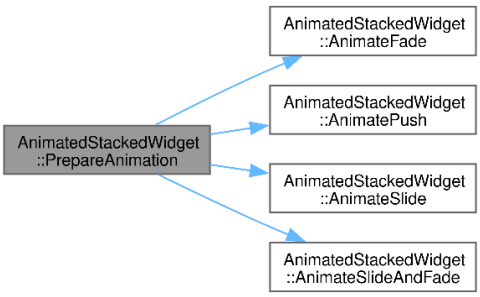

Fig. 5.4 Presentation of animation types of animated widget stack

5.4.2 GL transition widget

`GLTransitionWidget` is an OpenGL widget that renders smooth, animated sliding transitions between two other widgets. It uses QPainter in an OpenGL context to draw sliding pixmaps.

6. Chat Architecture

6.1 Communication Stream

`CommunicationStream` mentioned in Chapter 3, is the visual heart of the chat interface in the ***Wavelength*** application, implementing an OpenGL rendering system that combines dynamic wave animation with interactive message presentation elements.

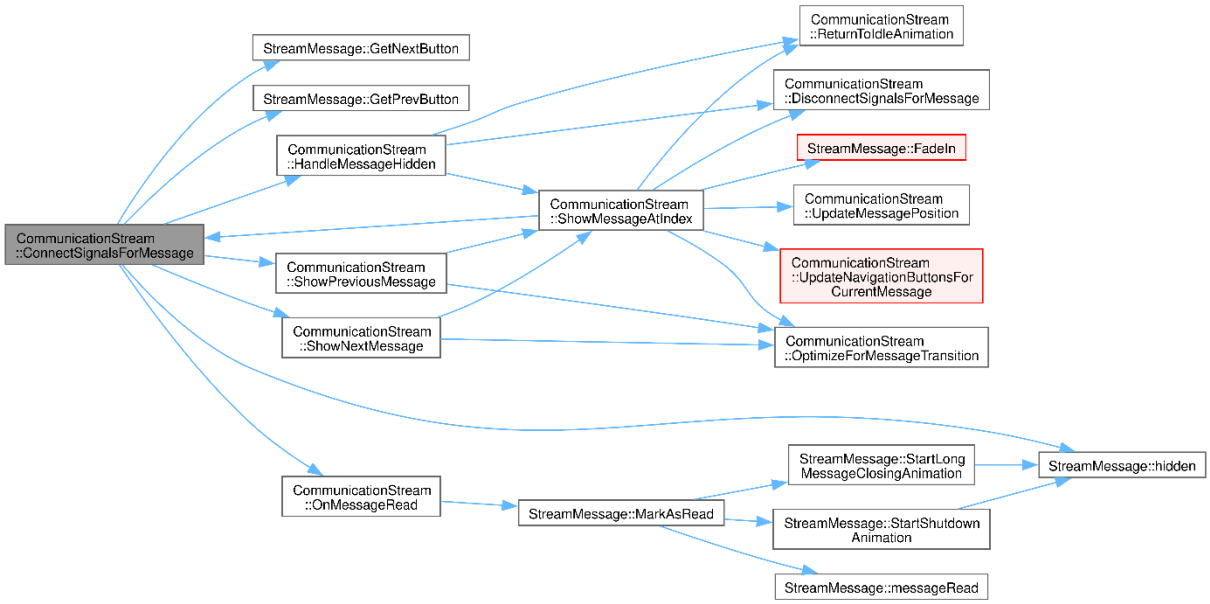

Fig. 6.1 Signal connections in the communication stream class

6.1.1 OpenGL rendering architecture

`CommunicationStream` uses a state-of-the-art OpenGL pipeline to render a dynamic wave that responds to sound levels and communication status. The wave geometry is procedurally generated in the vertex shader, and its appearance is controlled by a number of parameters such as amplitude, frequency, velocity and thickness. The fragment shader adds visual effects such as grid lines, noise and scanning to create a distinctive cyberpunk look.

The system uses `QOpenGLVertexArrayObject` (VAO) and `QOpenGLBuffer` (VBO) for efficient management of geometry and vertex attributes. The program shader is compiled and linked during initialization, and animation parameters are passed as uniforms to the shaders in each frame.

6.1.2 Dynamic wave animation

The wave animation is driven by `QTimer`, which calls the slot `UpdateAnimation` at regular intervals. This slot updates animation parameters such as time offset, wave amplitude and noise intensity. The wave amplitude is interpolated toward a target value, which is set based on the sound level.

The system implements three visual states:

1. Idle (default state with simple wave and occasional interference),
2. Receiving (active animation when receiving messages)
3. Displaying (state when the message is actively displayed).

Transitions between states are animated using the `QPropertyAnimation`, which ensures smooth and visually appealing changes.

6.1.3 Message displaying

`CommunicationStream` manages the list of messages (`StreamMessage`) and displays them superimposed on the wave animation. Messages are added to the stream using methods `AddMessage` and/or `AddMessageWithAttachment`, which create objects `StreamMessage` and initiate the receiving animation.

The system implements message navigation using the arrow keys (left/right) and closing messages using the Enter key.

Slots `ShowNextMessage` and `ShowPreviousMessage` update the index of the current message and display the corresponding message. Slot `OnMessageRead` is called when the user closes the message, which can remove the message from the list or return to the Idle state.

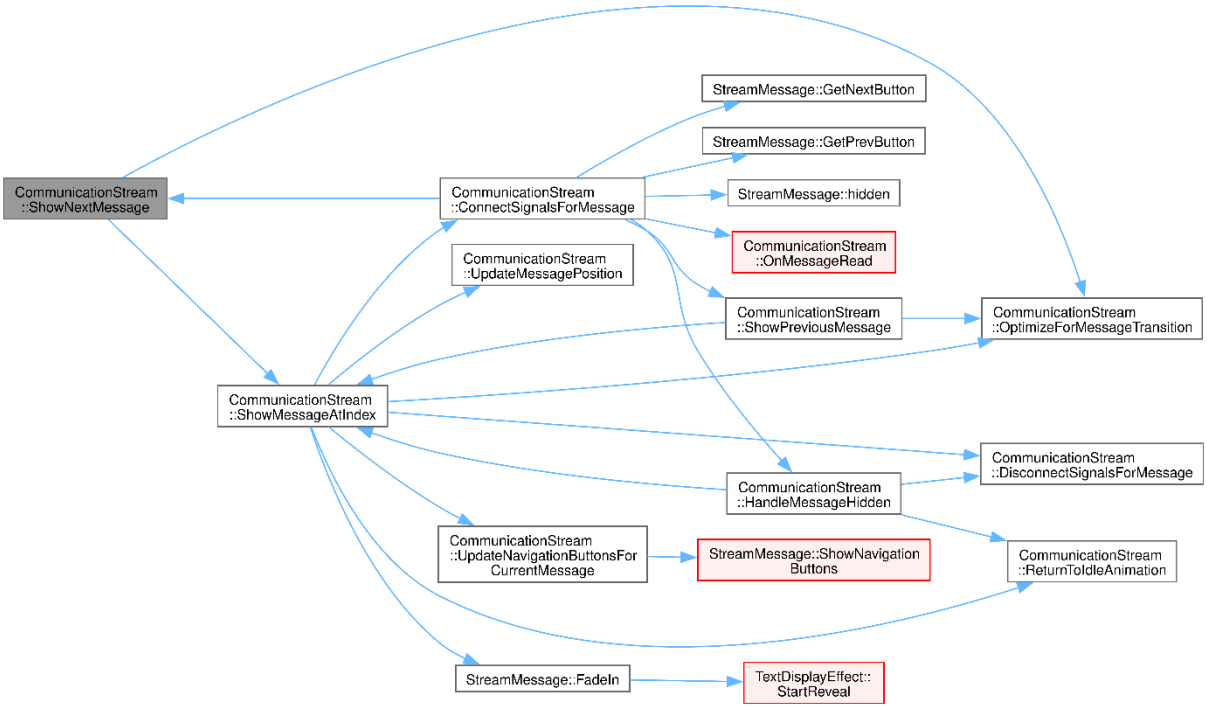

Fig. 6.1.3 Visualization of how the method of showing the next message in a row works

6.1.4 Visual effects and personalization

`CommunicationStream` implement a number of visual effects that enhance the user experience. Distortion effects are randomly generated in the Idle state, adding an element of unpredictability and dynamism. The color of the stream can be customized using the configuration settings, allowing you to personalize the appearance.

The system also implements the display of the user ID of the user who is currently transmitting audio. The user ID is displayed in the upper right corner of the stream, and its color is generated based on the ID hash.

6.1.5 Audio system integration

`CommunicationStream` integrates with the app's audio system, responding to sound levels and visualizing them in a wave animation. Method `SetAudioAmplitude` sets the target wave amplitude based on the sound level, causing the wave to pulse in rhythm with the sound.

6.2 Stream Display

`StreamDisplay` provides a central point for managing and presenting messages in the ***Wavelength*** application's chat interface. Acting as a controller for the `CommunicationStream`, This class implements an advanced message queuing, processing and visualization system, providing a smooth and responsive user experience.

6.2.1 Message queuing and processing architecture

`StreamDisplay` uses an asynchronous queue-based message processing model (`QQueue`) and timer (`QTimer`). New messages are added to the queue using the slot `AddMessage`, and then processed sequentially with a delay, creating a gradual display effect. This architecture avoids blocking the main UI thread during intensive communication.

The system implements the distinction between regular messages and progress messages (`progress_messages`), which are identified by a unique identifier (`message_id`). Progress messages allow you to update the contents of existing messages, which is especially useful when transferring files.

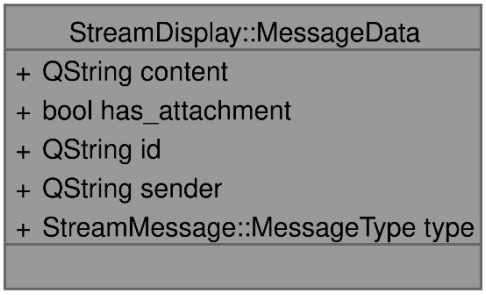

Fig. 6.2.1 Structure storing message information

6.2.2 Message lifecycle management

`StreamDisplay` manages the life cycle of displayed messages with a map (`QMap`) `displayed_progress_messages_`, which tracks currently displayed progress messages.

When the progress message is destroyed, the slot `OnStreamMessageDestroyed` removes the corresponding entry from the map, preventing memory leaks and incorrect updates.

The system also implements the method `Clear`, which removes all messages from the display and processing queue, providing a clean initial state.

6.2.3 CommunicationStream integration

`StreamDisplay` works closely with the `CommunicationStream`, by giving it control signals such as sound levels (`SetAudioAmplitude`), ID of the transmitting user (`SetTransmittingUser`) and intensity of interference (`SetGlitchIntensity`). This architecture makes it possible to separate message management logic from visualization logic, which increases the modularity and testability of the system.

6.2.4 Dynamic frequency adjustment

Method `SetFrequency` allows you to dynamically set the frequency and name of the stream, which updates the displayed information in the `CommunicationStream`.

This method also cleans all existing messages, ensuring a consistent interface state.

6.3 Stream Message

`StreamMessage` provides the primary visual element in the ***Wavelength*** application's communication stream, representing a single text message or attachment. This component goes beyond standard message windows, implementing advanced visual effects, animations and interactive elements that are consistent with the application's cyberpunk aesthetic.

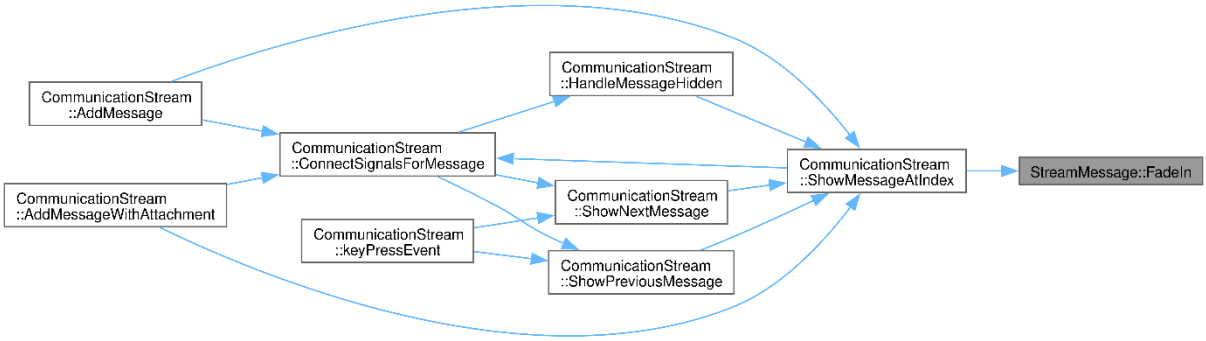

Fig. 6.3. Illustration of the cooperation of the news class with the stream class

6.3.1 Architecture and message types

`StreamMessage` It supports three main types of messages:

1. Received (`kReceived`),
2. Sent (`kTransmitted`)
3. System (`kSystem`)

Each of these types has a distinct color scheme and visual style. The component architecture is based on the Qt class hierarchy, using the `QWidget` as a base and `QPropertyAnimation` for seamless control of visual properties such as transparency, glare intensity and electronic shutdown progress.

The system implements dynamic switching between different methods of displaying content depending on the length of the message. Short messages are displayed using an animated text reveal effect (`TextDisplayEffect`), when long messages are placed in the scroll area (`QscrollArea`) with long text effect (`LongTextDisplayEffect`).

6.3.2 Visual effects and animations

`StreamMessage` implements a number of visual effects and animations that enhance the user experience.

`ElectronicShutdownEffect` simulates the shutdown of an old CRT TV or monitor, gradually darkening and distorting the message. Glow effect adds a subtle visual touch that animates as you interact with the message.

The system also implements fade-in/fade-out animations, which are used when displaying and hiding messages. These animations are controlled using `QPropertyAnimation`, which ensures smooth and visually appealing transitions.

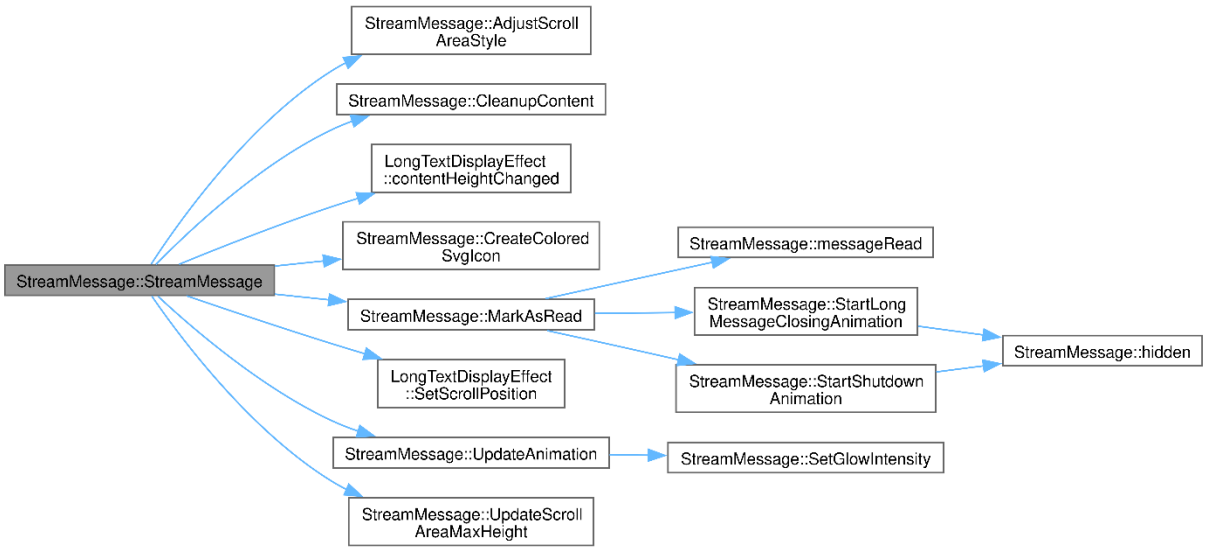

Fig. 6.3.2. A number of animations related to the news class

6.3.3 Attachment handling

`StreamMessage` handles attachments using a special tag in the HTML content. Method `AddAttachment` analyzes the HTML content, extracts the attachment information (ID, MIME type, filename) and creates an appropriate widget to display the attachment (`AttachmentViewer`).

The system implements dynamically adjusting the size of the message according to the content, especially when the attachment is loaded. The AdjustSizeToContent method calculates the optimal message size based on the dimensions of the attachment and screen limitations.

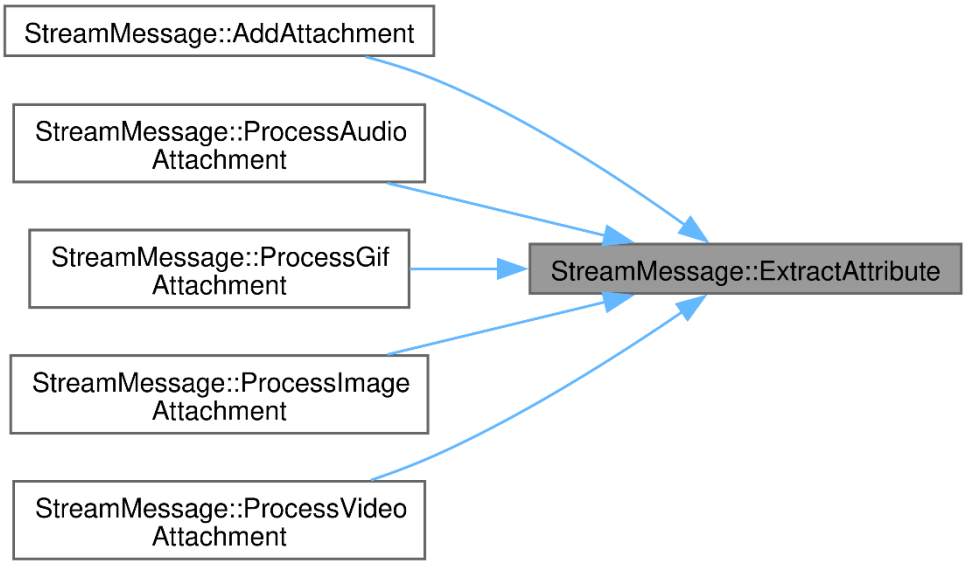

Fig. 6.3.2. Extracting attachment information in a message class

6.3.4 User interaction

`StreamMessage` implements user interaction with navigation buttons (next/previous message) and a „Mark as Read" button (visible as a check in the application - ✅).

Navigation buttons allow users to browse the message stream, while a „Mark as Read" button closes a message and removes it from the display.

The system also implements support for keyboard events, allowing users to navigate through messages using the arrow keys and close messages using the Enter key.

6.3.5 Custom painting and cyberpunk aesthetics

`StreamMessage` uses custom painting to render a distinctive cyberpunk look. Method `paintEvent` draws a background, neon border, glow effect, decorative lines and AR-style markers, as well as sender text and other visual elements based on the message type.

The system also implements dynamic style customization of the scrollbar area based on the message type, ensuring a consistent interface appearance.

7. Message Handling

7.1 Message Handler

`MessageHandler` is the central node for message operations in the ***Wavelength*** communications architecture, orchestrating the processes of creating, analyzing and extracting data from JSON messages sent via WebSocket. Implemented as a singleton, the class provides consistent and controlled access to message manipulation functionality across the system.

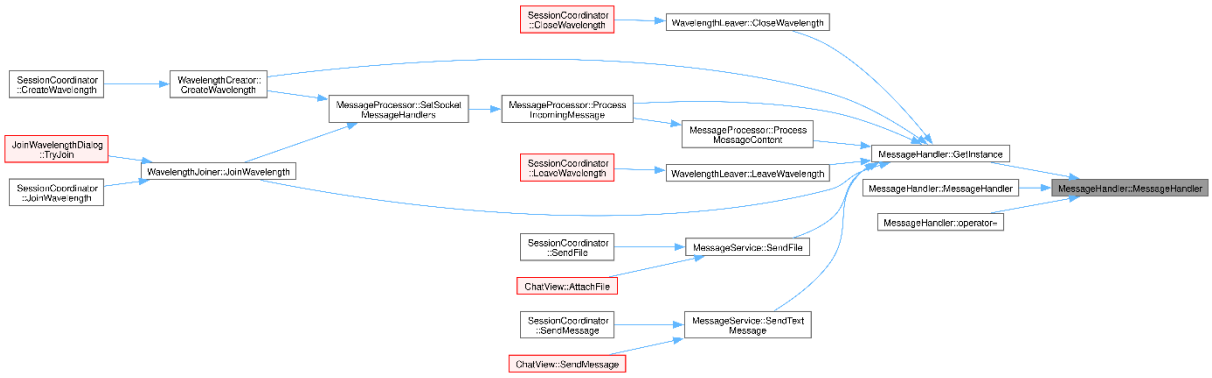

Fig. 7.1. Operation node visualization for message handler class

7.1.1 Generating and sending system commands

The component provides static methods for generating unique message identifiers (UUIDs) and constructing and sending system commands in JSON format. Function `SendSystemCommand` encapsulates the process of creating a JSON object with a command type and parameters, converting it to a string and sending it through a designated WebSocket.

7.1.2 Preventing duplicate messages

`MessageHandler` implements a mechanism to prevent the processing of duplicate messages by maintaining an internal set of (`QSet`) identifiers of already processed messages. Method `IsMessageProcessed` verifies that the identifier is in the collection and `MarkMessageAsProcessed` adds a new identifier while managing the size of the cache.

7.1.3 Message processing and data extraction

The class provides static methods for processing JSON messages (function `ParseMessage`) and extraction of key fields such as type, content, frequency and sender ID. These features provide abstraction from the implementation details of the JSON format, facilitating access to data in a typically secure manner.

7.1.4 Create protocol-specific requests

`MessageHandler` contains a set of static methods to create JSON objects representing Wavelength protocol-specific requests, such as authentication, registration and drop frequency requests. These functions encapsulate the logic of message creation, promoting consistency and reducing code redundancy

7.1.5 Status management and resource cleaning

The class provides method `ClearProcessedMessages` to clear the internal cache of processed message IDs, enabling status management and preventing excessive memory consumption.

7.2 Message Service

`MessageService` is the central messaging service in the ***Wavelength*** communications architecture, encapsulating the logic for sending text messages, files, Push-to-Talk (PTT) requests and raw audio data over WebSocket connections. This singleton service manages the sending process, using a `WavelengthRegistry` to locate the appropriate WebSocket connections for a given frequency.

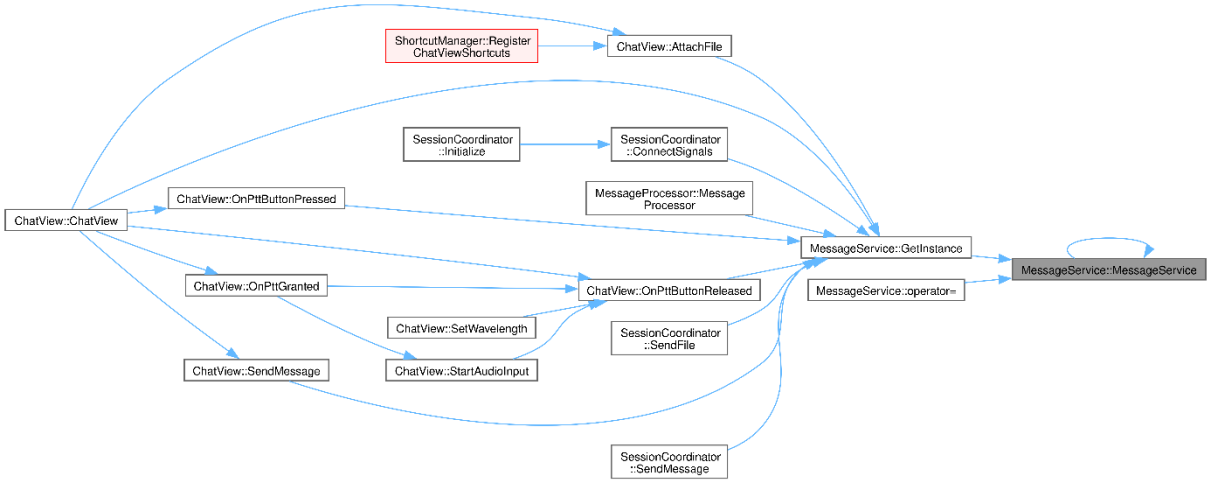

Fig. 7.2. Illustration of the coordination of the central messages desk

7.2.1 Sending text messages and files

The component provides methods to send text messages (function `SendTextMessage`) and files (function `SendFile`). Sending files is done asynchronously using the `AttachmentQueueManager`, which prevents the main UI thread from blocking. Files are encoded to base64 before sending, and the progress of sending is tracked using signals `progressMessageUpdated`.

Implementation of function `SendFile` can be seen below:

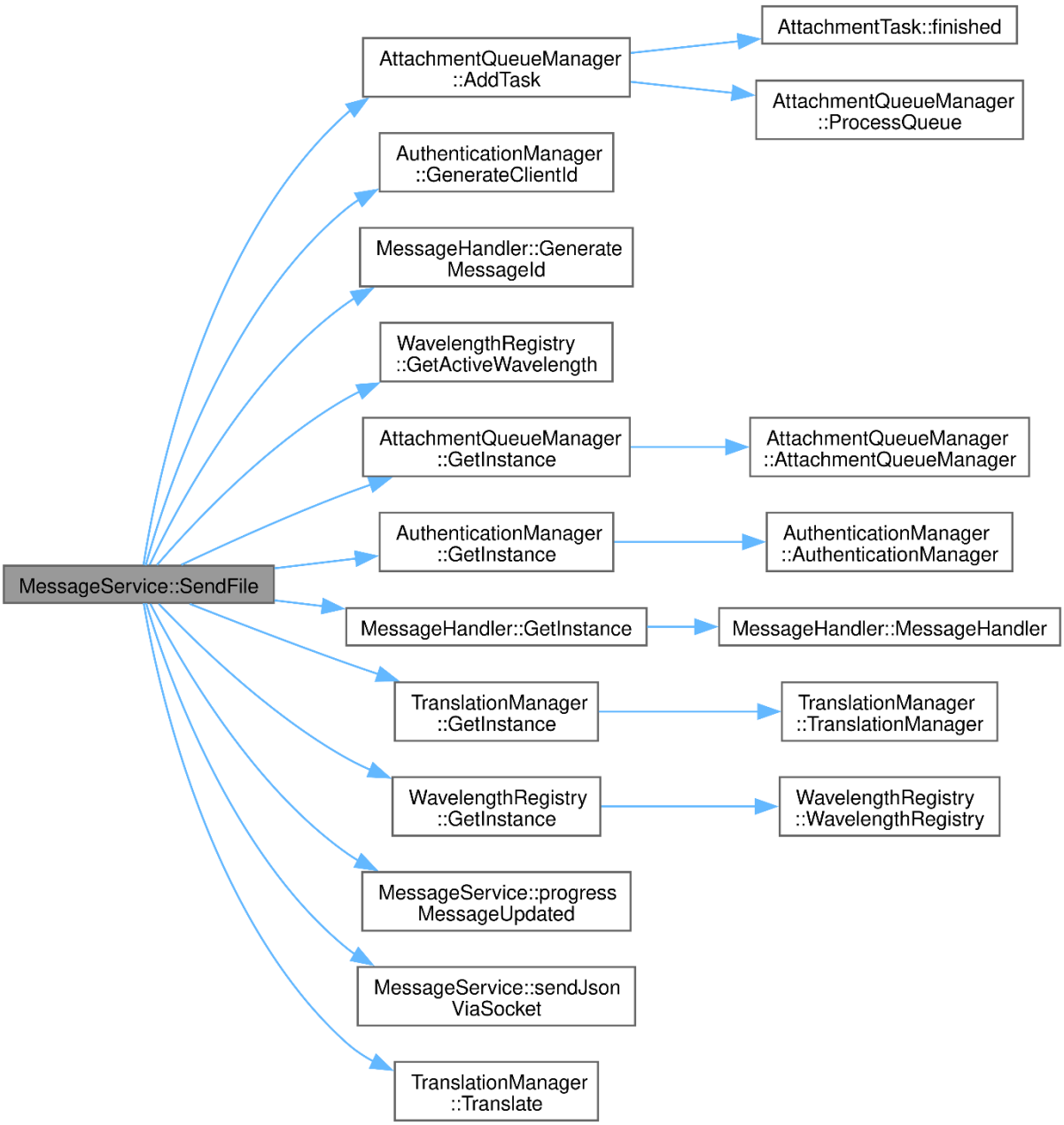

Fig. 7.2. Visualize the file upload method from the perspective of the news service

7.2.2 Push-To-Talk (PTT)

`MessageService` manages the PTT process, providing methods to send requests (function `SendPttRequest`) and releases (function `SendPttRelease`) for a specific frequency. In addition, the component allows sending raw audio data (function `SendAudioData`) during an active PTT session.

7.2.3 WebSocket connection management

The component uses `WavelengthRegistry` to locate the appropriate WebSocket connections for a given frequency. Method `GetSocketForFrequency` fetches a pointer to a WebSocket while checking its validity.

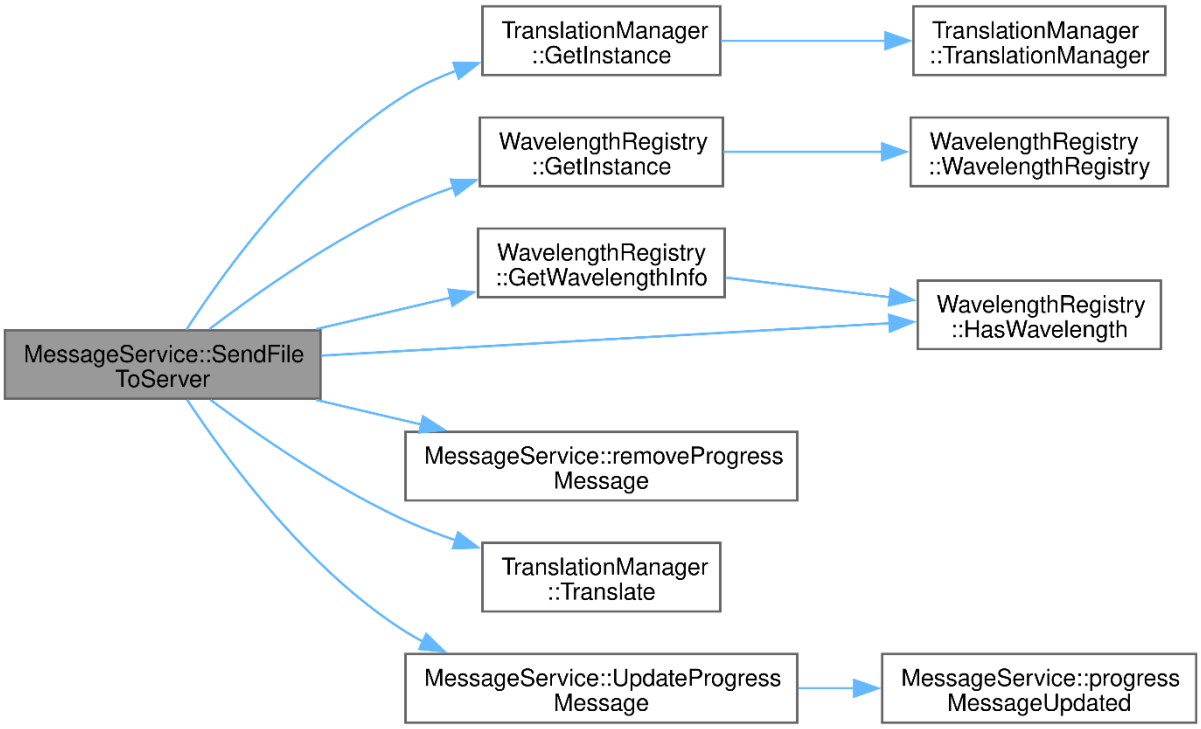

Fig. 7.2.3. Using the Wavelength register when transferring a file to the server

7.2.4 Signaling and event handling

`MessageService` emits a series of signals that indicate various message, PTT and audio data events. These signals include:

- `messageSent`: emitted after successfully sending a text message
- `progressMessageUpdated`: emitted while processing and uploading files for status updates
- `removeProgressMessage`: emitted upon completion (success or failure) of file upload to remove the progress message
- `sendJsonViaSocket`: emitted by the background file processing task when the JSON message is ready to be sent
- `pttGranted`: emitted when the server grants permission for audio transmission (PTT)

- `pttDenied`: emitted when the server refuses to allow audio transmission (PTT)
- `pttStartReceiving`: emitted when another user starts audio transmission on a given frequency
- `pttStopReceiving`: emitted when the currently transmitting user stops sending audio
- `audioDataReceived`: emitted when raw audio data is received by WebSocket
- `remoteAudioAmplitudeUpdate`: emitted when the server sends an update about the amplitude of the remote audio

7.2.5 Message caching and customer ID

`MessageService` stores a buffer of sent text messages (map `sent_messages_`) and the customer ID (field `client_id_`). The message buffer allows sent messages to be redisplayed, and the customer ID identifies a unique instance of the service.

7.3 Message Processor

`MessageProcessor` is the central component for processing incoming WebSocket messages in a ***Wavelength*** application, acting as an intelligent dispatcher that analyzes, verifies and redirects messages to the appropriate handling procedures. This singleton component ensures consistent interpretation of messages, prevents duplicates and emits signals to inform about system events.

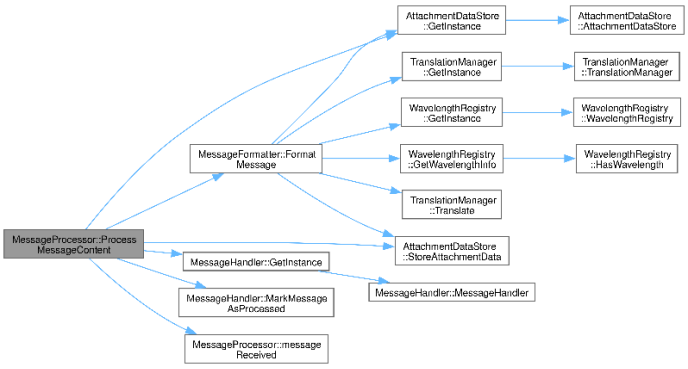

Fig. 7.3. Illustration of how the message processor works when processing content

7.3.1 Message analysis and verification

The component receives raw text (JSON) and binary (audio data) messages from the WebSocket and then performs a series of verifications, including checking the message ID, frequency matching and identifying the message type.
The method `ProcessIncomingMessage` analyzes the JSON structure and then passes control to the appropriate private methods depending on the type of message.

7.3.2 Support for different types of communication

`MessageProcessor` supports different types of messages, including:

- Text messages (type *`message`* or *`send_message`*): processed by `ProcessMessageContent`, which checks for duplicates, handles attachments, and formats the message with a `MessageFormatter` and emits a signal `messageReceived`
- System commands (type *`system_command`*): processed by `ProcessSystemCommand`, which supports commands such as *`ping`*, *`close_wavelength`* and *`kick_user`*, By emitting appropriate signals (e.g. `userKicked`, `wavelengthClosed`).
- User join/leave events (type *`user_joined`* and *`user_left`*): processed by `ProcessUserJoined` i `ProcessUserLeft`, that format system messages and emit a signal `systemMessage`.
- Frequency closure (type *`wavelength_closed`* or command *`close_wavelength`*): processed by `ProcessWavelengthClosed`, which updates `WavelengthRegistry` and emits signal `wavelengthClosed`.
- Binary data (audio data): processed by the `ProcessIncomingBinaryMessage`, which emits a signal`audioDataReceived`.

7.3.3 WebSocket connection management

The component provides a method `SetSocketMessageHandlers`, which combines signals `textMessageReceived` and `binaryMessageReceived` WebSocket sockets with the corresponding processing methods in the class `MessageProcessor`. This method provides dynamic assignment of message handling routines to WebSocket connections.

7.3.4 Event signaling

`MessageProcessor` emits a series of signals that inform about various events in the system, including:

- `messageReceived`: emitted after processing a simple text message or message with an attachment
- `systemMessage`: Emitted after processing a system message (e.g., joining/leaving a user)
- `wavelengthClosed`: emitted when the frequency is closed
- `userKicked`: emitted when the user is ejected from the frequency of the
- `pttGranted`: emitted when the server grants permission for audio transmission (PTT)
- `pttDenied`: emitted when the server refuses to allow audio transmission (PTT)
- `pttStartReceiving`: emitted when another user starts audio transmission on a given frequency
- `pttStopReceiving`: emitted when the currently transmitting user stops sending audio
- `audioDataReceived`: emitted when raw audio data is received by WebSocket
- `remoteAudioAmplitudeUpdate`: emitted when the server sends an update about the amplitude of the remote audio

7.4 Message Formatter

`MessageFormatter` is a key component of Wavelength's message presentation system, providing static utility functions for transforming message data into HTML format, ready for display in the chat interface. This class encapsulates the logic for formatting, timestamp management, sender identification and attachment handling, ensuring a consistent message appearance.

A demonstration of how it works can be found below:

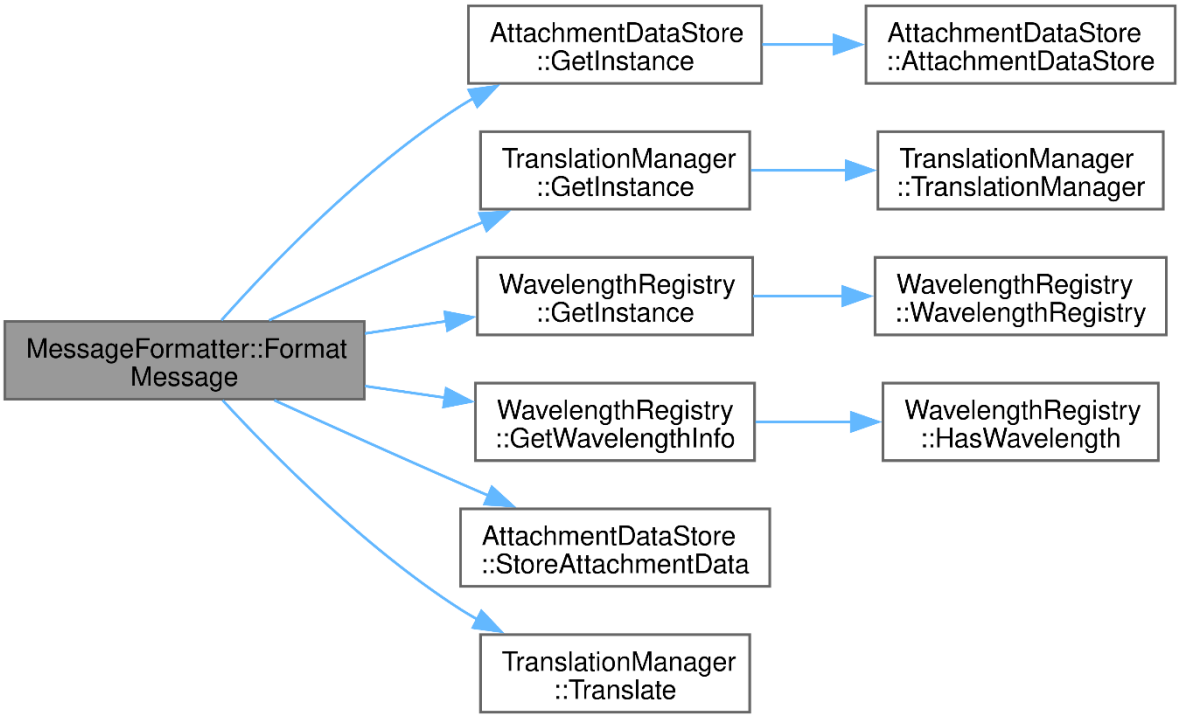

Fig. 7.4. Visualization of inter-class message formatter connections

7.4.1 Formatting JSON messages into HTML

The main function of the component is `FormatMessage`, which parses a JSON object representing the message, extracting information about the sender, content, timestamp and attachments. The method generates HTML code that includes the formatted text of the message, sender information (along with the „You” tag for custom messages) and placeholders for attachments.

7.4.2 Support for attachments and placeholders

For messages with attachments, `FormatMessage` generates a div element with data-* attributes that store information about the attachment (identifier, MIME type, file name). These attributes allow the attachment to be dynamically loaded and displayed later in the interface.

7.4.3 Formatting file size and system messages

The component also provides functions for formatting the file size in a user-readable way (e.g., „1.2 MB”) and for formatting system messages, adding a timestamp and styling the text.

8. Chat Visual Effects

Visual effects play a key role in creating an immersive and engaging user experience in the ***Wavelength*** application. These components, while primarily visual, significantly affect the perception of content and interaction with the chat interface.

8.1 Text display effect

`TextDisplayEffect` is a custom widget that displays text with a cyberpunk terminal-style „typing” (or more like „typing out”) effect. It simulates typing text character by character, with an optional (though enabled by default) typing sound, random noise and a pulsing cursor.

8.2 Long text display effect

`LongTextDisplayEffect` is a widget for displaying potentially long text with a cyberpunk effect and efficient rendering. It wraps lines of text based on the width of the widget, renders only the visible part of the text, has a background gradient and a scanning effect.

8.3 Electronic shutdown effect

`ElectronicShutdownEffect` is a graphic effect that simulates the shutdown of an old CRT monitor or electronic device. It animates in two phases: rolling the image into a thin line with noise and fading the line with glow effects.

8.4 Mask overlay effect

`MaskOverlayEffect` is a custom overlay widget that creates a „revealing” effect using a moving scan line. Initially, it covers another widget with a semi-transparent mask with interference, and the scan line reveals the contents of the widget from underneath.

9. File and Attachment System

The attachment management system allows users to exchange files of various types in an efficient, secure and visually consistent manner with the overall aesthetic of the application. These components, working in harmony, provide a seamless user experience when sending, receiving and viewing attachments.

9.1 Attachment Placeholder

`AttachmentPlaceholder` acts as a temporary container for attachments in the chat stream, presenting information about the file (name, type) and allowing the content to be loaded and displayed asynchronously. This component implements dynamic loading, error handling and user interaction.

The component's architecture is based on an asynchronous loading model, using a `AttachmentQueueManager` for managing background tasks. After initialization, `AttachmentPlaceholder` automatically adds the loading task to the queue, and then updates the user interface based on the progress and result of the operation.

The component supports different types of attachments (images, GIFs, audio, video) by dynamically creating appropriate widgets (`ImageViewer`, `GifPlayer`, `AudioPlayer`, `VideoPlayer`). The system also implements scaling and cropping mechanisms, ensuring optimal display of attachments in the chat stream independent of the application window size.

Below is an example of using the placeholder class, for displaying a video file:

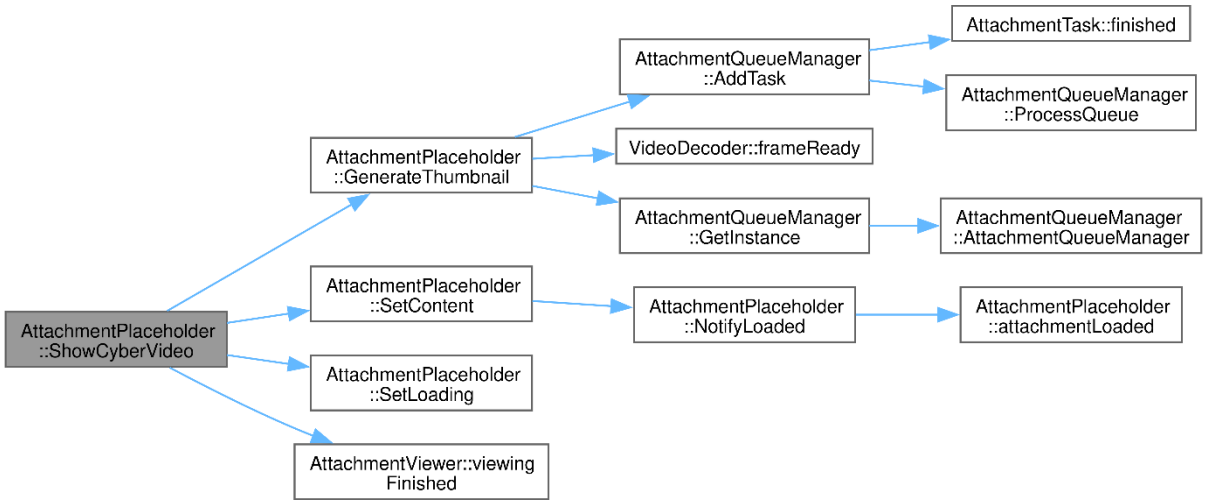

Fig. 9.1. Use of `AttachmentPlaceholder` for creating a placeholder for a video file

9.2 Attachment Queue Manager

`AttachmentQueueManager` manages the task queue for loading and processing attachments, limiting the number of concurrent operations to avoid overloading the system. This singleton component provides thread-safe access to the task queue and efficient use of CPU resources.

The component architecture is based on `QThreadPool`, which manages a pool of worker threads. `AttachmentQueueManager` adds tasks to a queue and then runs them in available threads. When the task finishes, the component emits a signal, indicating that the operation is complete.

The system also implements task prioritization mechanisms, allowing higher priority to be given to tasks related to the display of attachments in the currently displayed message.

9.3 Attachment Data Store

`AttachmentDataStore` temporarily stores Base64-encoded attachment data, allowing access to the data from different threads in a thread-safe manner. This singleton component provides fast access to attachment data without reloading from disk or the network.

The component's architecture is based on `QMap`, which stores attachment data under a unique identifier (UUID). Access to the map is protected by QMutex, providing thread-safe read and write operations.

The system also implements data cleaning mechanisms, removing unnecessary attachments from storage after they have been processed or displayed.

9.4 Automatic scaling of attachments

`AutoScalingAttachment` Automatically scales the attachment content (e.g. image, GIF) to the maximum allowed size, preserving the aspect ratio. This component ensures optimal display of attachments in the chat stream, preventing the available space from being exceeded.

The component architecture is based on `QWidget`, Which contains another widget (attachment content). `AutoScalingAttachment` monitors the size of the content and scales it as needed, preserving the aspect ratio (aspect-ratio).

The system also implements user interaction by displaying a „Click to enlarge” label when the cursor hovers over an attachment. Clicking on an attachment displays it at full size in a separate dialog box.

10. Multimedia Support

The media handling system is an integral part of the ***Wavelength*** application, allowing users to exchange a variety of media file formats in a smooth and efficient manner.

10.1 Images

10.1.1 Image decoder

`ImageDecoder` provides the foundation for handling static images in a ***Wavelength*** application, using Qt's native capabilities to decode image data. This class, while relatively simple to implement, provides a solid foundation for displaying popular image formats such as PNG and JPEG.

The component's architecture is based on a synchronous decoding model, using the QImage::loadFromData() method to process image data.

After successful decoding, ImageDecoder emits imageReady() and imageInfo() signals, transmitting the decoded image and information about its dimensions and alpha channel. If an error occurs, an error() signal is emitted, informing the reason for the failure.

10.1.2 Image Viewer

`ImageViewer`, based on the image decoded by the `ImageDecoder`, presents it in the user interface, automatically scaling the content to fit the available space. This component goes beyond simple image display by implementing asynchronous loading, error handling and dynamic resizing.

The component's architecture is based on an asynchronous loading model, using a `QTimer::singleShot()` for a delayed method call `LoadImage()`, which initiates
the decoding process. After receiving the signal `imageReady()` from `ImageDecoder`, `ImageViewer` updates content of `QLabel`, displaying the decoded image.

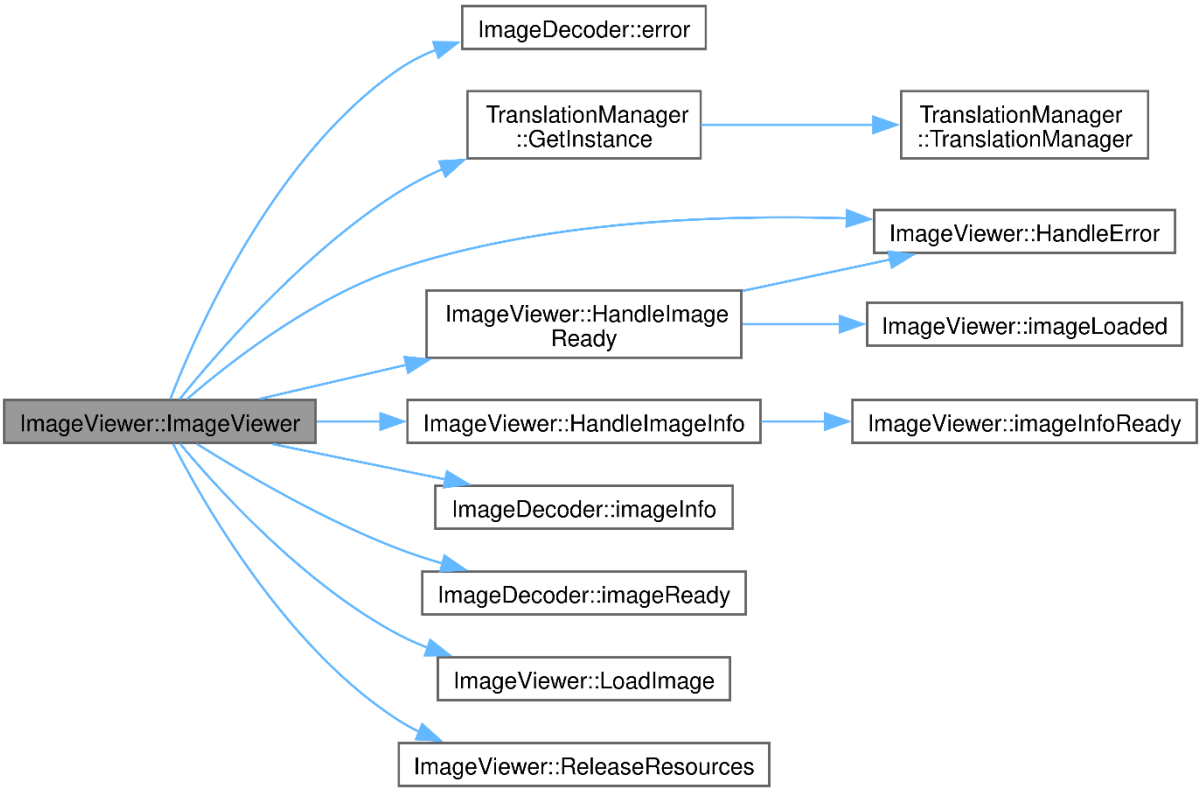

Fig. 10.1.2. Connections of the photo display along with the decoder

10.2 GIF

10.2.1 GIF Decoder

`GifDecoder` provides an advanced component for decoding animated GIF images, using the FFmpeg library to process data asynchronously. Running in a separate thread, the class provides smooth playback of GIF animations without blocking the main UI thread.

The component's architecture is based on a multi-threaded decoding model, using a `QThread` to run the decoding loop. `GifDecoder` uses FFmpeg to read GIF data, decode the frames, and convert the pixel format to a `QImage::Format_RGBA8888`. After decoding each frame, a signal is emitted `frameReady()`, transmitting the decoded image.

10.2.2 GIF Player

`GifPlayer`, based on the decoded frames by the `GifDecoder`, presents a GIF animation in the user interface, automatically starting playback when the mouse cursor is hovered. This component goes beyond simple animation playback by implementing dynamic scaling, error handling and user interaction.

The component's architecture is based on an asynchronous playback model, using signals `enterEvent()` and `leaveEvent()` to control the playback status. When you hover the mouse cursor, `GifPlayer` calls the method `StartPlayback()`, which starts the animation playback. When you leave the mouse cursor, the method `StopPlayback()` is called , which stops playback.

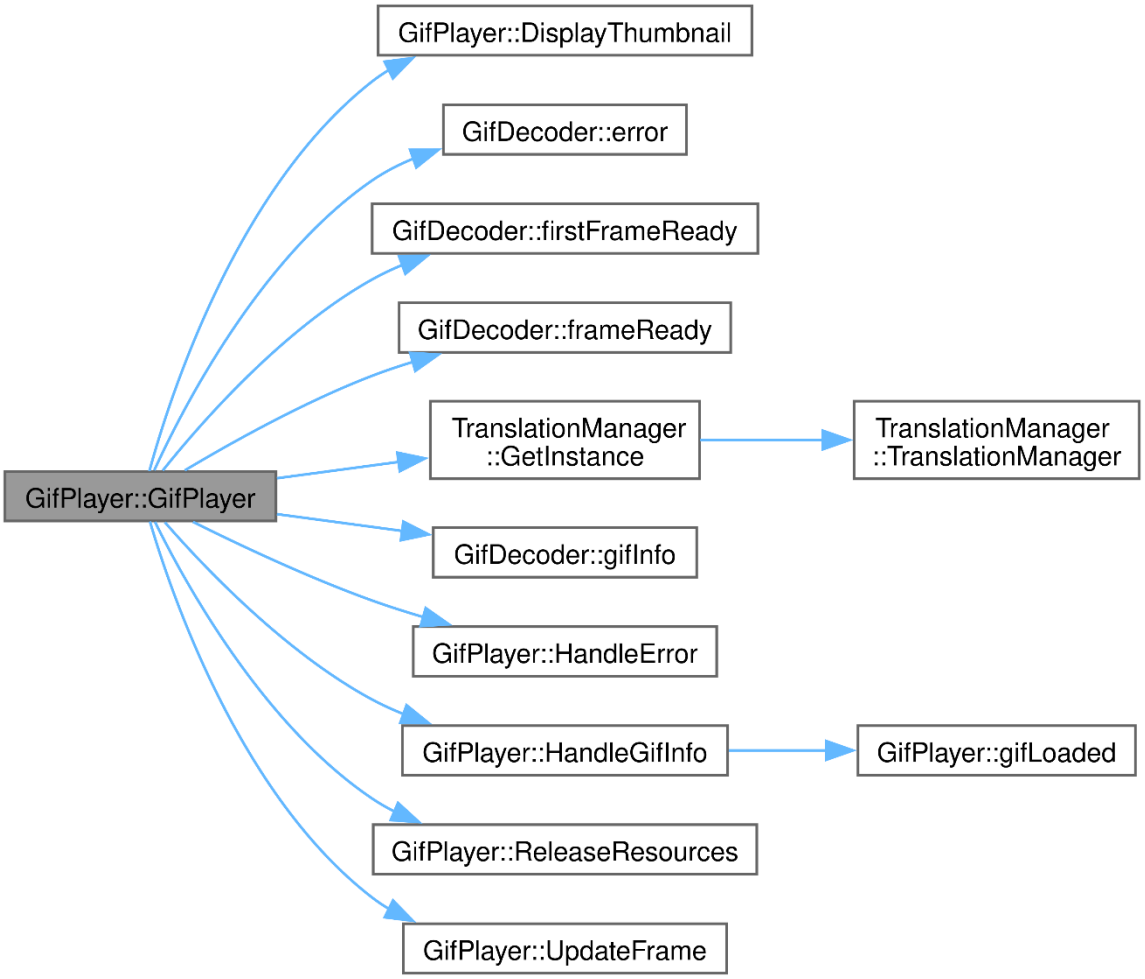

Fig. 10.2.2. Synchronization of the GIF decoder with the corresponding player

10.3 Audio

10.3.1 Audio Decoder

`AudioDecoder` provides an advanced component for decoding audio files, using the FFmpeg library to process data in an asynchronous manner. Running in a separate thread, the class provides smooth audio playback without blocking the main UI thread.

The component's architecture is based on a multi-threaded decoding model, using a `QThread` to run the decoding loop.

`AudioDecoder` uses FFmpeg to read audio data, decode samples and convert the format to `QAudioFormat`. After decoding each sample, the data is sent to the `QAudioOutput` for audio playback.

10.3.2 Audio Player

`AudioPlayer`, based on decoded samples by the `AudioDecoder`, plays audio in the user interface, offering full control over playback. This component goes beyond simple audio playback by implementing custom controls, spectrum visualization and dynamic volume adjustment.

The component's architecture is based on a custom user interface that includes play/pause buttons, progress slider, volume slider and spectrum visualization. `AudioPlayer` uses Qt signals and slots to handle user interaction and playback status updates.

Below is the full synchronization between the audio player and its decoder:

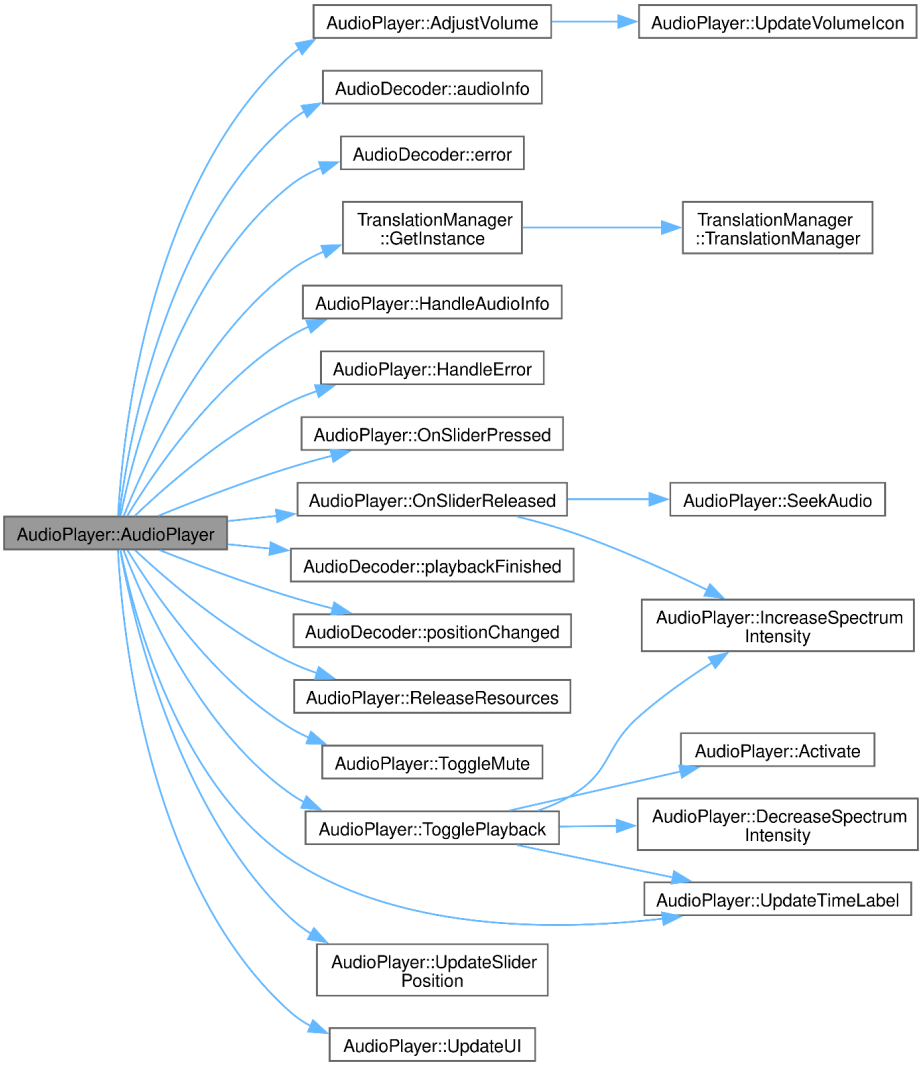

Fig. 10.3.2. Full synchronization of the audio player with its decoder

10.4 Video

10.4.1 Video Decoder

`VideoDecoder` is an advanced component for decoding video files, using the FFmpeg library to process data in an asynchronous manner. Running in a separate thread, the class provides smooth video playback without blocking the main UI thread.

The component's architecture is based on a multi-threaded decoding model, using a `QThread` to run the decoding loop. `VideoDecoder` uses FFmpeg to read the video data, decode the frames and convert the pixel format to a `QImage::Format_RGB888`. After decoding each frame, the frameReady() signal is emitted, transmitting the decoded image.

10.4.2 Video Player

`VideoPlayer`, based on the decoded frames by the `VideoDecoder`, presents video in the user interface, offering full control over playback. This component goes beyond simple video playback by implementing custom controls, visual effects and dynamic resizing.

The component's architecture is based on a custom user interface that includes play/pause buttons, a progress slider, a volume slider and information labels. `VideoPlayer` uses Qt signals and slots to handle user interaction and playback status updates.

Below is the full synchronization between the video player and its decoder:

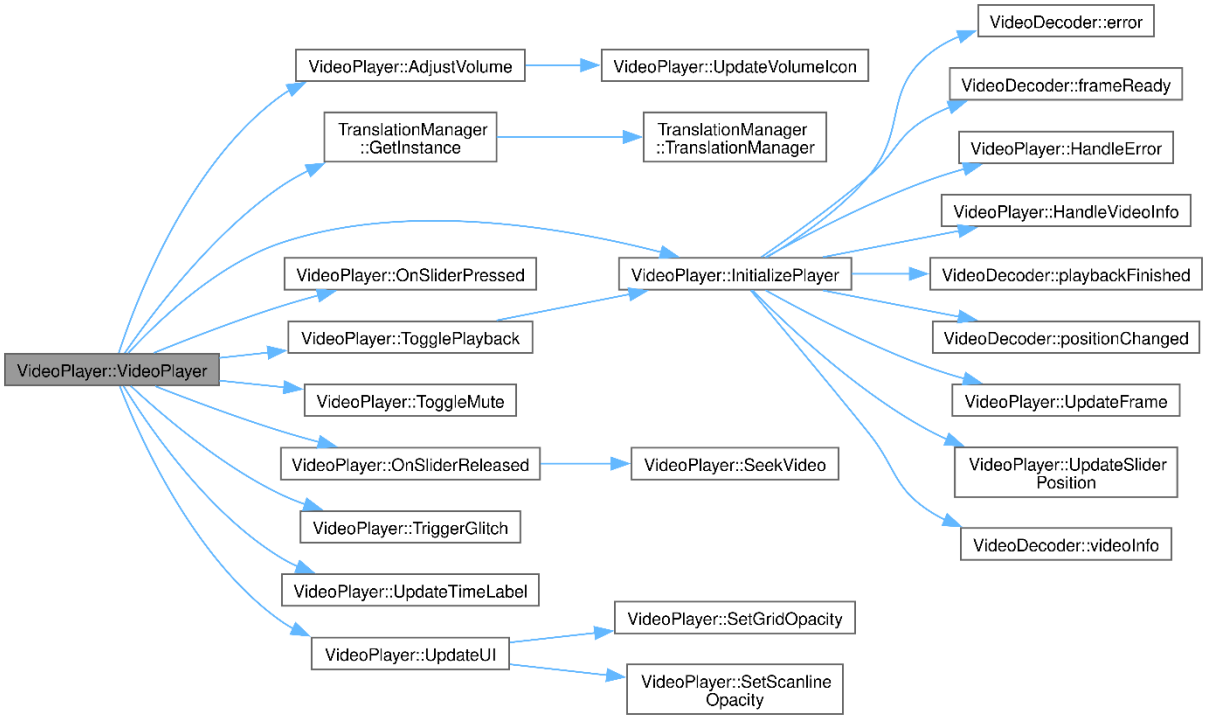

Fig. 10.4.2. Full synchronization of the video player with its decoder

10.5 Main Media Viewer

`AttachmentViewer` provides a versatile container that integrates and presents a variety of attachment types in the Wavelength application, providing a consistent and engaging user experience. This component goes beyond simple content display by implementing cyberpunk-style decryption animation, dynamic layout management and user interaction.

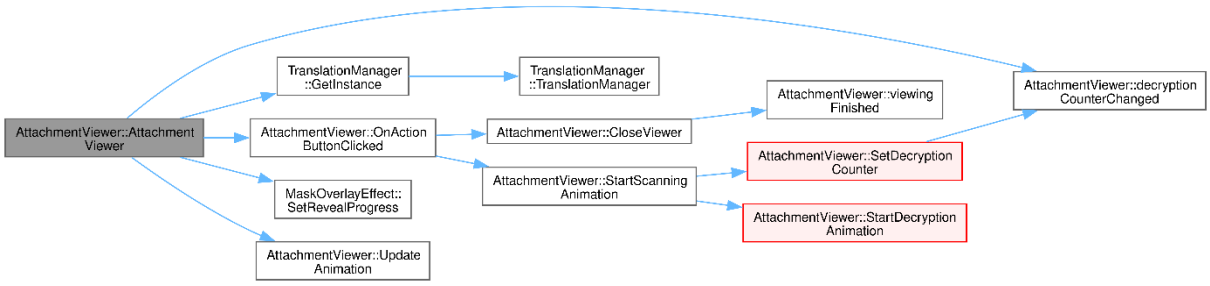

Fig. 10.5. Visualization of how the attachment viewer class works

10.5.1 Decryption animation

A key element of `AttachmentViewer` is a decryption animation that simulates the process of revealing the contents of an attachment. Initially, the content is hidden with `MaskOverlayEffect`, Which displays a moving scan line. As the decryption progresses, `MaskOverlayEffect` gradually reveals the content from top to bottom, creating a cyberpunk-style visual effect.

The decryption animation is controlled by the property `decryptionCounter`, which takes values from 0 to 100. This value is animated with a `QPropertyAnimation`, which provides a smooth and visually appealing transition.

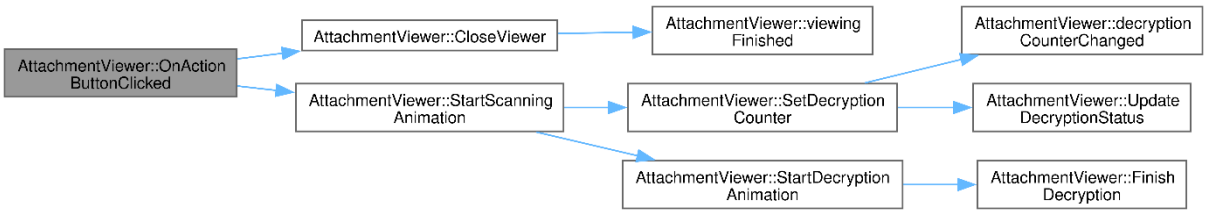

Fig. 10.5.1 Animation flow of decryption of attachments

10.5.2 Dynamic content management

`AttachmentViewer` dynamically manages content, creating and displaying the appropriate widget depending on the attachment type. The component supports images, GIFs, audio and video, using dedicated players (`ImageViewer`, `GifPlayer`, `AudioPlayer`, `VideoPlayer`) for content displaying.

After the decryption animation is complete, `AttachmentViewer` hides (and calls the destructor of) the class `MaskOverlayEffect` and displays the contents of the attachment.

The system also implements error handling, displaying an error message if an attachment fails to load or display.

10.5.3 User interaction

`AttachmentViewer` implements user interaction with an action button that initiates the scanning and decryption process. Once decryption is complete, the action button turns into a close button, allowing the user to close the attachment view.

It has also been programmed to handle mouse events, displaying a *„Click to enlarge”* label when the cursor hovers over an attachment. Clicking on an attachment displays it at full size in a separate dialog box.

10.5.4 Custom painting and cyberpunk aesthetics

`AttachmentViewer` uses custom painting to render a distinctive cyberpunk look.

Method `paintEvent` draws a frame, decorative lines, AR-style tags and status (security level, lock status).

It has also been created to dynamically adjust the style of the scrollbar area based on the attachment type, which ensures a consistent appearance of the interface.

11. Session Coordination

11.1 Session Coordinator

`SessionCoordinator` provides the point of session management in a ***Wavelength*** application, acting as a facade that simplifies interactions with the system's various components, such as creating, joining, leaving frequency, message handling, state management and configuration. This singleton component provides a consistent interface for session management, encapsulating the complexity of internal interactions.

The coordinator has connections to every instance contained in an application session, as well as to many methods from them. Below is a full graph of the Coordinator's connections with all the attachment points:

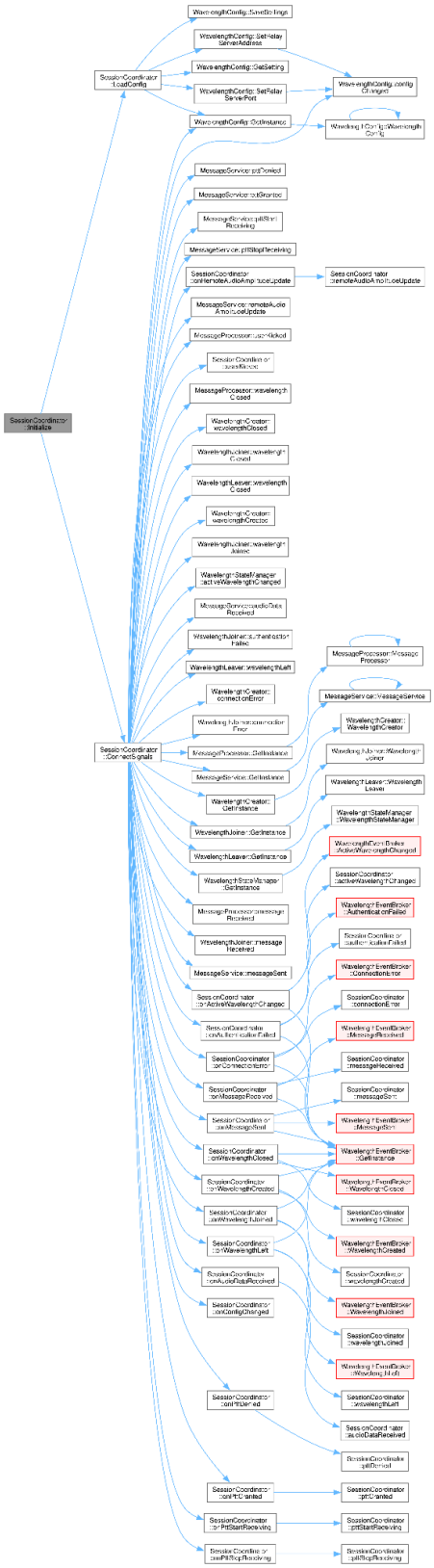

Fig. 11.1 Complete graph of attachment points of the Coordinator class (unreadable)

11.1.1 Facade architecture and task delegation

A key role of `SessionCoordinatora` is to act as a facade that simplifies interactions with various Wavelength components. Instead of directly communicating with each component, other parts of the application communicate with the `SessionCoordinator`, which delegates tasks to the appropriate components.

This architecture provides a loose link between components, making it easier to test, maintain and extend the system. `SessionCoordinator` acts as a central control point, providing consistent and controlled session management, as can be seen in the graph shown earlier (Fig. 11.1).

11.1.2 System initialization and configuration

`SessionCoordinator` is responsible for initializing the system, including loading the configuration, connecting signals between components, and setting default configuration values if the configuration file does not exist. Method `Initialize` performs these tasks, ensuring that the system is properly configured before it starts.

11.1.3 Session lifecycle management

`SessionCoordinator` manages the session lifecycle by providing methods to create, join, leave and close frequencies. These methods delegate tasks to the appropriate components:

- `WavelengthCreator`,
- `WavelengthJoiner`,
- `WavelengthLeaver`,

and then update the status of the system.

11.1.4 Handling messages and communications

`SessionCoordinator` provides methods to send text messages and files, delegating tasks to `WavelengthMessageService`. This component also receives signals about the arrival of new messages and forwards them to the

appropriate parts of the application.

11.1.5 Event signaling

`SessionCoordinator` emits a series of signals that inform about various events in the system, including:

- `wavelengthCreated`: emitted after the creation of a new frequency
- `wavelengthJoined`: emitted when attached to the frequency
- `wavelengthLeft`: emitted after leaving the frequency
- `wavelengthClosed`: emitted when the frequency is closed
- `messageReceived`: emitted when a new message is received
- `messageSent`: emitted after the message is sent
- `connectionError`: emitted when a connection error occurs
- `authenticationFailed`: emitted in case of authentication failure
- `userKicked`: emitted when the user is kicked from the frequency of the
- `activeWavelengthChanged`: emitted when the active frequency changes
- `pttGranted`: emitted when the user is given permission to transmit PTT
- `pttDenied`: emitted when the user was denied permission to transmit PTT
- `pttStartReceiving`: emitted when another user starts PTT transmission
- `pttStopReceiving`: emitted when another user ends a PTT transmission
- `audioDataReceived`: emitted upon receipt of audio data
- `remoteAudioAmplitudeUpdate`: emitted upon receipt of remote audio amplitude update

These signals allow other parts of the application to respond to changes in session state asynchronously.

11.2 Wavelength Events

Frequency management components form the foundation of the ***Wavelength*** application's communication system, allowing users to create, join and leave communication channels. Working in synergy, these singleton classes provide a consistent and controlled frequency management process.

11.2.1 Wavelength Creator

`WavelengthCreator` is responsible for initiating the process of creating a new frequency. This component connects to the relay server, sends a request to register the new frequency and manages the initial WebSocket connection until the registration result is received.

11.2.2 Wavelength Joiner

`WavelengthJoiner` is responsible for initiating the process of joining an existing frequency. This component connects to the relay server, sends an authentication/join request for the specified frequency and manages the WebSocket connection.

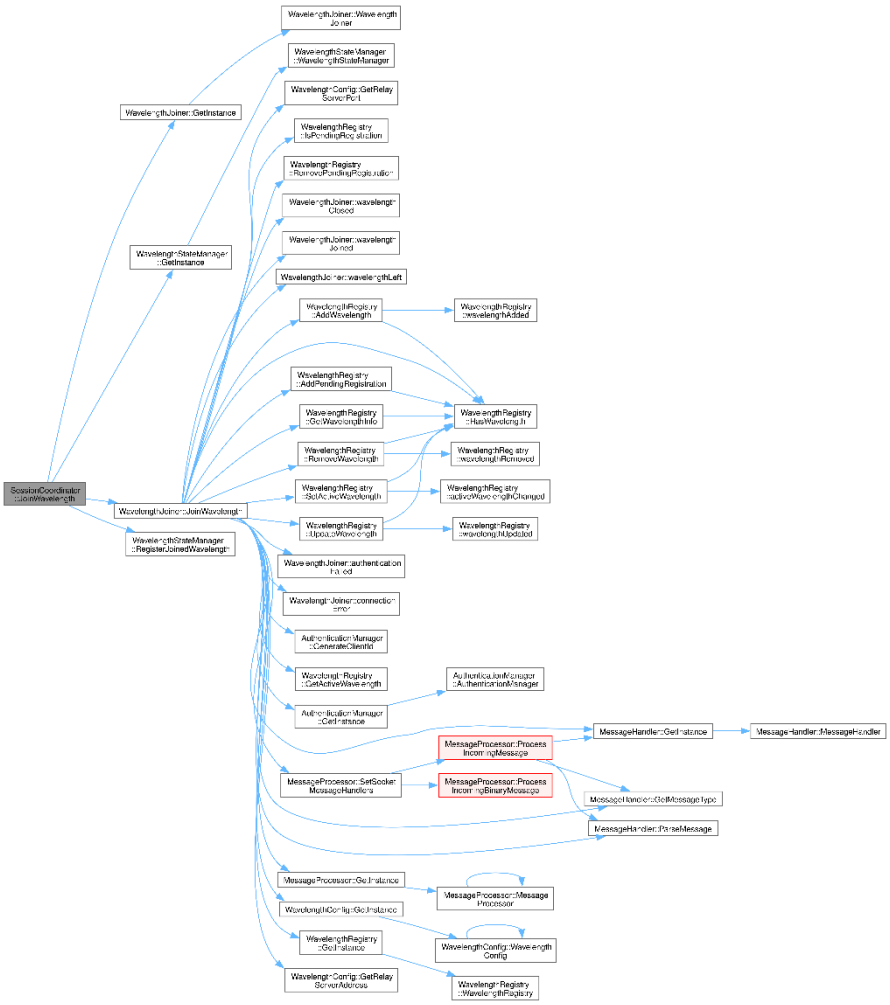

Fig. 11.2.2 Operation of connecting to an existing frequency

11.2.3 Wavelength Leaver

`WavelengthLeaver` is responsible for handling the process of leaving or closing the frequency. This component disconnects from the active frequency or closes the specified frequency if the user is the host.

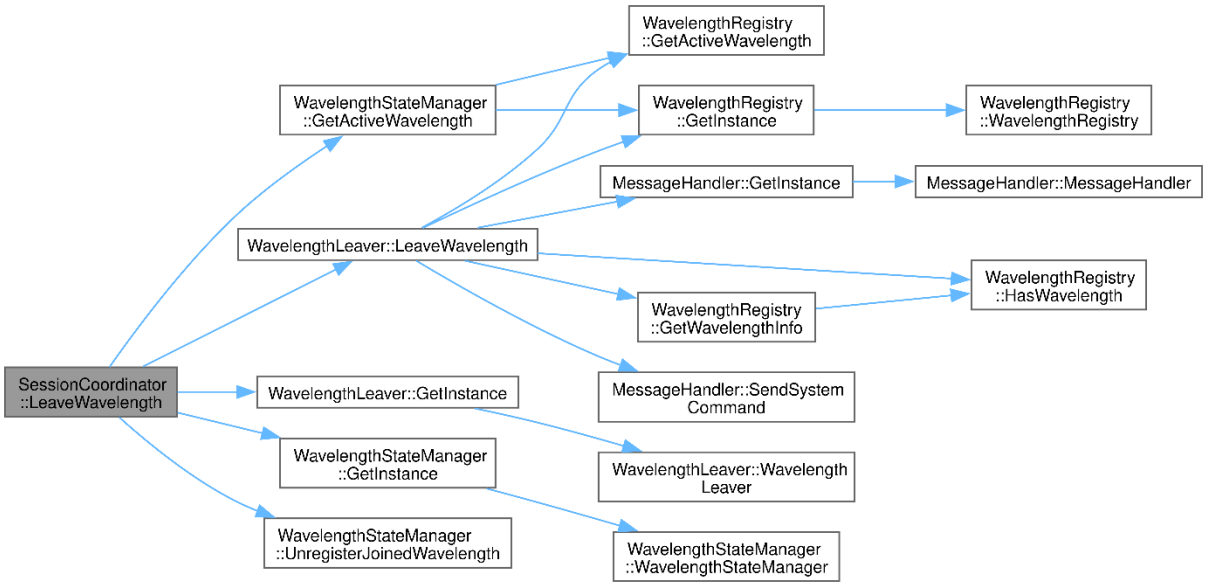

Fig. 11.2.3 Demonstration of frequency dropout operation

11.3 State Management Services

System services provide centralized state management, communication between components and event handling. They are critical to the life cycle of the entire application.

11.3.1 Event broker

`WavelengthEventBroker` implements the publish-subscribe pattern, allowing loose coupling between application components. Components can publish events
(e.g. *`wavelength_created`*, *`message_received`*) with associated data (`QVariantMap`), and other components can subscribe and respond to certain types of events without direct references to the publisher.

11.3.2 State Manager

`WavelengthStateManager` provides a centralized interface for frequency and application status management. The component tracks information about attached frequencies, connection status, session data and active frequencies. It provides convenient methods for accessing information about frequencies (information, host status, password protection, connection status) and general application status (active frequency, attached frequencies).

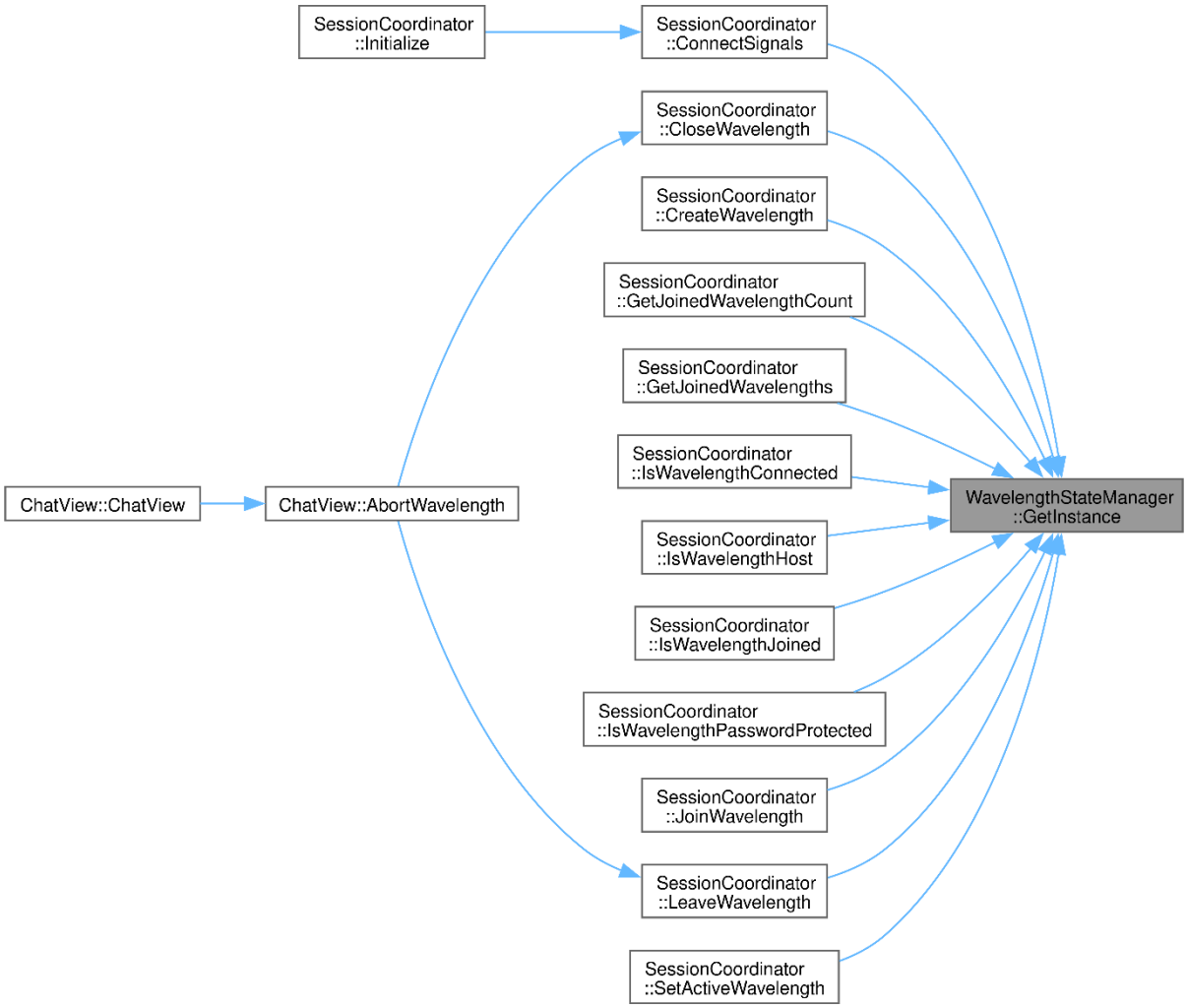

Fig. 11.3.2 Direct links of the state manager to the Session Coordinator (chapter 11.1)

12. Settings and Configuration

12.1 Settings tabs

The settings tabs in ***Wavelength*** provide the user with intuitive and organized access to configure various aspects of the program.

Each tab groups related options, allowing you to personalize the interface, network behavior and keyboard shortcuts.

12.1.1 Appearance Settings

`AppearanceSettingsWidget` allows you to configure the application's appearance, including background color, Blob color (Chapter 13), stream color, grid appearance and title bar style. Changes are immediately applied to WavelengthConfig, resulting in real-time updates to the interface. The user is able to change the colors seamlessly using the `ClickableColorPreview.`
This is a small widget that displays a color preview and allows the user to click on it, usually to open a color selection dialog box.

12.1.2 Wavelength Settings

`WavelengthSettingsWidget` allows configuration of Wavelength-specific settings, mainly the preferred starting frequency. Includes validation of the input to ensure that the frequency is within acceptable limits.

12.1.3 Network Settings

`NetworkSettingsWidget` allows you to configure advanced network settings, such as the address and port of the relay server, connection timeout, keep-alive interval and maximum number of reconnection attempts.

12.1.4 Shortcuts Settings

`ShortcutsSettingsWidget` allows you to configure keyboard shortcuts for various actions in the application. It displays a list of available actions and the shortcuts assigned to them, allowing you to modify them.

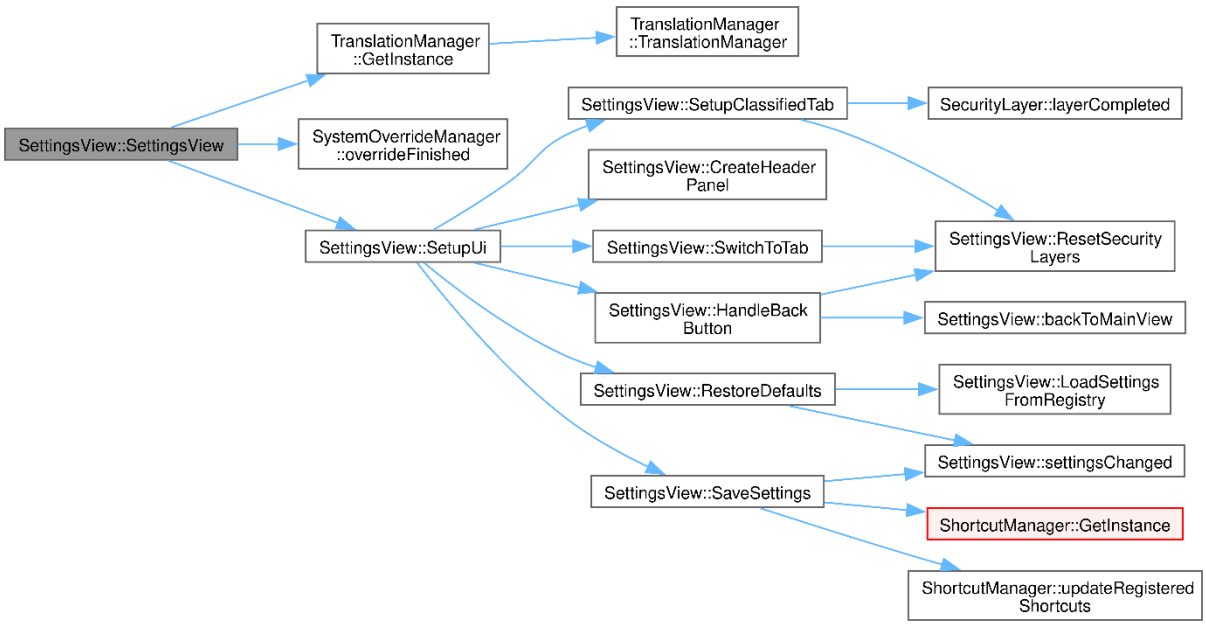

Fig. 12.1.4 Direct access to individual tabs from the application settings view panel

12.2 Classified Section

12.2.1 Security Layers

Security layers are a unique part of the settings interface in the Wavelength application, implementing a series of interactive challenges that the user must overcome to access advanced features. Each layer represents a distinct verification mechanism, simulating futuristic authentication systems.

12.2.1.1 Security base layer

`SecurityLayer` is an abstract base class for all security layers. It defines a common interface for initialization (`Initialize`) and reset (`Reset`) of layer, as well
as the layerCompleted() signal, emitted upon successful completion of the challenge

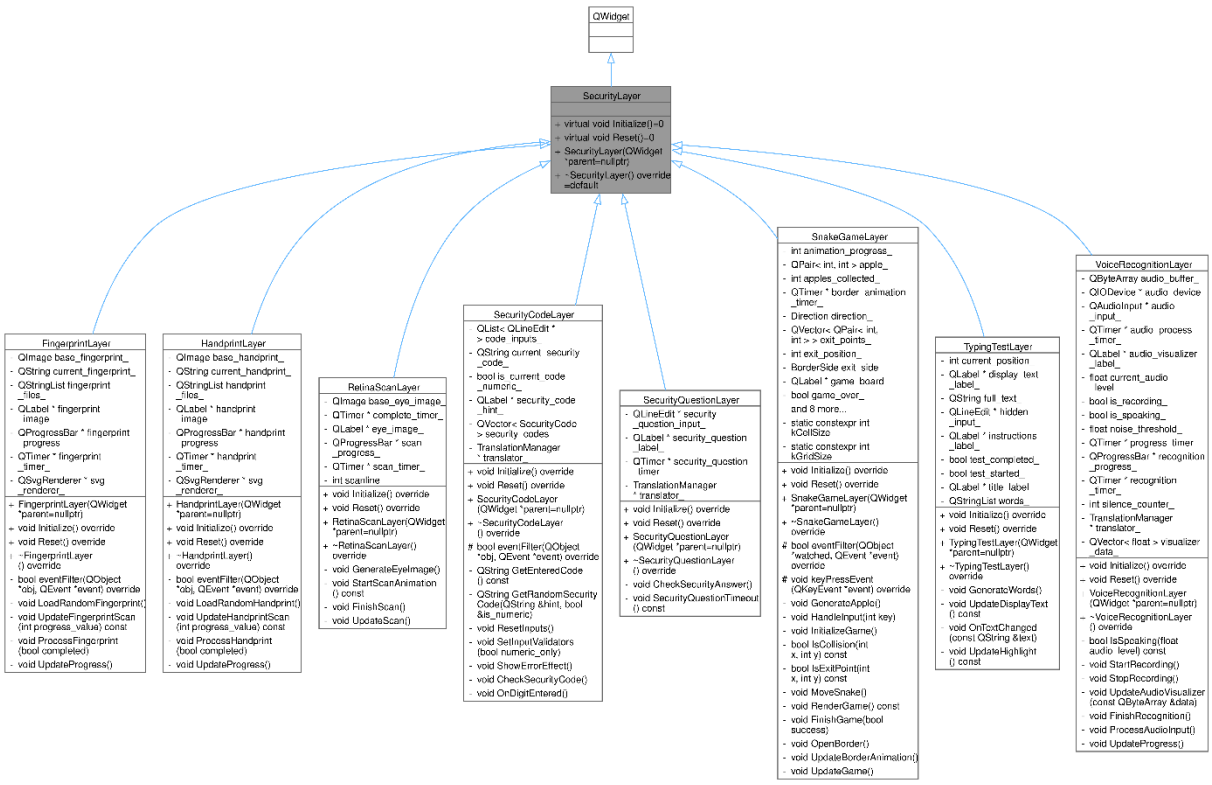

Fig. 12.2.1.1 Links all security layers to the main abstract base class

12.2.1.2 Security Code Layer

`SecurityCodeLayer` requires the user to enter a randomly generated code of 4 characters (numbers or letters). The layer displays a hint associated with the code and supports navigation between input fields using the arrow keys and the Backspace key.

12.2.1.3 Typing Test Layer

`TypingTestLayer` presents the user with a randomly generated piece of text that he or she must accurately transcribe. The layer tracks the user's progress, highlighting correctly typed characters in green and the current character with a distinctive background.

12.2.1.4 Retina Scan Layer

`RetinaScanLayer` simulates the retinal scanning process. It displays a randomly generated eye image (generated iris color based on the user's unique ID) inside a circular scanner, and an animated scanning line moves vertically across the image. When the scan is complete, the interface elements turn green.

12.2.1.5 Fingerprint Layer

`FingerprintLayer` Simulates fingerprint authentication. It displays a fingerprint image (loaded from an SVG file) and requires the user to press and hold on the image. A progress bar fills in while the mouse button is held down, and the fingerprint image visually updates during the „scan”.

12.2.1.6 Handprint Layer

`HandprintLayer` Simulates handprint authentication. It works analogously to `FingerprintLayer`, but displays a hand image and requires the user to press and hold on the hand image.

12.2.1.7 Snake Game Layer

`SnakeGameLayer` implements the classic Snake game as a challenge. The user controls the snake using the arrow keys, and the goal is to collect 4 apples.

Once the apples are collected, an exit opens on one edge of the board, and the user must lead the entire snake through the exit.

12.2.2 Special components

12.2.2.1 Floating energy sphere widget (*Joy*)

`FloatingEnergySphereWidget` is the visual centerpiece of the system's override sequence, presenting an interactive energy sphere on full screen. The component uses OpenGL to render a dynamic sphere that responds to time, sound amplitude and mouse interactions.

The component's architecture includes procedural generation of sphere geometry, dynamic vertex deformation, real-time sound visualization, simulation of impact effects, and an easter egg implementation of Konami code that triggers a destruction sequence with accompanying sphere destruction sound.

`FloatingEnergySphereWidget` is using `QOpenGLShaderProgram` to render the sphere, and `QAudioDecoder` for sound amplitude analysis.

Described more extensively earlier in Chapter 3, subsection 2.3 (3.2.3).

12.2.2.2 System Override Manager

`SystemOverrideManager` orchestrates a „system override” sequence that includes visual and system changes, creating an immersive user experience. The component manages changing the wallpaper to black, minimizing windows, displaying system notifications and presenting an interactive `FloatingEnergySphereWidget` on fullscreen.

The component architecture includes support for administrator privileges (Windows[3] only), installing keyboard and mouse hooks (Windows only) to block unwanted input, and restoring the system's state after the sequence ends. `SystemOverrideManager` is using `QSystemTrayIcon` to display notifications and `QMediaPlayer` for playing optional background audio.

13. Blob System

The BLOB system is an acronym for **Byte, Lightweight Conflict-Free Object** (pl. *Bajtowy, Lekki Obiekt Bezkonfliktowy*, since ***BLOB***), which is a nod to the user's goal of making his time in the application more pleasant.

13.1 Blob system core

13.1.1 Blob configuration

Namespace `BlobConfig` groups structures and enumerations that define the parameters of the Blob's appearance, physics simulation, resting state and general animation state. Structures `BlobParameters`, `PhysicsParameters` and `IdleParameters` encapsulate parameters that affect appearance, physical behavior and animation at rest.

[3] The mechanism found its implementation only on Windows systems, as on Linux systems it is a far more advanced process that requires interference with the entire environment, due to the lack of a direct system API.

13.1.2 Blob Animation

`BlobAnimation` is the main widget responsible for rendering and animating the dynamic Blob. This class integrates the simulation of physics (`BlobPhysics`), rendering (`BlobRenderer`), state managment (`IdleState`, `MovingState`, `ResizingState`), event handling (`BlobEventHandler`) and state transitions logic (`BlobTransitionManager`) to create a smooth and interactive animation of the entire Blob.

The component architecture is based on `QOpenGLWidget` for rendering with hardware acceleration and on a separate thread for physical calculations, which ensures smooth operation of the user interface. It also uses timers to update animation, check window position and handle resizing events.

The component implements three animation states:

1. Idle: The blob is stationary or shows subtle animations when at rest.
2. Moving: Blob responds to the movement of the window (inertia).
3. Resizing: Blob responds to the change in window size (compressibility).

`BlobAnimation` is using `BlobEventHandler` to filter and process window events, `BlobTransitionManager` for analyzing motion, calculating velocity and initiating transitions between states, `BlobPhysics` to simulate physical interactions and `BlobRenderer` for rendering the Blob and background grid.

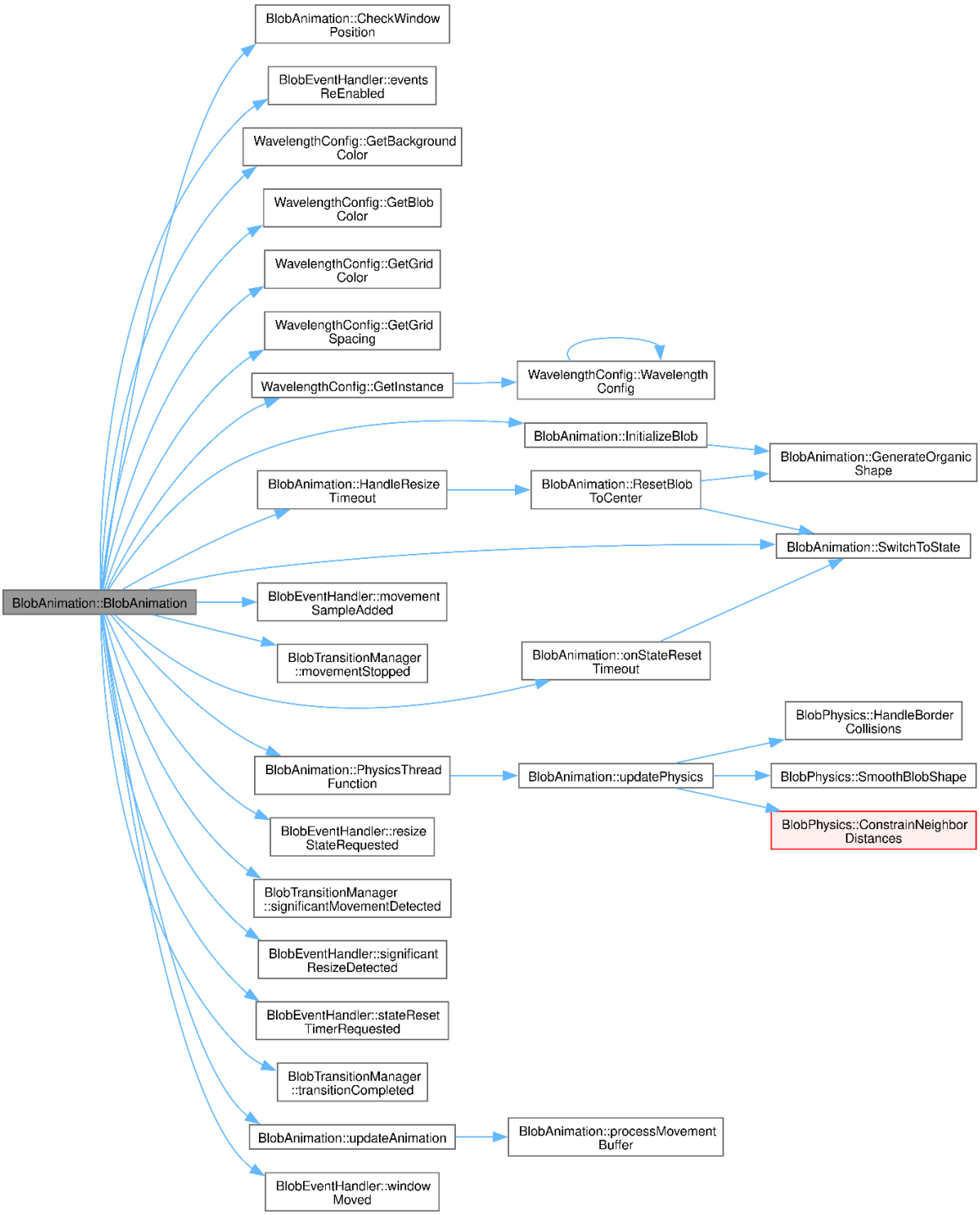

Fig. 13.1.2. Presentation of the dependencies of the different classes and their connections to the animation base class

13.1.3 Blob Physics

`BlobPhysics` is responsible for calculating physical interactions between Blob control points, including internal forces, collisions with window edges, and distance constraints between points. This class uses numerical integration methods to update the position and velocity of the control points based on the forces acting. The physics is based on fluid mechanics to make the Blob behave like a dense substance.

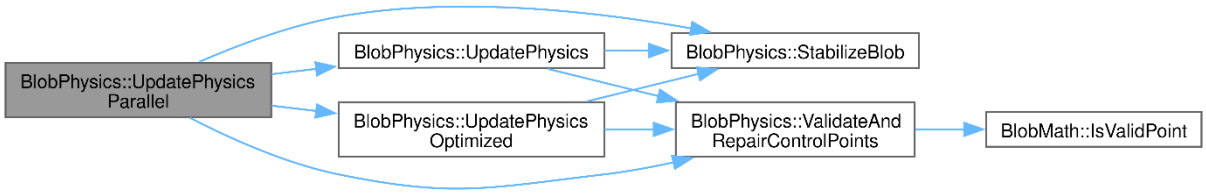

Fig. 13.1.3. Graphical visualization of the method of updating physics during a detected event

13.1.4 Blob Rendering

`BlobRenderer` is responsible for rendering the Blob, background mesh and other visual elements. This class uses OpenGL for rendering with hardware acceleration, including vertex and fragment shaders to generate dynamic visual effects.

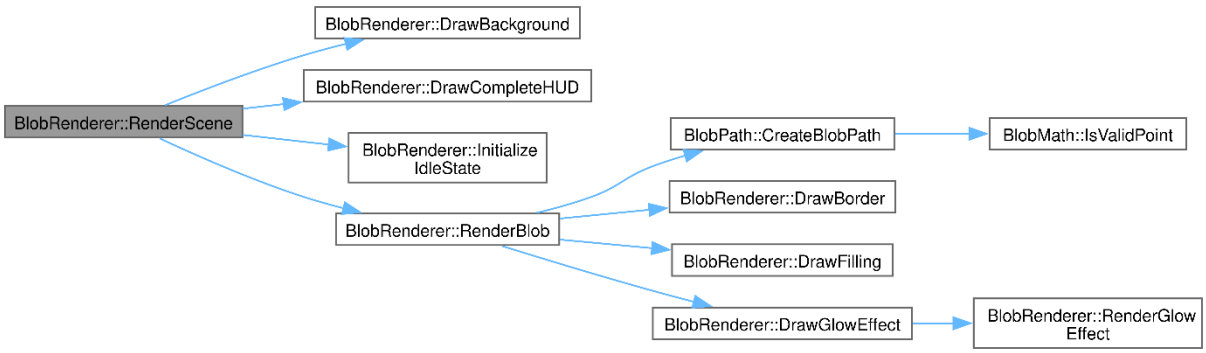

Fig. 13.1.4. Animation scene rendering flow

13.1.5 Blob event handling

`BlobEventHandler` is responsible for filtering and processing window events, such as motion, resizing and keyboard events. This class passes the events to the corresponding components (`BlobTransitionManager`, `BlobPhysics`) to update the animation status.

13.1.6 Management of state transitions

`BlobTransitionManager` is responsible for analyzing window motion, calculating velocity and initiating transitions between animation states. This class uses a motion sample buffer to smooth the input data and avoid sudden state changes.

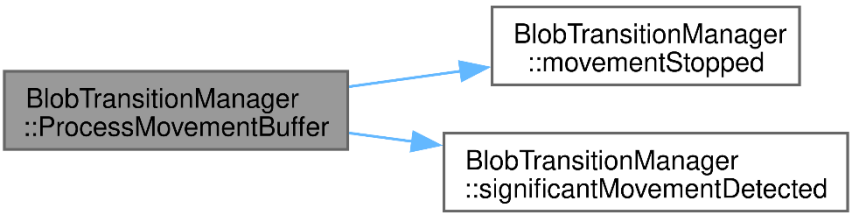

Fig. 13.1.6. Processing the movement buffer and forwarding it through signals

14. Support Systems

The supporting systems form the foundation of Wavelength's application infrastructure, providing key services such as connection management, authentication, data storage and support tools.

14.1 Wavelength Registry

`WavelengthRegistry` acts as a central repository of information about active and pending frequency connections. This class manages the list of known frequencies, stores information about their status (attached, hosted, pending registration), WebSocket connections and status (active, closed).

The component architecture is based on `QMap` to store frequency information and `QSet` to track pending registrations. `WavelengthRegistry` emits signals indicating changes in the register, allowing other components to respond to changes
in the state of connections.

Below is a full graph with connections to this „central repository”:

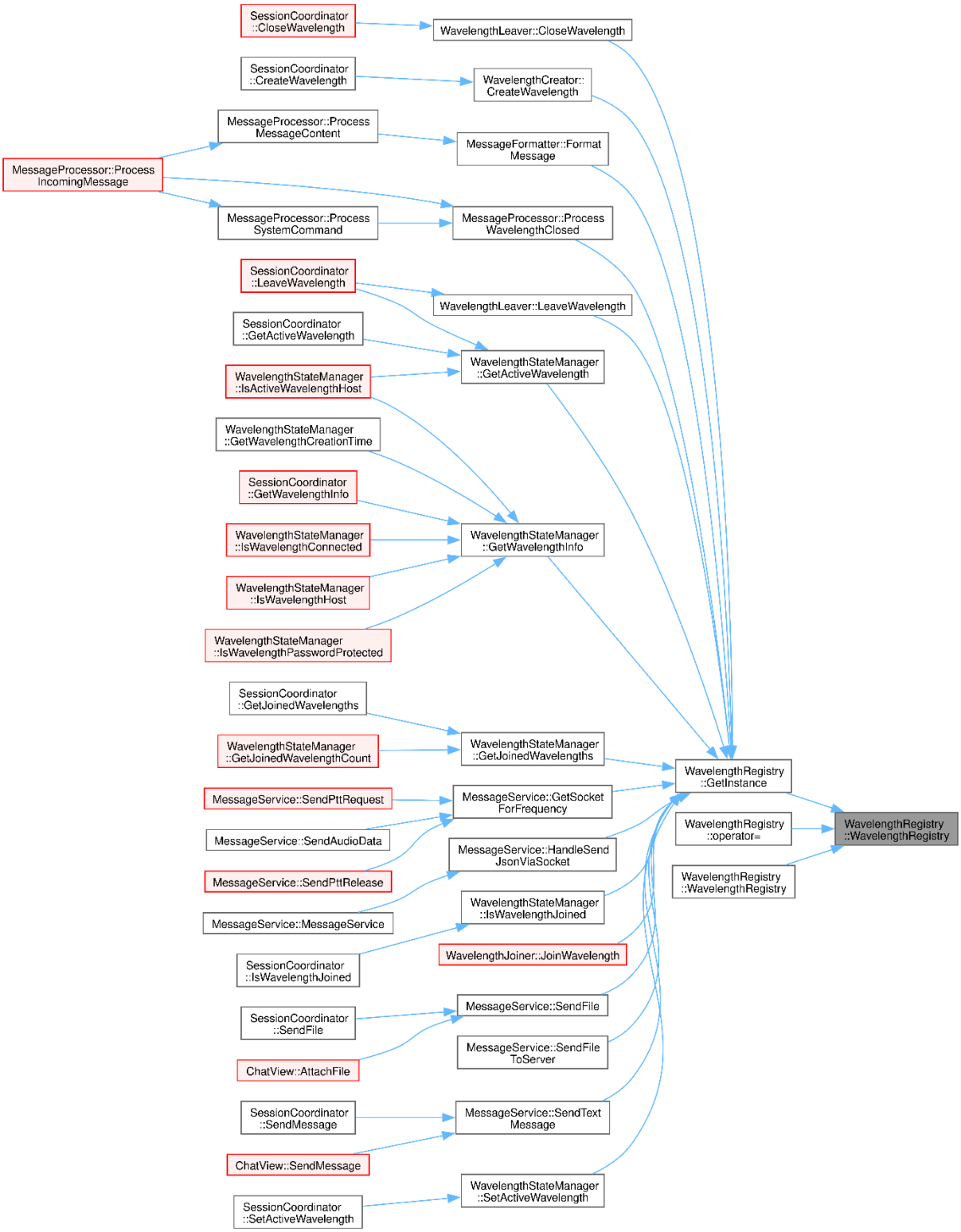

Fig. 14.1. Full call graph for the application registry

14.2 Database Manager

`DatabaseManager` manages the connection to an external `PostgreSQL` database, using the `pqxx` library. This class provides centralized access to the database, encapsulating connection details and providing a method to check the connection status.

The component's architecture is based on `std::unique_ptr` to manage the lifecycle of a database connection. `DatabaseManager` attempts to connect to the database during initialization and sets the flag `is_connected_` depending on the result.

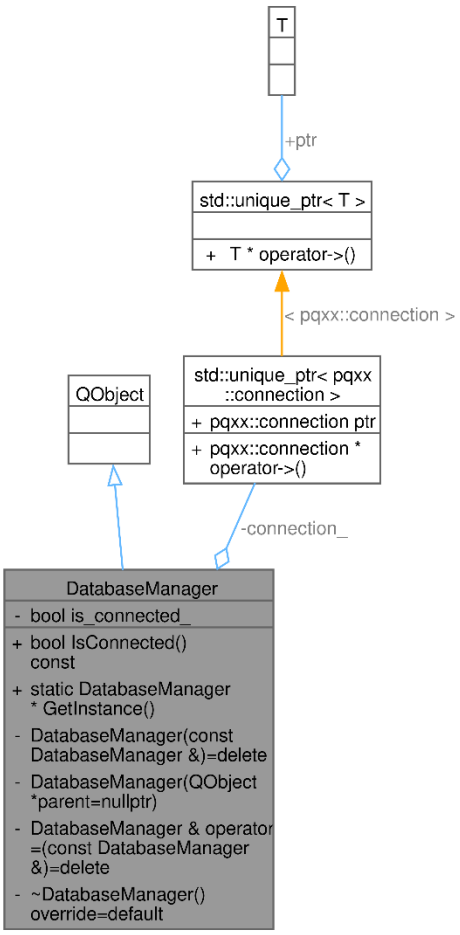

Fig. 14.2. Presentation of the database manager class

14.3 Authentication Manager

`AuthenticationManager` manages the authentication and session handling process for Wavelength frequencies. This class generates unique client IDs and session tokens, registers and verifies passwords for frequency, creates authentication response messages, stores, verifies and manages active client sessions,
and cleans up obsolete sessions.

The component's architecture is based on `QMap` to store frequency passwords and session information. `AuthenticationManager` provides methods to register passwords, verify passwords, create authentication responses, store sessions, verify sessions, deactivate sessions, and clean up outdated sessions.

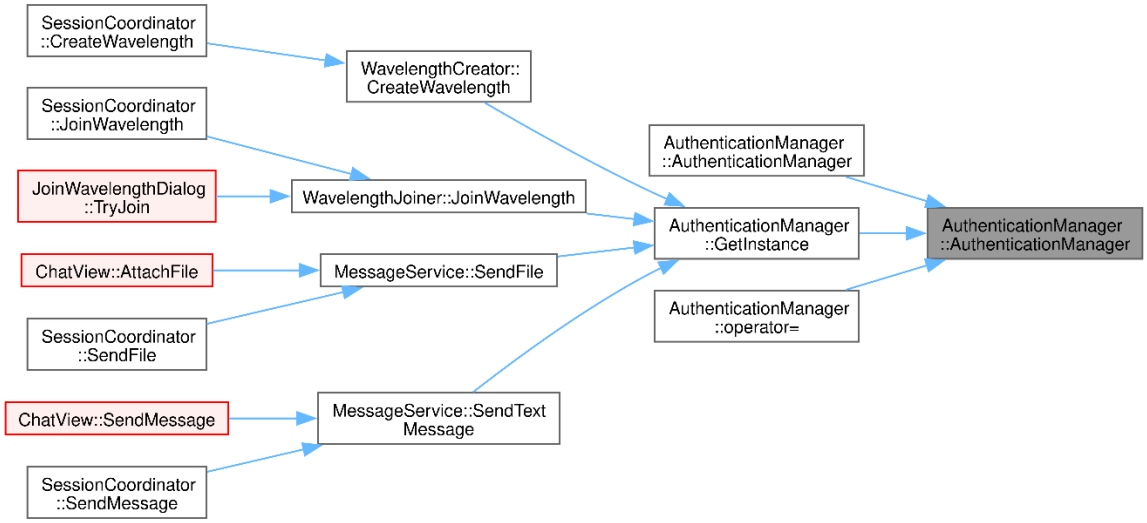

Fig. 14.3.1 Classes and their methods assisted by authentication manager

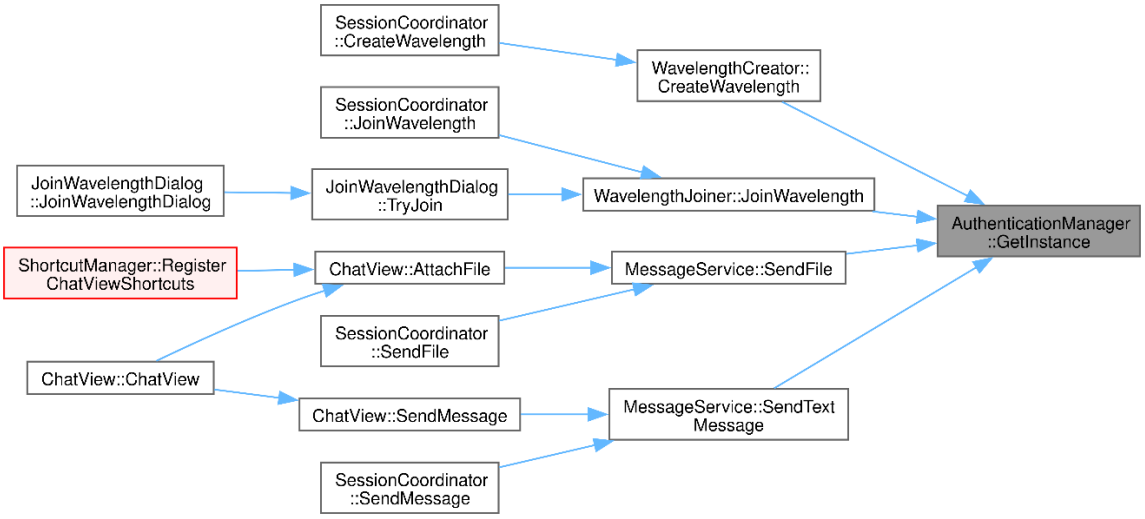

Fig. 14.3.2 Classes and their methods that access application manager instances

Summary

The End! After a long journey through the nooks and crannies of code, algorithms and interfaces, I can finally look proudly at ***Wavelength*** - an app that combines the world of communication with a cyberpunk aesthetic - which I personally find incredibly appealing.

Achieved goals

My goal was to create an application that stands out from the competition, as well as the group, not only for its functionality, but most importantly for its unique style and immersive experience. We managed to implement an advanced text and voice communication system, enriched with visual effects inspired by the Cyberpunk[4]. An attachment management system, OpenGL animations, interactive security layers - these are just some of the elements
that make up a cohesive and engaging ecosystem.

Used design patterns

During the development of Wavelength, I consciously used proven and familiar design patterns to ensure modularity, scalability and maintainability of the code. Singletons (`MessageHandler`, `AttachmentQueueManager`), Facades (`SessionCoordinator`), Factories (tworzenie widżetów załączników), Observator (`WavelengthEventBroker`) - these are just some of the tools that helped me tame the complexity of the project.

[4] Cyberpunk – futuristic game, based on a dark future, created by CD PROJEKT RED studio

Development opportunities

For me, ***Wavelength*** has become a project with huge, virtually unlimited potential for development. The possibilities are almost limitless:

- AI integration: automatic message translation, sentiment analysis.
- Augmented reality (AR): Display user information and messages in real space.
- Decentralization: implementation of P2P protocol for increased privacy and security.
- VR support: moving the chat interface to virtual reality.
- Or even migrating to the Qt6 version, for a longer life of my code.

Issues and solutions

During the development of ***Wavelength***, I encountered a mass, series of technical challenges that required a creative approach:

- OpenGL performance: shader optimization, vertex reduction, use of caching.
- Thread synchronization: Using mutexes, queues and asynchronous operations, eliminating starvation, live (and dead)-locks, etc.
- Multimedia support: The use of the FFmpeg library to decode various file formats and the myriad segmentation errors that come with it.
- Interface consistency: Designing reusable components and using consistent naming conventions.

I consider this project as one of the best I have ever created.

I will definitely develop it further to keep it as long as possible.

Links

- Program source code (open-source): Wavelength
- Documentation of the application, with the ability to manipulate graphs and complete members of each class: Documentation
- ElevenLabs