CISC 322: Software Architecture

Gami: A Minigame Feature for Jami

Group 21

https://cisc322.github.io/Jami-ing-Out/

April 09, 2021

Authors:
Bruce Chidley - 20104323, 17bjc4@queensu.ca
Ben Minor- 20101983, ben.minor@queensu.ca
Teodor Ilie - 20100698, 17ti5@queensu.ca
Renee Tibando - 20113399, 17rat3@queensu.ca
Zack Urbaniak - 20124496, 18zeu@queensu.ca
Alice Petrov - 20111076, 17ap87@queensu.ca

Abstract

The following report proposes and examines the conceptual implementation of "Gami", a new multiplayer minigame feature in Jami. Jami is a distributed and open source communication platform developed by Savoir-faire Linux which supports instant messaging, call, and video chat features. The addition of a minigame feature will have both positive social implications and encourage users to remain active on the platform. Like the conceptual architecture of Jami, Gami's architecture will follow a Model View Controller design pattern driven by a peer-to-peer communication network, thus introducing no new major dependencies between subsystems and minimizing the overhead of its implementation. Gami introduces three new major components: The Game Engine, the Data Manager, and the OS Interface, which are broken down and discussed in both high level and low level detail. To illustrate the impact of its implementation, we discuss testing features, NFRs, and the associated risks and limitations of Gami. We further our discussion by reviewing two conceptual use cases involving Gami: starting a game and taking a turn. We then evaluate the proposed Gami architecture via a SEI SAAM Architectural Analysis, using identified stakeholders and NFRs to propose two alternative implementation approaches. In the final sections, we discuss the derivation process and lessons learned.

Contents

1. Introduction

In previous reports, we derived the conceptual and concrete architectures of Jami: a distributed peer-to-peer and SIP based communication platform that runs entirely on individual devices which was developed and maintained by the Montreal based Savoir-faire Linux. Following our previous analysis, we propose the conceptual implementation of a new feature in Jami: “Gami”. Gami is a minigame style addition to the Jami messenger feature allowing two players to play games over the peer-to-peer network. The addition of a minigame feature will have both positive social implications and encourage users to remain active on the platform.

This report will thoroughly discuss the details of Gami’s implementation, beginning with the architectural impact followed by a breakdown of the high and low level implementation details. Gami introduces three new major components: The Game Engine, the Data Manager, and the OS Interface, which follow an MVC (Model View Controller) design pattern much like Jami itself. All communication between players is done via the peer-to-peer OpenDHT network. Gami is closely integrated with Jami’s original architecture, meaning we minimize any impact on Jami’s original architecture.

To further our discussion of Gami’s impact on Jami by discussing testing, NFRs, and the associated risks and limitations of Gami. We illustrate the implementation of Gami using two new use cases: starting a game and taking a turn. We follow with an SEI SAAM Architectural Analysis to evaluate our proposal, using identified stakeholders and NFRs to propose two alternative implementation approaches. We conclude with a discussion of the derivation process and lessons learned.

2. Overview of Proposed Feature

Our proposed enhancement to Jami is to implement a feature where two users gain the functionality of playing a correspondence minigame together, similar to the game feature on iMessage. We have decided to call this new feature “Gami”. Gami will allow a user to initiate a correspondence minigame with another user directly in their messaging chatroom. The other user accepts their game invitation and the two users make moves with no time constraint, sending the result of their move to each other with every turn. The scores and other turn results will be stored on the DHT (central server), so each user will have to upload the result of their turn and also draw from the server before their new turn commences. The internal calculations for the game, including the game rules and mechanics, will be stored locally on the user’s device. Video games are becoming increasingly prevalent as younger generations are becoming more and more involved with different technologies. Especially during the pandemic, people are spending much more time indoors than before, and are looking for things to do. Adding a minigame feature to Jami not only has positive social implications, meaning that users will be doing activities other than strictly messaging or calling each other, but it provides another game that people can play during the pandemic. For this reason, we believe that adding Gami will bring new users to the platform, and will cause users to remain active on Jami for much longer than they would otherwise.

3. Current State

Jami's main sections are the Client, which interprets user's inputs and is responsible for display, the LRC, which specializes in data movement and flow across Jami, the Daemon, which interprets user actions, calculates results, houses most other internal functions and communicates with servers, and the servers, which store global information that many devices can access. The current state of Jami does not have any subsystems in place relating to games or any type of complex simulation aspects separate from messaging and calling. Jami's Client and LRC have many functions in place necessary for games to be played relating to the flow of information. Only minor tweaks would need to be made for strictly the flow of information, such as the handling of game-specific user inputs. However, the Client does not currently have a system in place for displaying complex multimedia, which would be needed for games to run at a high quality. Finally, the DHT does not have game capabilities and would need to be slightly modified to be able to hold new types of information relating to each user's turn.

4. Architectural Impact

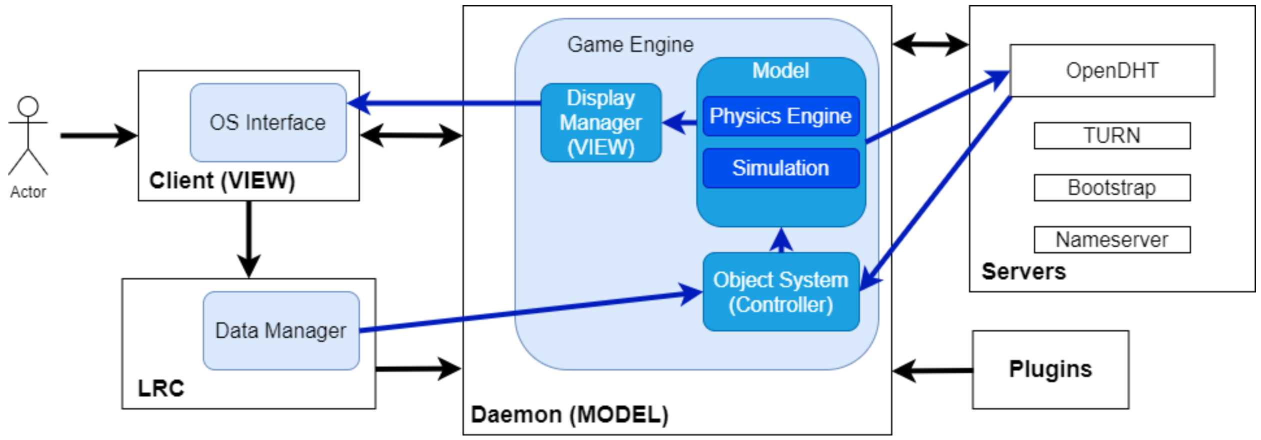

Figure 1. Updated Jami Architecture Containing Gami

One of our goals in designing the minigame feature was to minimize the architectural impact of the prior system. In other words, Gami is to introduce as few additional dependencies and subsystems as possible. To do so, we addressed the current state of the system and implemented a minigame feature in such a way that keeps the architecture of Jami consistent. As discussed previously, Jami follows a MVC (Model View Controller) design pattern driven by a peer-to-peer communication network where the daemon acts as the model, the client as the view, and the DBus/LRC as the controller. The OpenDHT is used for peer-to-peer communication. The high level subsystems of Gami are built into this MVC design pattern, and follow the same architectural style as Jami's other features. All communication between players is done via the peer-to-peer OpenDHT network, and the high level components of Gami follow an MVC design pattern themselves. Since no new dependencies are introduced between major subsystems, we minimize the architectural impact of Gami and remain true to Jami's original design.

5. High Level Changes

At its highest level, Jami introduces three new major subsystems: The Game Engine, the Data Manager, and the OS Interface. We give a brief overview here before discussing the details of each in subsequent sections. Actions and input made by the player are first sent to the Data Manager, which resides in the LRC and acts as an interface between the client and daemon (more specifically the OS Interface and Object System). It is important to note that the Data Manager component only exists in systems which make use of the LRC, since certain clients communicate directly with the daemon. Where the LRC exists, so may a local database. The Data Manager makes use of this local SQLite database to retrieve game data and game states from the file system, such as high scores and preferences, if the player so chooses.

The Object System, which resides in the daemon, acts as the controller. This system makes use of the device RAM to maintain information describing the current state of the game and communicate with the Physics/Simulation engine to make real-time changes as data is passed in through the LRC or directly from the client. By acting as a controller, the Object System can manage what kind of data it passes and delay how the data is processed to a later state. This allows for an additional degree of flexibility, since data being passed is typically game specific.

The Game Engine is responsible for implementing the actual game, and can be designed as a generic piece of software that handles a number of game genres. Naturally, the Physics Engine component handles computation related to the game state (object behaviour, audio, etc) and the Simulation engine handles computation related to game components such as AI agents and NPCs. The Display Manager handles the graphics of the game state and makes direct use of the OS Interface to implement them.

The last major subsystem introduced is the OS Interface, which manages all front end operations related to the client itself. This interface makes use of the native OS wherever possible in order to make Gami simpler and more lightweight.

6. Low Level Changes

The following directories are impacted and discussed in subsequent sections:

- install (installation related to all added components)
- lrc/src (data manager)

- lrc/test (testing related to data manager)
- lrc/src/database.cpp (database access)
- daemon/src (source code for the game engine)
- daemon/contrib (object system and additional functionalities of the game engine)
- daemon/bin (information flow via game state and display manager)
- daemon/test (testing related to game engine)
- daemon/contrib/src/opendht
- client/src

6.1. LRC Changes (Data Manager)

Recall the use and design of the LRC from previous reports: “The LRC consists of three main folders in itself: lrc/build-local, lrc/test, and lrc/src. The LRC is a daemon middleware/client library for desktop clients which interacts with the Daemon through the DBus API. It acts as an interface between the clients and the Daemon to ensure consistent behavior and portability between operating systems. The largest of the three subdirectories, *lrc/src*, consists of the main LRC components. The DBus API is located here, as well as managers, renderers, and code associated with the local database”.

The Gami Data Manager will act as an additional interface that manages game processes. The only dependence it should introduce is a direct dependence on the DatabaseManager, which is an existing interface between the client and the SQLite Database. It does so in order to retrieve game data and game states from the file system. Components related to the installation of the Data Manager will reside in the install/LRC folder, and those related to testing the Data Manager will reside in the lrc/test folder. Hence, the Gami Data Manager will act much like any other interface in the LRC residing in lrc/src, and not alter the concrete architecture of the LRC in any major way.

6.2. Daemon Changes (Physics Engine and Simulation)

The data processed by the Object System is passed into the Model subsystem of the Game Engine. The model is responsible for determining new game states based on user input and other in-game occurrences. The Physics Engine and Simulation work together to perform the necessary calculations for updating the game state that will be passed to the Display Manager and shown to the user.

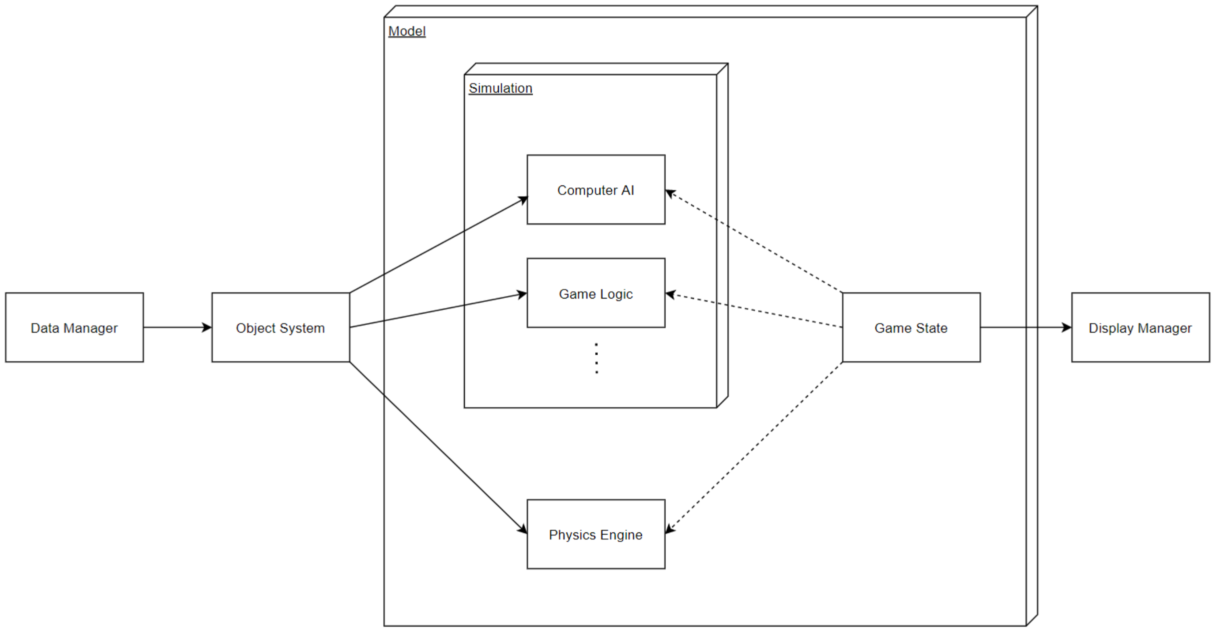

Figure 2. Conceptual Architecture of the Game Model

The parts of the Simulation and Physics Engine do not communicate directly, but rather update a global game state, which they also all draw information from that will be used in their calculations. This way, we can remove direct dependencies between parts of the Simulation and Physics Engine. While the elements of the Simulation subsystem and the Physics Engine all update the game state in a similar way, we decided to visually separate the two, because we felt that the Physics Engine was large enough to warrant being detached. Once all of the necessary calculations are made for a given part of the game, the global game state's information is passed to the display manager to be processed and relayed to the user.

6.3. OpenDHT Changes

Referring back to our first report, the purpose of the DHT is to build and maintain a hash table of key-value pairs which is saved across all the different peers of the Jami network. In Jami, the DHT has additional functionalities - the ability to store arbitrary values of sizes up to 64 KB, a listen operation (similar to get) that informs the requesting node of changes of values at a provided key for a few minutes, a value ID to distinguish different values stored under the same key, and a value type, attached to every value, that allows different expiration, creation, or editing policies to be applied to different values[4].

We can use these pre-existing features to implement our in-chat gaming without any changes to the current DHT structure. Since our implementation of Gami involves first downloading the games onto the users' devices, all we need to pass from user to user is information that Jami can interpret to be a game state. We already have the ability to store multiple values under the same key thanks to value IDs, so each active game's state can be stored as a separate value under the key, and can be removed once the game is completed or after a set amount of time of inactivity (thanks to OpenDHT's expiration times). Since we can store arbitrary values up to a size of 64 KB, we will have no memory issues storing the current game state of active games. Finally, we can notify each player when it is their turn to play in the same way we alert a user of a new call or message via the listen operation.

6.4. Display Manager

The Display Manager acts as a way to present the corresponding view according to the game state. It receives input from the Gami model to determine what front end information to pass on to the Client view. According to a given game-state, for example the layout of a chess board, the display manager will update the Client's local view to visually match the state. It essentially is the intermediate step between what the Game Engine outputs and what the User sees through the OS interface. It resides within the Game Engine which operates on the Daemon.

6.5. Client OS/Interface

The new client interface will feature some significant changes. The Gami icon will be located on the right side of the message box. From there, the user can tap on the icon which will prompt them to select a game, and then send it similar to how they would with a regular message, as seen from Figure 3. The user and their corresponding messaging partner can then simply tap on the game message within the chat to play their respective turns. If the user does not wish to use the Gami functionality, they can simply ignore the Gami icon and continue to message their correspondent as per usual.

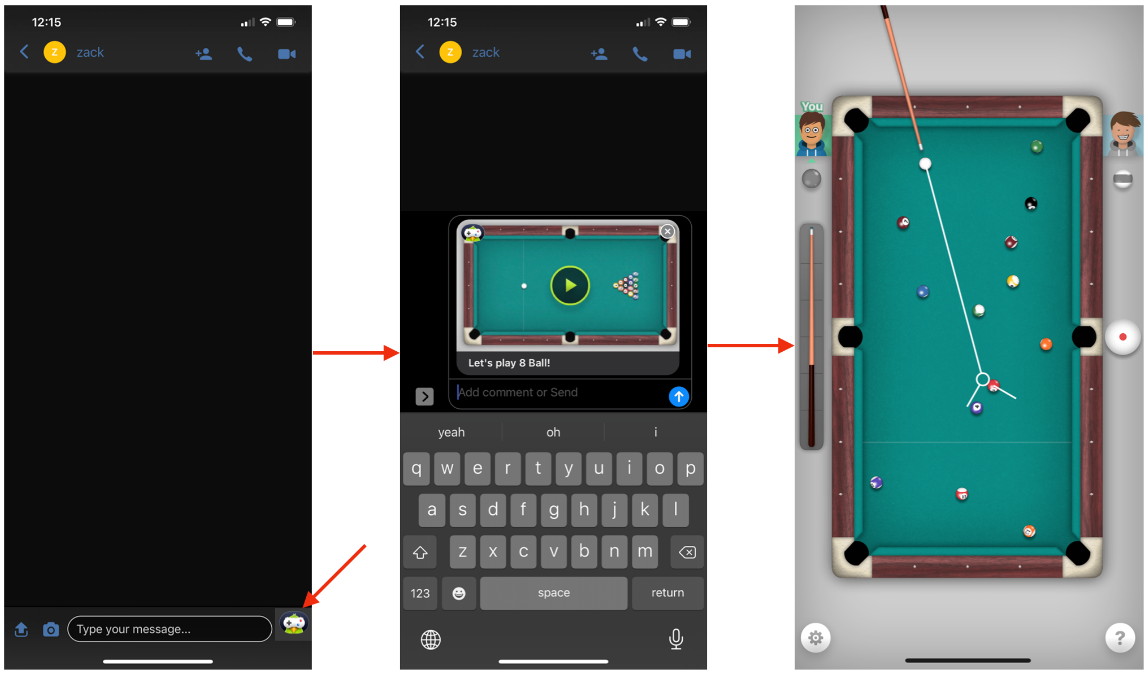

Figure 3: New Jami client interface with Gami (courtesy of Game Pigeon[5])

The available games are seen below in Figure 4. The user can select any of the following state-based games to play with their messaging correspondent. Towards the top right corner is where the number of the wins the user has is displayed. We discussed how we will make use of an SQLite database within the LRC to track statistics such as these.

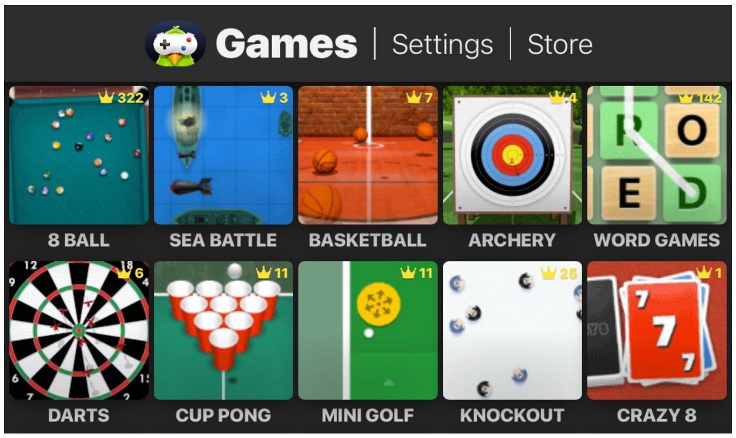

Figure 4: Some of the available state-based games to play on Gami (courtesy of Game Pigeon[5])

7. Testing

As we learned in module 2 of the course, having the correct requirements are essential to making and maintaining good applications. There are five main components we learned - requirement engineering, architecture analysis, design and implementation, testing, and finally maintenance[2]. As we can see from this, testing is a key component that comes right before release, so taking full advantage of this phase is important to ensure that both user expectations and functional and non-functional requirements are met.

Adding a game engine to a previously lightweight messaging app can have non-negligible effects on many aspects of the software's performance. The most obvious of these is load times within the app. We would compare the pre-Gami version of the application to the new version that contains it, and run tests to see if/how load times differ. The main load times we would analyze are those that directly involve the game engine. We would check if loading the app on startup was slower, as well as all the functionalities of the texting view since that is where Gami is used. These would include any lag within the texting view, load times of the games themselves, as well as any delays on retrieval times from the DHT since we are potentially adding many values under each key. If we had time, we would also check the load times of processes not directly utilizing the game engine. Since we implemented the game engine within the daemon, and since the daemon is the core of Jami, we cannot assume that it's presence will not slow down other processes that work with the daemon.

Beyond these software tests, we believe there are some device analytics and performance tests it would be worthwhile to investigate. With the additional complexity and size that would likely come with the implementation of Gami, we would want to monitor Gami's effects on a device's battery life and overall temperature. We would likely lose users if our new implementation was associated with device damage or excessive wear and tear. One final analytic we would investigate, and arguably the most important of them all, is app usage statistics. In order for an application to be successful, it is important that it has a sufficient user base - especially when it runs on a peer-to-peer, distributed network. Therefore, it would be crucial to gather information as to how much Gami was contributing to bringing in new users and keeping existing users active on our platform.

8. NFR Impact

The implementation of Gami has little effects on maintainability. Due to the fact that we decided to keep the games saved on each user's device. In order to add or enhance a game, developers would need to have the users update their software to add these features. In terms of evolvability, developers can implement more features to games or add more mini games for users to play among their friends. Our proposed enhancement of a mini game would allow for Jami to evolve over time as more features get added. Gami allows for developers to constantly be able to evolve their software and create games that appear to their users. Developers need to test how Gami works among different platforms. It is essential that Gami works for all users and does not cause any latency or bugs. Therefore, testability is essential as we do not want users to stop using Jami due to in game problems. Lastly, it is important that Gami has strong performance. This requires a fast response time as we do not want users to experience latency. It is essential that the games do not lag as it will draw users away from the software.

9. Associated Risks and Limitations

Some of the potential risks we considered when implementing Gami were security risks, software bloating, and DHT storage issues. The implementation of Gami is effectively done by putting the client-server architecture typical of online games within Jami. With most client-server architectures, the security of sensitive information and data is a concern. However, we believe that by using the DHT to store game states, we have found a workaround to this potential risk. Not only do we know that the DHT is already a secure server, but the information about the game states is not sensitive - it does not contain personal data; it just tells the players how their game board should look. A more likely risk of implementing Gami would be software bloating. Adding additional components to Jami, especially complex ones like a game engine, can lead to reduced performance and a greater application footprint. Finally, there is the potential for DHT storage issues caused by storing excessive game states. Since each Jami user could technically have a live game with each of their contacts, we run the risk of having too many values under each key within the DHT. In order to circumvent this risk, we could instead come up with a limit to the number of live games a user can have. Another limitation would be the variety available within Gami's game library. Due to how we plan on passing game information through the DHT, we are limiting the types of games we can support. Only simple turn-based games would be playable, whereas live multiplayer games would not.

10. Use Cases

10.1. Starting A Game

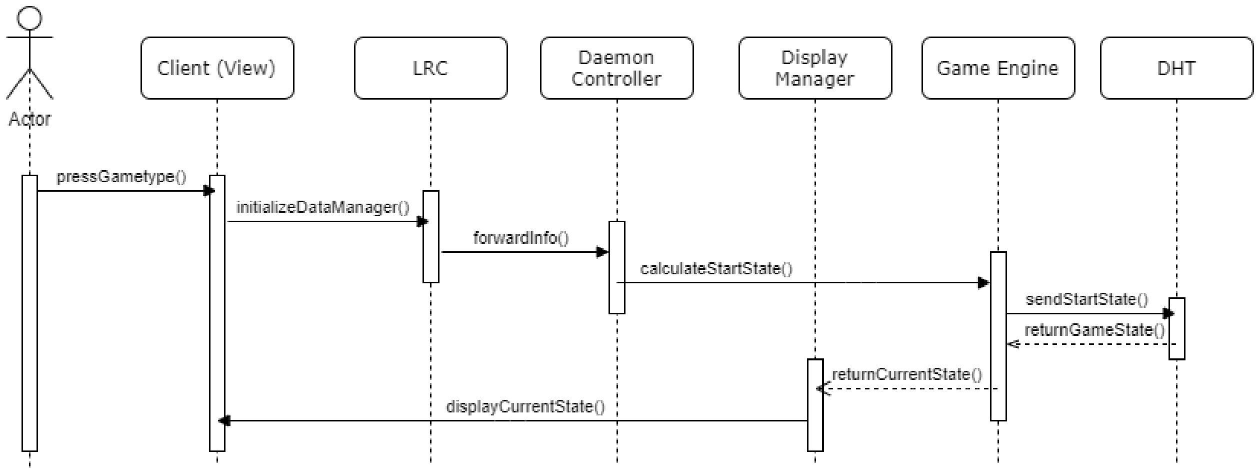

Figure 5. Use Case for Starting a Game

Our first use case that makes use of the Gami architecture improvement is relatively simple. Here we have an actor starting a game with one of their contacts. The use case starts with the actor clicking the game type they want to play. We display the start game options within the text chatroom so there is no need to go to a different menu or select a contact to play with after pressing the desired game. From there, we initialize the data manager with the default data values for starting the game, which we pass forward through the daemon to the game engine. The game engine, thanks to it's built-in physics and simulation capabilities, utilizes these values to calculate the state of the game. From here, it sends the start state of the game to the DHT. Both the challenger and the player that is being challenged receives the updated state of the game from the DHT to ensure synchronized game states. We say that the player that is challenged makes the first move, so the player that starts the game does not have an option to take a turn and the starting state of the board is returned and displayed on their device. Having the challenged player go first also alleviates the need to request to play the game - the challenged player can make a move and start the game, or can ignore the request and the game remains unplayed.

10.2. Taking A Turn

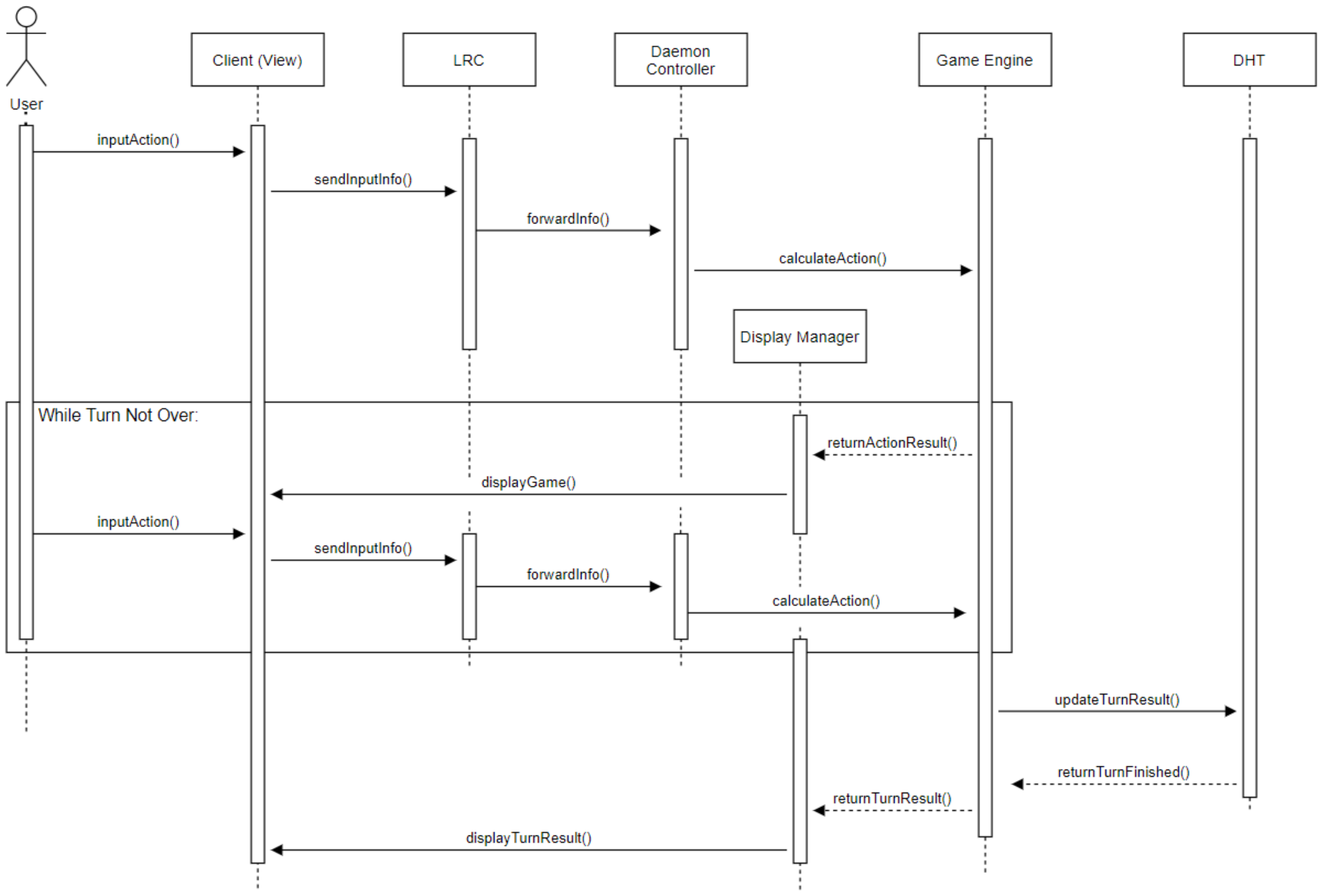

Figure 6. Use Case for Taking a Turn

This use case begins with the game being displayed on the user's device. To commence the use case, the user inputs a specific action related to the correspondence game they are playing. Their action (one or more clicks or keyboard inputs) is read in by the Client, and then passed to the LRC. The LRC manages the user's action, and passes it through to the Daemon's Controller designed to receive the game inputs. The controller then forwards the user's action to the actual game engine, where game rules, simulation, and physics engines reside. After the result of the user's action is calculated, the result is returned to the Daemon's Display Manager. The display manager then actually outputs the result of the user's action to the client, which the user can see and interact with. This process is repeated until the user's turn is officially over, as a turn can potentially have many parts associated with it depending on the game being played. Once the turn has been calculated by the game engine to be over, the result of the turn is sent to the DHT, where information is updated accordingly. Then, the DHT sends a message back to the main game engine indicating that the turn is over, which is sent to the display manager and displayed to the user on the client, terminating the use case.

11. SEI SAAM Architectural Analysis[3]

11.1. Store Game Data On A Server Not On Clients' Devices

1. The major stakeholders are the users and the developers of Jami.
2. Users' top NFR is privacy, given they are using Jami. They are also likely interested in response time, availability, and reliability. Developers' top NFR is reusability to allow the game to be implemented from previously designed libraries, and not from scratch. They are also likely interested in the capacity of the game system and maintenance, as capacity is important to compare a server implementation, and maintenance is important in order to determine how often the game system has to be updated.
3. From the user perspective, privacy is better if the game information is stored locally on players' devices, as opposed to a server. Availability will not be affected if the server is online 24 hours a day, as the P2P implementation would achieve the same result. However, reliability and response time may be improved with a server, depending on how well the server is maintained.
   From the developer perspective, reusability is better with a server implementation, as there are more libraries available for this more common game implementation. Maintenance and capacity are also going to be better if the server is well structured.
4. In conclusion, the P2P implementation is a safer approach, and more in line with the privacy focus of Jami overall. The only potential improvement is that a strong server could improve some aspects, but it would affect privacy too much and likely not appeal to the Jami user base.

11.2. Support Single Player Mini Games

1. As before, the major stakeholders are the users and the developers of Jami.
2. User's top NFR is playability. They want a game that is easy to play and fun. They also would want portability, as the game should be available on all the Jami platforms. In addition, they would want good response time for their games. Developers, on the other hand, would be most interested in integration into the game system. As before, they are likely interested in reusability which will make the implementation easier to do.
3. From a user perspective, playability is improved by adding a single player mode, as it adds more game options and would appeal to more users. Portability may be reduced, as it will be harder to make the game available on all platforms. Response time is unchanged.
   For developers, integration and reusability are both worse - a more complex game system will be more difficult to build and maintain.
4. In conclusion, the added playability that users can enjoy, although not much more work for the developers potentially, is likely not a useful feature, as users probably have better single player games they can play on their other devices.

12. Derivation Process

Our group decided to first split up and create our own versions of what each of us thought the architecture would look like once we implement Gami. During this time we researched other applications that required a game component. By doing this we were able to understand the different components that are required when creating a game. We then had a meeting once we understood the different components and how they interact within a game architecture. We implemented it within our previous concrete architecture.

13. Lessons Learned

Our ideation of Gami resulted in significant learning curves which ultimately benefited our team in multiple aspects. We had to approach this assignment in a much different manner compared to the previous assignments as they revolved around creating an architecture for an already existing system, while in this assignment we had to create our own system and architecture. This required the team to think in a more creative process to further understand how to contribute to an open source project. Deriving an architecture from existing source code is one thing, but creating your own architecture based entirely off a proposed idea is something completely new. We iterated through multiple ideas to come up with the most efficient way of implementing Gami. It taught the team about the troubles of adding features to an open source project which is so deeply built already, such as how to use existing components for use by Gami instead of creating complete new components which could cause complications with the entire system. Looking back at our work, the team strongly values the obstacles and complications we went through to derive the architecture for Gami, as the lessons we learned will prove to be beneficial down the road when creating new architectures.

14. Conclusion

In summary, we have provided an in-depth analysis of our proposed Jami feature: the “Gami” minigame feature. We have discussed the low and high level implications of this added feature, its proposed implementation in the context of Jami’s existing architecture, as well as providing use cases to show how two scenarios would be performed among the different components. We also conducted associated risk analysis, discussed the limitations of Gami, and showed an SEI SAAM Architecture Analysis to evaluate the outcomes of two different implementation approaches, using the stakeholders and their relevant non-functional requirements to delve deeper into these.

Given our extensive analysis, we believe Gami would be a useful addition to the Jami platform. It would provide users with a minigame system which, while not very difficult to implement from the standpoint of Savoir-faire Linux, would have what we believe to be a beneficial impact on users who are looking for more ways to engage with the platform. We propose that adding the Gami feature would be a good step in continuing to expand the scope of Jami, and it would be in keeping with the core principles that govern Jami’s purpose.

15. References

[1] Doherty, M. (2003). A Software Architecture for Games.

[2] Hassan, A., & Adams, B. (n.d.). *Module 02: Non Functional Requirements (NFR)—Quality Attributes*.

[3] Kazman, R., Webb, M., Bass, L., & Abowd, G. (n.d.). SAAM: A Method for Analyzing the Properties of Software Architectures. Retrieved April 07, 2021, from https://resources.sei.cmu.edu/asset_files/WhitePaper/2007_019_001_29297.pdf

[4] wiserweb. (2020, March 23). *Savoirfairelinux/opendht*. GitHub. https://github.com/savoirfairelinux/opendht/wiki

[5] Zlotskiy, V. (n.d.). GamePigeon. Retrieved April 07, 2021, from http://gamepigeonapp.com/