Proposed Enhancements to ScummVM: ScummHub

December 2, 2024

Queen's University
CISC322
Instructor: Prof. Bram Adams

Corey McCann 20344491 21ccm12@queensu.ca
Hayden Jenkins 20344883 21hjdm@queensu.ca
Ben Leray 202347460 21bl45@queensu.ca
Simon Nair 20297356 21scn1@queensu.ca
Jeremy Strachan 20336601 21js138@queensu.ca
Ryan Van Drunen 20331633 21rdbv@queensu.ca

1. Abstract

ScummVM is an open-source program that emulates classic games, enabling a retro gaming experience on modern platforms. The program supports over 325 adventure games, with additions to its library being consistently expanded thanks to its hardworking development team and loyal fanbase. However, the software currently lacks a modern social layer to connect players. To address this limitation, we propose ScummHub, a social hub built within ScummVM that integrates friend connections, activity and achievement sharing, screenshot uploads, and game sharing. This enhancement aims to enrich the ScummVM experience by creating a more engaged and connected community.

We'll explore two different implementations of ScummHub, the first being the more conventional client-server model where centralized servers handle requests and manage all data and activity synchronization. The second approach is a peer-to-peer model where users serve as both hosts and participants, hosting their own data for others to access through a tightly interconnected network, reducing dependency on centralized infrastructure.

Using the SAAM Analysis, we will evaluate both these approaches using key non-functional requirements and evaluate their impact on stakeholders. Finally, we'll discuss the implementation challenges, as well as the potential risks and impacts on testing and security from such an enhancement.

2. Introduction & Overview

2.1 Benefits of the Proposed Enhancement

Our proposed ScummHub enhancement modernizes the existing platform by adding new social features while still prioritizing its core gaming experience and preservation of retro games. By adding friend connections, activity sharing, and achievement tracking, we are able to create a much more interactive and community-driven experience that incorporates a social aspect around users' favorite games. Rather than only being able to play games in isolation or relying only on Scumm Forums for conversations, users can now interact intuitively with social features from within the main application.

Screenshot sharing allows players to capture their favorite moments and share them on their profiles for friends to see, fostering creativity and engagement. The game-sharing features allow users to easily download games directly through the ScummVM Social menu, eliminating the difficult process of obtaining the game from 3rd party sites or transferring them from original disks and then trying to import them into ScummVM. This makes the platform much more convenient and user-friendly, greatly reducing the barrier to entry.

Integrating a much-needed modern social layer directly into ScummVM enhances its open-source and community-driven nature while preserving its core gaming experience and increasing engagement among both new and longtime users.

2.2 Review of Concrete Architecture

The existing concrete architecture that we derived for ScummVM features a layered and modular system that incorporates interpreter, layered, and client-server architecture styles. The user mainly interacts with the User Interface Layer, which features components for the Launcher UI,

Rendering Framework, Themes, and a Common UI for managing reusable elements like widgets, icons, and dialogs.

In the backend, the Core Engine Layer is the main system that manages the virtual machine, plugins, scripting, and synchronization among other components. It interacts directly with the Game Engine Layer, which contains engines like SCI and SCUMM, which emulate the specific environments and requirements for different games. The Platform Abstraction Layer handles cross-platform compatibility by abstracting hardware-specific details such as audio, graphics, and storage through different interfaces. This allows ScummVM to run seamlessly on multiple different platforms without the need for different components or a whole new codebase. The Subsystem Layer features specific components for the Graphics Subsystem, Audio Subsystem, Device File Subsystem, and Input Output System, which are used directly by the Core Engine and Game Engines for low-level operations like game rendering, sound mixing, and other main functions. Finally, the Common Layer contains standard shared libraries and functions that are used by other components and layers to reduce code duplication and increase maintainability.

3. Proposed Enhancements

3.1 Client-Server Implementation (Alternative)

Our proposed client-server implementation for ScummHub adds an online, centralized service that allows ScummVM players to share game progress, upload achievements, and download game content directly from the cloud. This architecture offers benefits like simplified data management, secure, centralized control, and a consistent user experience. The enhancement introduces new server-side components as well as new client-side layers to connect to this centralized hub, enhancing the social aspects of the main UI.

The server-side architecture adds four key layers: Repository, User, Game, and the Controller Layers. The Repository Layer manages all persistent data related to ScummHub. It includes three core components: the Database, which stores user information, achievements, and game progress; the Object Store, which manages unstructured data like player screenshots; and the Game Store, which hosts downloadable game files for users. The User Layer manages social features, consisting of components like the User Achievements Service (for tracking achievements), User Activity Service (for monitoring user activity), User Account Service (for managing account data), and Screenshot Service (for handling user-uploaded screenshots). The Game Layer provides services for the hosted game files, with the Game Download Service allowing users to download games from the server and the Game Sharing Service supporting file-sharing between users. Finally, the Controller Layer orchestrates the interactions across these layers. It includes the User Service (for handling user requests), Friendship Service (for managing relationships and activity sharing), Game Service (for coordinating game downloads and sharing), and the Authentication Service (for secure login and transactions).

On the client side, two major additions are made: the API Access Layer and modifications to the User Interface Layer. The API Access Layer manages communication between the ScummVM client and the ScummHub server. It includes the Authentication Manager, which handles secure login and manages session tokens, and the Request Manager, which processes requests to server-side services, such as fetching user achievements or updating shared data. The Request Manager also

depends on the Common Layer for common libraries like HTTP and cURL. The API Access Layer exposes simple APIs for other client-side components to access server-side features like achievements, screenshots, and game downloads.

The User Interface Layer adds a new Social UI Component to handle the rendering and display of social features, including user activities, friends, and achievements. This component interacts with the Rendering Framework for consistent visual presentation, ensuring it matches the overall look and feel of the UI. The only new dependency added on the client side is between the Rendering Framework in the UI Layer and the API Access Layer, allowing data on the server to be fetched and updated seamlessly.

Currently, the ScummVM architecture is modular and layered, with each layer functioning mostly independently. The client-server enhancement introduces additional server-side services while expanding existing client capabilities. The API Access Layer acts as the link between the server and the current ScummVM client, integrating new services without overhauling the existing system. Social features like achievements, screenshots, and friend activity are facilitated by new server-side services, while the Social UI and API Access Layer on the client side manage the presentation and communication of this data. With this addition, ScummVM provides a richer social experience while offloading heavy processing and storage tasks to a centralized server, maintaining its lightweight nature.

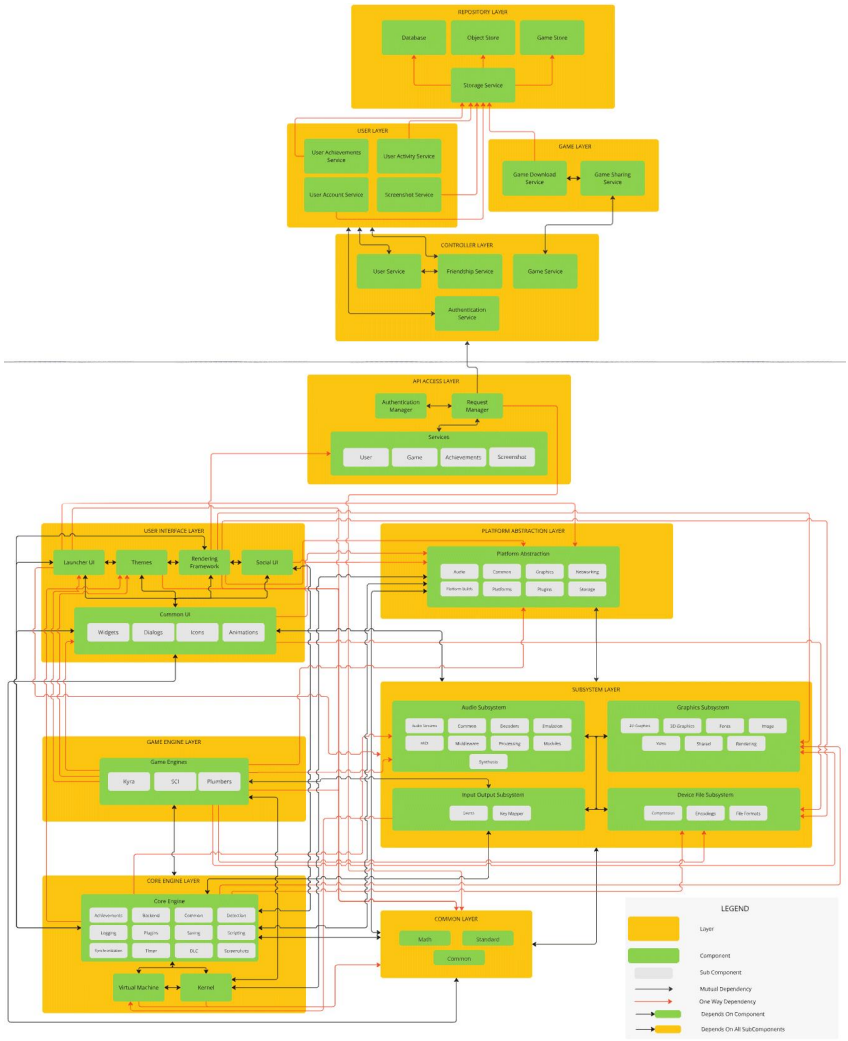

Figure 1: ScummHub Client-Server Implementation (Diagram) (Image)

3.2 Peer-To-Peer Implementation (Actual)

Our second proposed implementation involves a peer-to-peer (P2P) decentralized system that would reduce the reliance on centralized servers. By leveraging direct communication between users, the P2P model would be scalable, reduce server load, and, most importantly, be cost-effective. This implementation would require a new Communication Layer to handle communication between peers and a single backend server for peer discovery. This peer discovery service would be hosted in the cloud using the client-server style to solve one of the biggest pain points of P2P, peer discovery. Other new client components would include the Services Layer, including dedicated sub components for each of the main features such as the Activity Service, Discovery Service, Game Sharing Service, and Screenshot Service. These subcomponents would handle all of the communication, storage, and state management logic required by each feature, abstracting complex implementation details. By creating individual services that each expose their own API's other existing components could be easily modified or expanded to implement required features. The last new component would be a sub-component of the UI Layer that specifically manages the social UI and all the logic related to fetching and updating data, as well as taking screenshots and downloading games.

At the low level, architectural changes introduced by the P2P implementation include many new components and several new component dependencies. The Social UI Component within the UI Layer would connect with several of the existing components to display the entire social experience, such as user and friend activity, achievements, and screenshots. This component would rely on the Rendering Framework for visual rendering, Themes for styling, and the Common UI for shared UI elements such as widgets, icons, and animations. Integration with all of the Service Layer components would also be required. The Social UI component would also depend on the Core Engine for in-game overlays, allowing the user to take and upload screenshots as well as publish achievements. By accessing the Service Layer Components, the Social UI would facilitate P2P interactions and the entire social experience.

Service layer Components such as Activity, Achievements, Screenshots, Discovery, Friendship, and Game Sharing each depend on the Platform Abstraction Layer (PAL) for saving files and data to the user's local storage as well as the Communication Layer for communicating with peers and backend services. Some of these components, such as Screenshots, Achievements, and Activity, also depend on the Core Engine layer for managing and synchronizing gameplay-related data. These components interact with the Subcomponents within the Core Engine to facilitate these features.

The Local Storage Subcomponents within the PAL all rely on the Device Files Subsystem for access to the user's local physical storage system. This allows game files, screenshots, and other social data such as achievements, activity, and friendships to all be stored on the user's local machine rather than in the cloud. These subcomponents would handle saving and retrieval of the specific data being stored, abstracting the complexity of in-memory and object storage from other service components through simple interfaces.

The Communication Layer, which includes the Peer Communication Controller and the Backend Communication Controller, would used by each of the Service Layer Components to facilitate activity sharing, game transferring, peer discovery, and all other social features that involve

communication across the network. These Communication Controllers would depend on the HTTP and cURL libraries found in the Common Layer to facilitate different communication protocols.

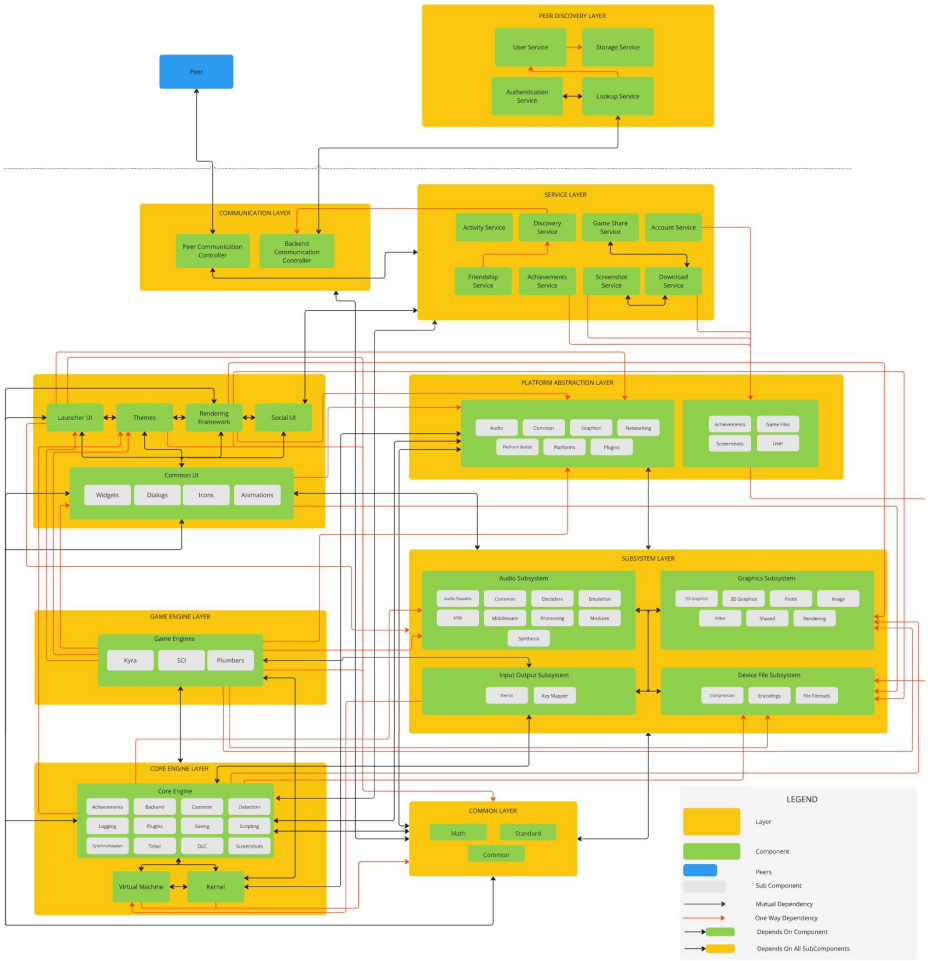

Figure 2: Peer-To-Peer Implementation (Diagram) (Image)

4. SAAM Analysis

This SAAM architecture analysis is structured by identifying the key stakeholders involved and discussing their relevant non-functional requirements (NFRs). For each NFR, we evaluate both proposed implementations, Client-Server and Peer-to-Peer, highlighting the strengths and weaknesses of each approach from the perspective of each stakeholder. For this analysis, we will consider the three major stakeholders, which are the players/users, developers/maintainers, and the platform providers.

4.1 Players/Users

Scalability: In a client-server model, scalability is more challenging compared to a peer-to-peer approach. As the user base grows, maintaining consistent performance becomes increasingly costly and difficult, as centralized servers must handle all requests and can become a source of bottlenecks, limiting the number of simultaneous users. Cloud-hosted databases also add another scalability constraint due to storage capacity limits, regional latency issues, and high operational costs. Maintaining a database to store all user records and data objects also becomes increasingly difficult with a large user base, as high traffic can lead to increased request times and decreased performance. A single database doesn't scale well, for example, a user in Asia accessing a database hosted in North America would experience much slower request speeds, highlighting the difficulty of scaling across

different regions. In contrast, the peer-to-peer model excels in scalability as it scales by distributing the workload among peers and across the network. Each new peer added to the network contributes their local resources, hosting their own data and interacting with other peers, who tend to be much closer in proximity. This decentralized approach increases the system's overall capacity and can easily handle a large user base (FS Community, n.d.).

Availability: The client-server model provides a generally high level of availability, but it can be vulnerable to server outages. A single point of failure in the centralized server may affect all users, resulting in periods of downtime. The peer-to-peer model, on the other hand, is more resilient to individual server failures. Since multiple peers serve data, if a single user goes offline, other peers can still share resources and data with all the other peers still online, maintaining the availability of services and minimizing disruptions. In this social context, if a user is offline, only their shared data and activity would be temporarily unavailable rather than critical resources or services required by the entire network, as in the client-server approach.

Responsiveness: Responsiveness is typically reliable in a client-server model for handling tasks like syncing achievements and game downloads, provided that the servers aren't overloaded and are relatively close to the user's region. However, if the servers are overloaded or are far from the user geographically, performance can degrade significantly. Addressing both of these issues often requires creating new server instances in more regions which increases complexity and is very costly. In a peer-to-peer model, responsiveness benefits greatly from peers being closer to one another as it reduces latency. However, this improvement is limited by the wide range of quality and speeds across residential networks, which can create bottlenecks. While the P2P connection can often be faster for some interactions, large data sharing is often more complicated and difficult, especially across networks limited by bandwidth. Both the server-sided and peer-to-peer approaches offer benefits and drawbacks.

Data Security and Privacy: The client-server model allows for greater control over data security as all information is stored centrally. This centralized approach allows for complex security mechanisms like encryption, firewalls, and centralized monitoring, to ensure that users' game achievements and account information are handled and managed securely. The peer-to-peer model introduces complexities regarding security since information is distributed across multiple nodes and is stored locally on a user's machine. Each peer becomes a potential vulnerability making it much harder to enforce secure and consistent data protection across the entire network. Privacy and data integrity depends on securing every node in the network, and since all peers are interconnected in the P2P model, any compromised peer is a risk to the entire network. Encryption and other security measures can still be implemented on a peer-to-peer network, the decentralized model just makes it much harder and more difficult to maintain.

4.2 Developers/Maintainers

Maintainability: The client-server model offers a greater level of maintainability. Centralized logic and data make updates, patches, and new features easier to implement and roll-out, which simplifies maintenance and ensures consistency across users. The centralized model also makes troubleshooting more straightforward. In contrast, the peer-to-peer model is much harder to maintain, as changes and updates need to be distributed to each peer individually, which can lead to

inconsistencies if users don't update. The lack of centralized control forces developers to maintain support for older software versions, as it's very likely that older versions of the software will need to interact with newer versions. This required interoperability increases the development complexity and also requires complex mechanisms to synchronize updates and ensure compatibility between peers.

Modularity & Complexity: The client-server architecture allows for modular design, making it easier to add or modify components without affecting other parts of the system. The clear separation of user-facing services from backend logic minimizes the risk of unintended consequences during changes, which helps the system evolve easily as needed. The centralized backend also reduces overall complexity compared to a distributed system, making implementation and development easier. In the peer-to-peer model, while services can also be designed modularly, integrating these components across distributed peers can lead to more complex interdependencies and an increased risk of errors. The P2P architecture requires a larger and more complex codebase for varying features like peer discovery, secure communication, local storage management, and data synchronization, all of which are required for the social network feature but aren't a part of ScummVM's primary goal. This additional functionality creates tightly coupled code and expands the responsibility of the main codebase, leading to a less modular design. It also further complicates the system and increases the risk of bugs and unintended interactions.

4.3 Platform Providers (Cloud Service Providers)

Reliability: The client-server model relies on cloud service providers for reliability. These providers offer strong infrastructure with high uptime, backups, and data protection. As long as the cloud provider remains reliable, so does the backend. However, reliance on a single provider can be risky if an outage occurs. In contrast, the peer-to-peer model reduces reliance on centralized servers by using a cloud service only for peer discovery, while direct interactions happen between peers. This setup minimizes dependency on central infrastructure but introduces variability, as it depends on the availability of individual peers, which can affect overall reliability.

Cost-effectiveness: A major drawback of the client-server model is the high cost of maintaining centralized servers, which increases as the user base grows due to increased storage and bandwidth requirements. Features like real-time syncing and game sharing can significantly drive up bandwidth usage, increasing operational expenses. In contrast, the peer-to-peer model is more cost-effective as it distributes resources among users, reducing the financial burden of maintaining centralized servers, with only minimal costs associated with the lightweight peer discovery service.

The client-server model has several advantages that make it attractive for a social hub, particularly in terms of ease of implementation, modularity, and maintainability. Centralized management simplifies the addition of new features, ensures consistency across users, and facilitates secure data handling. However, it comes with significant drawbacks, such as scalability, availability, and the high costs of cloud infrastructure. These limitations become particularly evident when considering the long-term growth of social platforms and the adverse consequences within a system such as ScummVM.

We believe that the peer-to-peer (P2P) model is the most ideal implementation of our social

enhancement because of its alignment with the project's open-source nature, cost-effectiveness, scalability, and reliability. ScummVM's open-source philosophy succeeded because of its decentralized approach and user-driven contributions. Unlike the client-server approach, which would require centralized services and a significant financial investment in cloud resources, the P2P approach would offload storage and bandwidth requirements to users, reducing almost all operational costs. This is extremely important as ScummVM lacks a revenue stream to support expensive cloud infrastructure.

The P2P model would also resolve issues related to regional scalability and reliability, which exist in the client-server model. As discussed, a centralized server would have greater latency for users in other geographic areas, requiring server replicas and load balancing, further increasing cloud costs. The P2P approach would scale as new users join and would be entirely dependent on local peer resources rather than a centralized server. Distributing storage across the P2P network would also avoid the complexity of managing large centralized databases, which need complex mechanisms to handle replication and synchronization and often don't handle simultaneous connections well.

To enhance the usability of the social hub a small and lightweight discovery service would need to be hosted in the cloud to facilitate peer discovery and allow users to find each other and send friend requests without needing to know how to directly connect to one another. This would completely resolve one of the biggest downsides of the P2P model by simplifying peer discovery and making peer connections seamless.

Although the P2P model has challenges surrounding version interoperability, secure communications, and updates, many of these problems can be mitigated through version control, encryption, and abstracted interfaces. With a thoughtful design and solid implementation, many of these drawbacks can be minimized and can, in turn, result in significant benefits. In conclusion, the P2P approach is the best implementation as it aligns with ScummVM's existing philosophy, reduces costs, scales effectively across regions, and provides the most resilient foundation for a social hub.

5. Enhancement Impact on ScummVM

The peer-to-peer implementation of ScummHub introduces significant enhancements to ScummVM's architecture, enabling a cost-effective and decentralized solution for social features. By reducing reliance on centralized servers and leveraging direct communication between peers, ScummVM can evolve to support new functionalities such as activity sharing, game sharing, and screenshot management. This enhancement not only modernizes the system but also aligns with the goals of reducing server costs and improving user experience through social engagement.

5.1 Impact on High-Level Architecture

At a high level, the architecture now includes a new Communication Layer to manage peer-to-peer interactions and backend peer discovery. This layer consists of the Peer Communication Controller and Backend Communication Controller, which facilitate communication protocols. The Service Layer has been expanded with modular components such as the Activity Service, Discovery Service, Screenshot Service, and Game Sharing Service, each exposing APIs for feature implementations. Additionally, a new Social UI component in the UI Layer is introduced to handle social interactions, such as displaying user activity, achievements, and screenshots. This UI relies on

existing components like the Rendering Framework, Themes, and Common UI for styling and visual rendering, ensuring seamless integration with the user experience.

5.2 Impact on Low-Level Architecture

At a lower level, the new Service Layer components depend on the Communication Layer for peer-to-peer data sharing and the Platform Abstraction Layer for storing user data locally. The Core Engine connects with the in-game social overlays and synchronizes gameplay-related data such as activity, screenshots, and achievements. The Social UI interacts directly with the Service Layer components to manage social features while depending on the Core Engine for in-game functionality. The Device File Subsystem has also been extended to support the storage of user data, screenshots, and game files on the user's local system, helping to abstract the complexity of storage operations across different platforms through clean interfaces.

5.3 Impacts on Directories and Files in Subsystems

These architecture changes introduced several new directories and files. The new Communication Layer would most likely be added within a *peer_controller.cpp,* and the *backend_controller.cpp* would handle communication with the peer discovery service. Both of these new files should be contained in a new folder called *controllers* within the *social* folder at the root of the project.

In the Service Layer, files such as *activity_service.cpp* and *screenshot_service.cpp* would need to be created and would contain modularized features like activity tracking and screenshot management. These services would exist within the *social* folder and would be used directly by the Core Engine through files such as *metaengine.cpp*.

The new Social UI component would be implemented in *social_ui.cpp*, which would be integrated with existing UI components and logic in the *gui* directory. The Social UI would also need to use files and logic within the *gui/themes* directory as well as files and components in the *imgui* directory and *display_manager.cpp* from the backend directory, which correspond to components in the Rendering Framework.

The Core Engine would also need to be updated with logic to support screenshot handling, activity, and game achievements affecting files such as *thumbnail.cpp, base/main.cpp,* and *acheivements.cpp*. Files in the *backends/network* directory would also need to be updated to facilitate the communication implementations in the *peer_controller.cpp,* and *backend_controller.cpp*.

Platform-specific files and interfaces in *backends/platform/sdl* and *backends/platform/posix* would also need to be significantly overhauled to add support for local user and game data storage. One implementation might involve introducing a *local_storage.cpp* file to act as a controller to abstract the management of local data. Implementing the social feature into ScummVM's existing architecture would require significant changes to many files and directories. However, a lot of the functionality required by many features already exists in some form in the codebase, making implementation fairly straightforward, the only difficult feature would be the actual P2P communication protocol and the local data storage abstraction.

6. Use Cases

6.1 Screenshot Upload Use Case

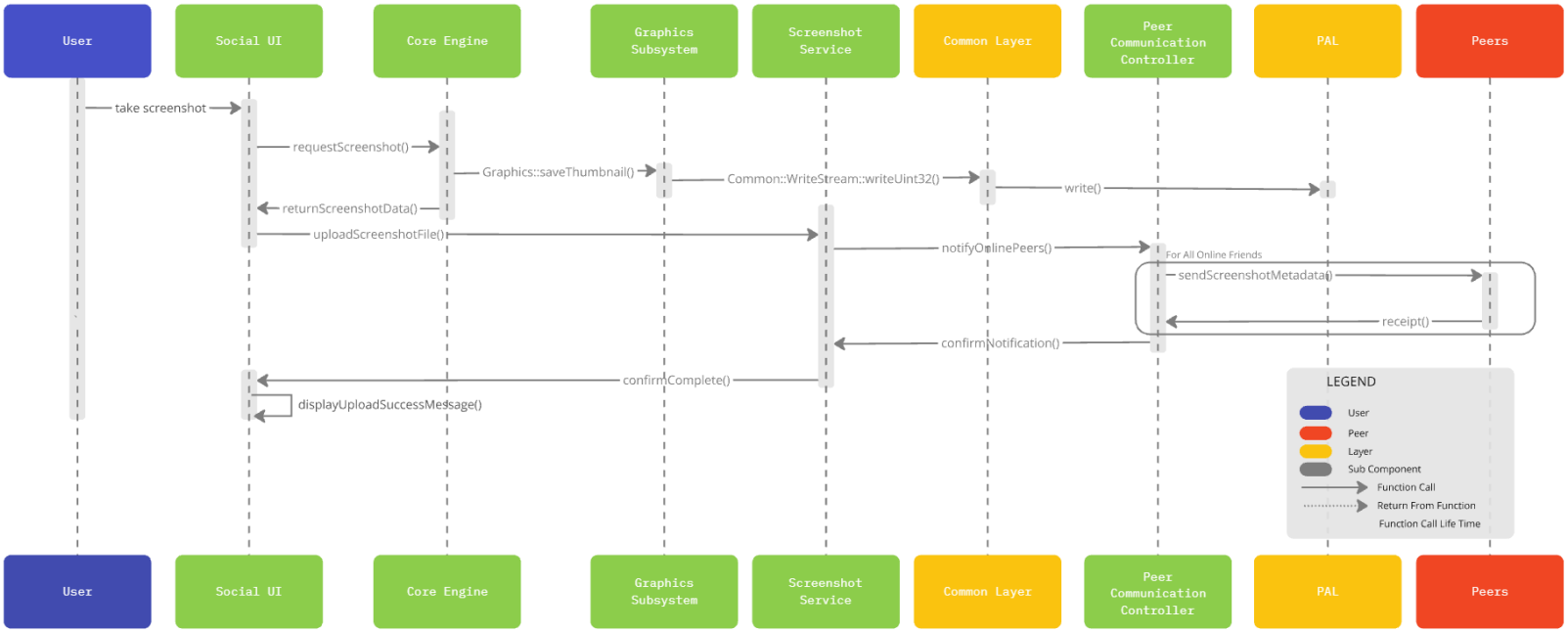

Figure 3: Screenshot Upload Use Case (Diagram) (Image)

The screenshot upload process demonstrates how the peer-to-peer architecture handles user-generated content sharing. When a user takes a screenshot, the interaction flows through several key architectural components. The Social UI component, which is part of the UI Layer, first interacts with the Core Engine to capture the screenshot data during gameplay. To take the actual screenshot, the Core Engine uses the Graphics subsystem to get a capture of the current games frame and then uses the Common Layer and PAL to write to local storage. This interaction showcases the integration between the new social features and the existing Game Engine functionality.

The Screenshot Service, a dedicated component within the Service Layer, then manages the screenshot data. This service performs two critical functions in the peer-to-peer system: local storage management and peer notifications. For storage, the Screenshot Service interacts with the Platform Abstraction Layer to ensure the screenshot entry and its metadata are stored locally on the user's device along with the actual image file, eliminating the need for cloud storage servers.

For sharing functionality, the Screenshot Service uses the Peer Communication Controller within the Communication Layer. This controller handles the complex task of notifying online friends about the new screenshot. Rather than uploading the actual screenshot to a central server, the system only shares metadata with peers, who can then request the screenshot directly from the user's machine when they want to view it. This approach significantly reduces bandwidth requirements and eliminates the need for cloud servers.

6.2 Game Download Use Case

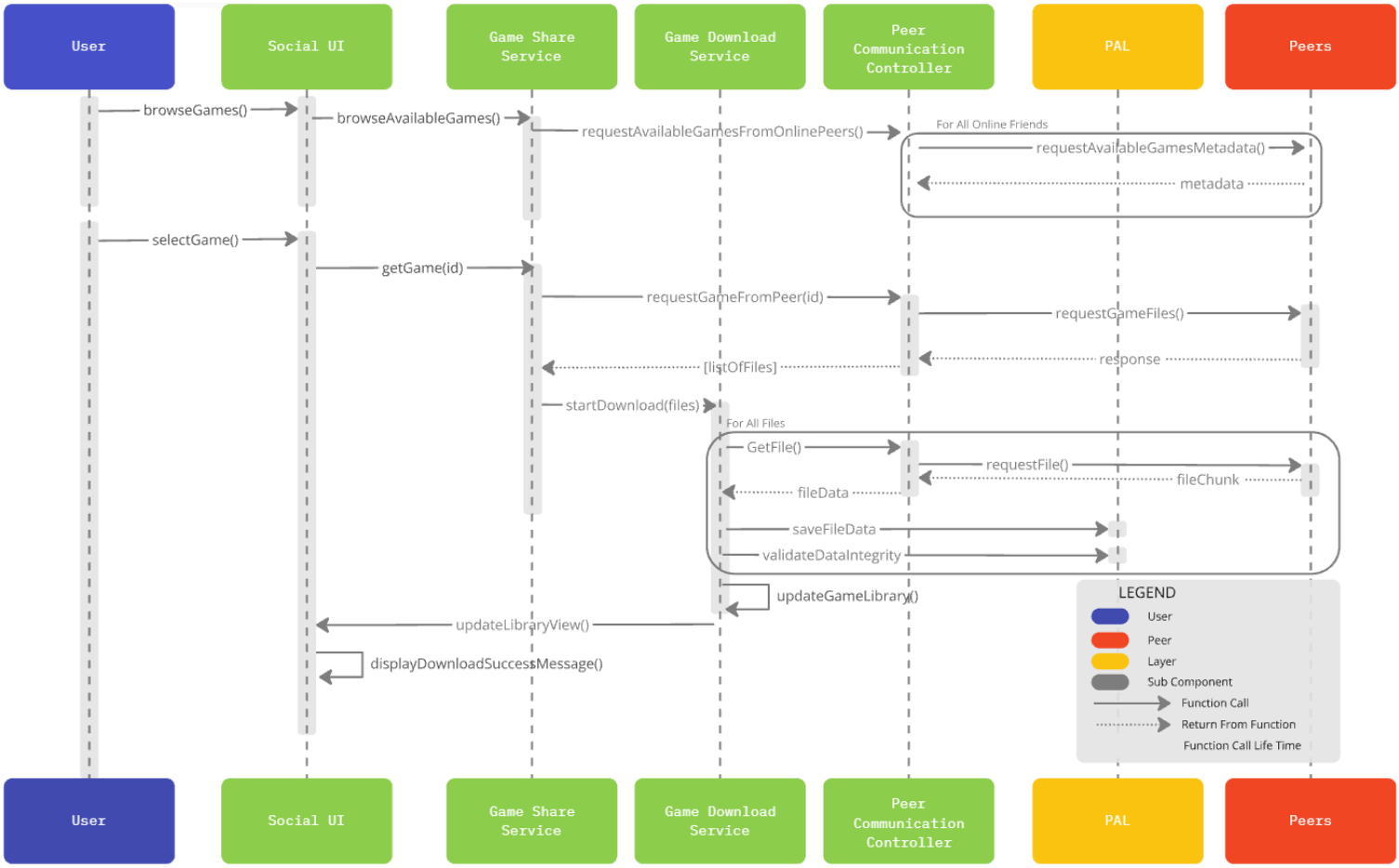

Figure 4: Game Download Use Case (Diagram) (Image)

The game download process showcases how the peer-to-peer architecture handles large file transfers between users. When a user browses the Social UI for available games from their friends, a request is sent to the Game Share Service, which uses the Peer Communication Controller to get all available shared game metadata from online friends.

When a user initiates the download of a specific game, the Social UI once again triggers a request through the Peer Communication Controller. This request is responsible for getting a list of all the required files needed for a specific game from the peer that is hosting the game. This list of files, as well as the metadata related to the peer and the game being downloaded, is then passed to the Game Download Service, which is responsible for downloading all of the game files to the user's local device. This service manages the connection with the peer and writes file chunks to the device using the Platform Abstraction Layer, which has an interface for interacting with the user's device filesystem. As files are saved, the service also validates the file's integrity to ensure that no packets or data were lost or corrupted during the network transfer.

Once the game has been downloaded, the Game Download Service is then responsible for updating the user's local game library and displaying a success message. The game library is what maintains the user's local game catalog and allows game information and data to be shared with friends for game sharing and other social features.

7. Testing

Implementing ScummHub requires a comprehensive and continuous testing strategy to ensure it meets industry standards, government policies, and user expectations. This approach integrates rigorous testing into all phases of the development process, ensuring proper functionality and stable performance from the development phase all the way through to the public release.

Continuous testing throughout the development phase is essential for maintaining software quality at early stages of development. This approach allows for early feedback on the state of the entire system, leading to early identification of bugs and allows developers to address the issues sooner, reducing cost and improving development efficiency. Regular testing throughout the development process also ensures that each new iteration is validated to ensure stability, maintain system performance, and reduce the chance of critical errors being introduced in later development phases. By including integration tests in the testing strategy, we would be able to ensure seamless interactions between ScummHub's new features and the existing components and features, validating that all systems work correctly and cohesively together.

Automated testing at the deployment stage would provide a final validation step before updates are published. These tests would validate the system's stability in a production-like environment before deployment. This is especially important considering the many different devices and platforms supported by ScummVM. Mocking and using simulation tools to test the peer-to-peer network would also enable replication of complex and difficult-to-test scenarios such as node failures and high-traffic conditions, ensuring that the P2P system is able to perform well under real-world production conditions.

An in-depth and sophisticated testing process that follows these key requirements would help maintain quality code, ensure proper coding standards, and reduce the likelihood of critical issues reaching end users in production settings.

8. Potential Risks

The implementation of ScummHub has several associated risks in terms of cybersecurity, copyright infringement, preventing the unauthorized distribution of games, and ensuring application maintainability.

8.1 Cyber-Security Risks

First and foremost, cyber-security concerns regarding the interactions between users are large potential risks. With potentially millions of users, ScummHub would contain large amounts of sensitive data such as emails, passwords, and personal information. Protecting this data from breaches is critical, as a leak would damage user trust and could potentially leak sensitive data. A knowledgeable development team and robust infrastructure would be essential for reducing unauthorized access to sensitive information. The sharing of viruses and malware is also an issue with any online community, as cyber-criminals can inject malicious code into almost anything. To safeguard all aspects of the platform, from mods to in-game communication, a strong antivirus and threat detection model would be required to protect players and their devices. Implementing strong authentication measures such as

two-factor authentication and monitoring locally saved data and peer requests would help mitigate threats but wouldn't be 100% foolproof.

8.2 Copyright Infringement & Content Moderation

With the ability to share games and other content through ScummHub, the issue of copyright infringement becomes more prevalent. The ScummVM team would need to ensure that games or content uploaded to the platform don't infringe on the intellectual property rights of others. Mods, skins, music, and the games themselves may inadvertently violate copyright laws. Monitoring this risk is complex, given the volume of uploaded content. Bad actors may try to exploit other developers' property rights by creating near-identical games or reselling low-quality assets. The challenge then becomes balancing open-platform freedom with protecting the legitimate property rights of developers. The moderation of uploaded content is also another similar and complex issue. With content-sharing platforms, there is always a risk of unauthorized and even illegal content being uploaded and shared with others. Monitoring upload sizes, restricting file types, scanning uploaded data, and implementing a robust reporting mechanism for users to report inappropriate content would help limit the distribution of unauthorized or inappropriate material.

8.3 Enforcing Ownership of Games

The challenge of preventing the unauthorized distribution of video games is a constant battle for the gaming industry. Skilled hackers and exploiters will constantly find new ways to crack games and share copies illegally. Stolen game keys and hacked accounts can be sold on third-party sites to undermine the revenue of legitimate developers and game studios. Identifying pirated copies of games and implementing strong game key checking would be needed to limit the unauthorized copying and distribution of paid games. Additionally, banning users involved in such activities, along with their network IPs or applying hardware bans, could act as a strong deterrent and prevent repeat offenses.

8.4 Maintainability

Maintaining a peer-to-peer style system is very difficult. New features and changes need to be tested extensively to limit potential impacts on players and users. Updates to ScummHub would need to be tested on all supported platforms and operating systems to ensure stability across all supported devices. ScummHub would also need to be backwards compatible to allow users on different build versions to communicate with each other. These challenges, combined with traditional software maintenance issues, could result in very lengthy and expensive maintenance costs, straining the development team and their limited resources, slowing down future releases and features.

9. Conclusion and Lessons Learned

The addition of ScummHub would undoubtedly enhance the overall user experience of ScummVM by bringing an interactive and engaging way to interact with other community members. The proposed peer-to-peer implementation of ScummHub represents a transformative step forward for ScummVM, preserving its open-source nature while addressing key challenges posed by the traditional client-server model. By decentralizing data storage and distributing workloads among peers, the P2P approach would offer significant advantages in scalability, cost-effectiveness, and resilience.

At both high and low levels, the architecture enhancements introduce critical new components such as the Communications Layer, Service Layer, and Social UI component. These additions facilitate peer-to-peer interaction, enhancing the social experience for users. However, the P2P model may face challenges in areas such as cyber-security, copyrights, and content moderation. Fortunately, many of these issues can be resolved with a proactive development team with modern development tools.

Ultimately, ScummHub modernized ScummVM, strengthening its position as a leader in modernizing the retro gaming industry. The peer-to-peer implementation of ScummHub not only addresses current limitations but also lays a solid foundation for future feature expansions, ensuring a cost-effective and user-driven evolution of the platform.

10. Naming Conventions

P2P = Peer to Peer
PAL = Platform Abstraction Layer
UI = User Interface
SAAM = Software Architecture Analysis Method
API = Application Programming Interface
HTTP = Hypertext Transfer Protocol
SCI = Sierra Creative Interpreter
SCUMM = Script Creation Utility for Maniac Mansion
cURL = Client for URL

11. References

"SAAM: A Method for Analyzing the Properties of Software Architectures", Kazman et al., ICSE 1994 (https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.8786&rep=rep1&type=pdf)

FS Community. (n.d.). ***Client-Server vs. Peer-to-Peer Networks*****. FS Community.** Retrieved November 26, 2024, from https://community.fs.com/article/client-server-vs-peer-to-peer-networks.html