CISC 322: Software Architecture

The Concrete Architecture of Jami

Derived by Group 21

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

March 30, 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 examines the concrete architecture of Jami, a distributed and open source communication platform. Where the conceptual architecture was derived from associated documentation and wiki pages, the findings presented here are drawn directly from source code. The concrete architecture of a system is often difficult to interpret, as there tend to be many convoluted dependencies and intricate components. In order to overcome this challenge, we leveraged the Understand tool and our previously derived conceptual architecture to better understand the concrete architecture of Jami. Consistent with our previous derivation, we propose that at its highest level the Jami architecture follows a MVC (model-view-controller) design pattern driven by a peer-to-peer communication network. However, in the course of our investigation we find some unexpected dependencies in the source code which are used to revise our conceptual architecture via reflexion analysis. To more thoroughly discuss its components, we have broken down the Jami architecture into a set of key subsystems including the Daemon, LRC, Client, Plugins, and Servers. In particular, we delve into a more detailed analysis of the Daemon. In order to better understand Jami's concrete functionality, we take a look at the control flow and external interfaces of the system, as well as reviewing the two use cases previously laid out: logging in/ adding a contact and making a call, and looking at them from the perspective of the concrete architecture and source code. In the final sections, we discuss the derivation process and lessons learned.

Introduction

In the previous report, we derived the conceptual architecture 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. The conceptual architecture derived previously was extracted from the documentation found on the Jami website and Github repository. This time around, the team was asked to derive the concrete architecture of Jami by leveraging the Understand tool in a more detailed investigation of the source code, as well as to iteratively revise the conceptual architecture using reflexion analysis.

This report will investigate the derivation of the concrete architecture of Jami, looking at individual subsystems and breaking down the daemon in more detail. It will involve a revision of the conceptual architecture, as well as an analysis of the control flow and external interfaces of the system in the concrete case. In addition, in order to comparatively illustrate the interaction between some of the key components previously mentioned, we will take the two use cases previously laid out: logging in/ adding a contact and making a call, and look at them from the perspective of the concrete architecture and source code.

At a high level, we believe our original conceptual architecture matches the concrete case: We propose that the Jami architecture follows a MVC (model-view-controller) design pattern driven by a peer-to-peer communication network, in which the daemon acts as the model, the client as the view, and the DBus (which is within the daemon) as the controller. However, we acknowledge some minor changes which we reflect in the revised conceptual architecture. These changes mainly include the development of plugins as a separate subsystem and the unexpected dependencies between the client and daemon, all of which are supported by reflexion analysis. We conclude with a discussion of the derivation process and lessons learned.

Overview of Concrete Architecture

The overall structure of Jami in the concrete architecture is a peer-to-peer distributed network functioning over a model-view-controller architecture. The MVC structure consists of the daemon as the model, the client as the view, and the DBus acting as the controller. The distributed network is handled by the OpenDHT server within the Servers component. Plugins was a new component added with respect to our original proposed conceptual architecture. The dependencies are shown in Figure 1.

Summary of Modifications to the Conceptual Architecture

The concrete architecture was very similar to the proposed conceptual architecture after investigation of the source code in *Understand,* with some added dependencies. Unexpected dependencies that resulted after analysis are shown in Figure 1 in red arrows, as per the legend. These are investigated in detail through a reflexion analysis in the later section of the report, using the W4 approach, of Which, Who, When, and Why, and here we present a summary of these modifications.

The first of these additional dependencies is the Plugins component, which has a one-way dependency to the Daemon. This differs with respect to the proposed conceptual architecture, in which we did not have a Plugins component at all, but believed they run on the user's device. An additional dependency is that the client also depends on the Daemon, and not only the Daemon on the client. This dependency is most likely unintentional, and should have been avoided, which we deduced from the fact that the low-level tools are outdated and not properly adapted by the developers. The next modification of the conceptual architecture previously proposed is that, instead of the user having a dependency from the Client and an optional dependency on the LRC, the user simply has a dependency on the Client directly. Finally, the database component was removed, as it was found, instead, to be a subcomponent of the LRC component.

Next, we turn our attention to alternative architecture styles for similar software. The main alternative to P2P architecture, when it comes to communication platforms, is the client-server architecture style. Client-server offers the advantage of better performance and responsiveness of, say, a video conferencing system, if it is powered by a performant server, but it has the drawback of being harder to scale, and all user data is stored on the central server. This limits communication privacy immensely, as companies can then use this data liberally. P2P, conversely, has the advantage of being very scalable, and of having no central storage of user data, providing much better privacy. However, its main drawback is that it can be less performant than a client-server system if the users, or peers, do not have strong hardware to keep the network running smoothly.

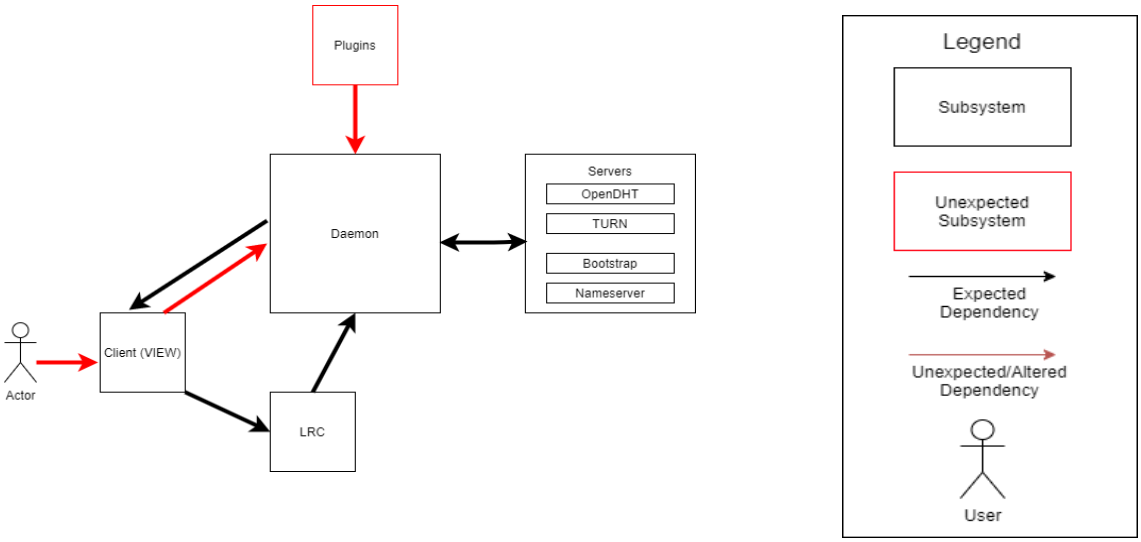

Figure 1: Jami Concrete Architecture. This figure depicts the concrete architecture of the Jami communication platform, as determined by source code analysis in Understand.

Subsystems

Daemon

Overview

From earlier documentation on the Daemon, we know that the daemon as whole is responsible for interpreting the results from other subsystems, connecting other subsystems together, and providing some other tools such as system configuration and multimedia functionality. Analyzing the dependencies and the connections between sections of the software, our understanding of the Daemon was proved mostly consistent with our prior analysis.

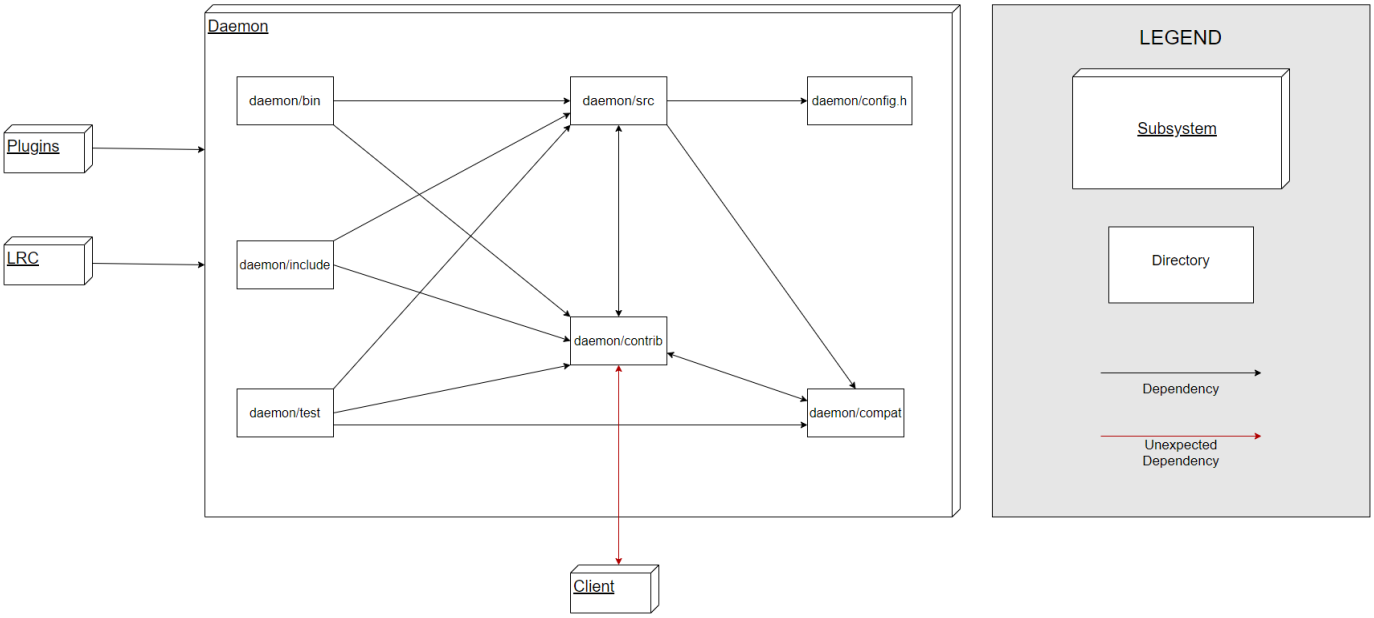

Figure 2. The concrete architecture of the Daemon

There are several parts in the Daemon that communicate with each other, which other elements of Jami draw upon. The Daemon's */bin* file is responsible for managing a large portion of information flow, as it contains the dbus and another folder called */nodejs*. From the last report on conceptual architecture, we know that the dbus contains several elements which are essential for other subsystems in the software receiving the correct information, and providing an interface for inter-system communication. On the other hand, the folder called nodejs is responsible for assisting in the reading and interpreting of node information in the DHT.

The */src* folder contains the main source code for the Daemon, and so nearly all of the other sections in the Daemon are dependent on this folder. In addition to sections being dependent on the source code folder, some source code files are also dependent on other files. Some source code is dependent on the *config.h* file, which stores system information relating to how parts of the software are being stored on a user's device. We also see that source code is dependent on several folders in the */contrib* section of the Daemon, which contains several special functions or tools that provide assistance or new functions for other subsystems used in Jami.The */compat* folder stores basic information such as the time of day, and other basic configurations that many

sections of the Daemon and LRC depend upon. Since the */compat* folder is used for the storing of important device information, rather than performing complex computations or other larger functions, it has no dependencies.The final important section in the Daemon is the */include* folder. This folder is only dependent on the source code, with incoming dependencies from the LRC. The */include* folder is responsible for the creation and storing of several dring interfaces. Files in this folder retrieve and interpret information from other parts of the Daemon that will be used by the LRC. Specifically, the majority of the files in the */contrib* folder relate to the configuration manager interface in the dring, which files in the */include* folder draw from.

The files included with the downloaded Jami package also have a test folder that is contained within the Daemon. No elements in the software depend on this test folder, while the test folder depends on the server as well as several sections of the Daemon. This folder is essential for testing the software and ensuring that everything is running properly. This is not necessary for Jami to run, but provides some additional insight for developers analyzing the software. There are many simulated sections from the Daemon contained in the test folder such as a connection manager, different utils, and a simulated file transfer. All of these simulated subsystems are run through the *test_runner.h* file, which simulates different cases or problems the Daemon may encounter.

Analyzing the dependencies created from the Understand tool, we can see that nearly every subsystem in Jami is reliant on the Daemon, as the Daemon as a whole is largely responsible for managing the flow of data and provides the main interfaces for the software. While every main subsystem depends on the Daemon, we can see that the Daemon only depends on the client and the central server. While we do not have code for the server, we can safely assume it functions in line with our previous understanding of it, as most servers of this nature behave similarly as a whole. We expect the Daemon must be dependent on the central server, as the Daemon must read information from the DHT and relay it to other sections of the system. However, the dependency on the client is unexpected. By looking at the structure at a lower level, we can see that *pjproject* in the *contrib* folder of the Daemon is dependent on a part of the section in the client used for reading QR codes called “optarg”. *Pjproject*, also called PJSIP is a multimedia tool, using audio, video, and other forms of media available on a given device. Through reflexion analysis, we have determined that the QR code library called *libqrencode* was downloaded and used in Jami, which appears to have been last updated around 2012, developed by Kentaro Fukuchi. Similarly, Pjproject is another tool downloaded and implemented in Jami, where the version being used appears to have been last updated around 2011, by Benny Prijono and Teluu Incorporated. It is unclear exactly who on the Jami team implemented these functions, but it is possible that since the files are so old, and since they are tools downloaded from other sources rather than being manually created code, this dependency was unintentional and could have been avoided if more was known about the tools’ low-level functionalities.

The DBus

In order to better understand the daemon, we chose to investigate the DBus (*bin/dbus*, the D-Bus XML interfaces, and C++ bindings) in more detail. The DBus is the API of the daemon. It depends on *daemon/contrib* (which contains software that has been contributed to the project, but which might not actually be maintained by the core developers) and *daemon/src*. It is depended on only by the file *daemon/bin/main.cpp*, which initializes DRing.

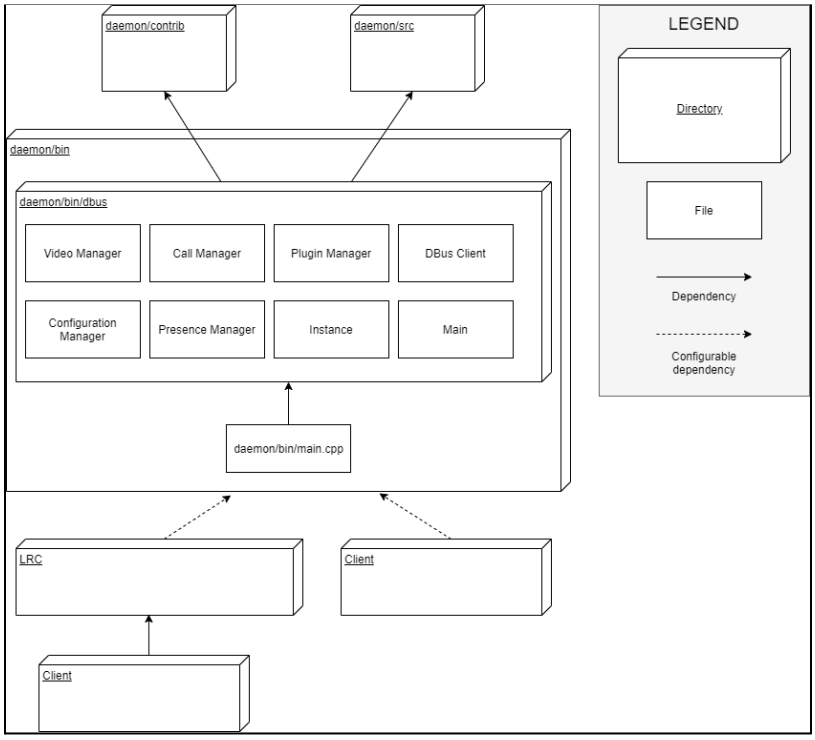

Figure 3. The concrete architecture of the DBus

It makes sense that no client is directly dependent on anything in *daemon/bin* (including the DBus). Clients need to be configured to control the daemon, typically via the LRC. As is written directly in *main.cpp*, "One does not simply initialize the client". Taking this into consideration, a closer look at the *bin/dbus* can tell us a lot about the daemon's features. Within the DBus, we see a plugin manager, presence manager, video manager, call manager, configuration manager, dbus instance and dbus client. Within each manager is a set of variables and methods related to the functioning of the daemon (these are not discussed here since they are too low level). Note that the DBus instance doesn't act as an interface but rather keeps track of clients registered to the core, and the DBus client initializes the individual managers. It also makes sense that *bin/dbus* itself relies on *contrib*, likely for third party/external libraries and *daemon/src*, through which the daemon can actually be controlled.

This low level subsystem can be compared to the layered architectural style. The DBus API acts as an intermediate layer used by the clients to control the daemon. This ensures consistency, privacy, and reliability however offers little in terms of flexibility.

LRC

From a high-level view of the project dependency graph the LRC immediately appears to be an important component. It is depended on by the install subsystem, as well as all clients with the exception of *client-uwp* (Universal Windows Platform) and *client-ios*. The LRC itself depends on the install subsystem and the daemon.The LRC consists of three main folders in itself: *lrc/build-local*, *lrc/test*, and *lrc/src*.

Recall the role of the LRC from the conceptual architecture: "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 DatabaseManager is an interface between the client and the SQLite Database, however should not be directly called from the client. The NewCallModel interface manages calls, the ContactModel interface manages contacts, and the ConversationModel interface manages conversations and messages" [7].

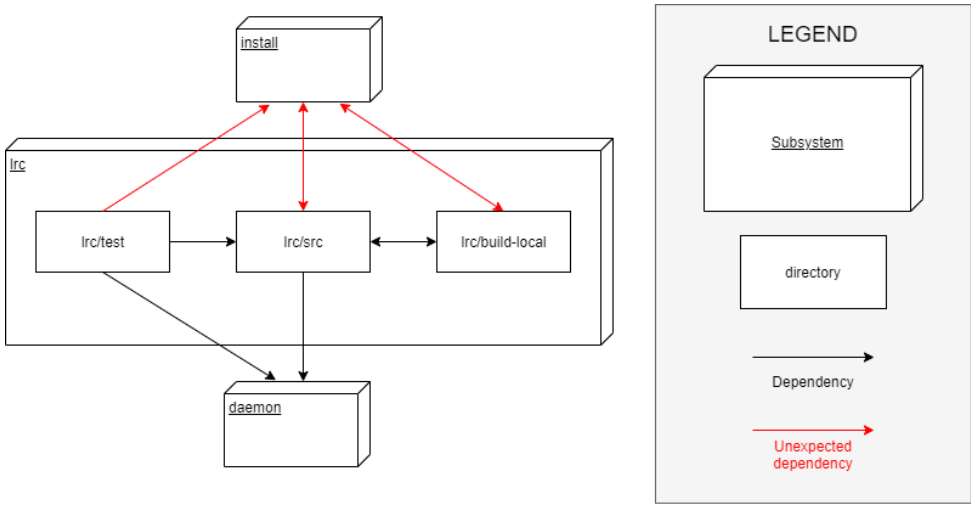

Figure 4. The concrete architecture of the LRC

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. *lrc/build-local* mainly consists of a number of DBus interfaces. *lrc/test* includes a number of mocks and tests for components implemented in *lrc/src*. Although low level files are not included in the diagram, a closer inspection of *lrc/src* shows that no client has direct access to the database, which is consistent with the conceptual architecture and documentation.

The concrete architecture deviates from the conceptual architecture of the LRC only by its mutual dependency on the install directory, which is generated locally and is not directly mentioned in the Jami documentation. Reflexion analysis shows that the LRC depends on the install subsystem for initializing the api, name directory, interfaces, etc. This process is confirmed and outlined in more detail by established developer Albert Babí Oller in the Jami build instructions [12], which was last updated a month ago, and by intern Hugo Lefeuvre in the LRC Installation instructions [11] from two years ago. Since it is used to initialize the LRC locally, it is safe to assume this is a valid dependency.

Clients

Jami is available for use on a number of widely used platforms, including: Windows, GNU/Linux, Mac OSX, iOS, Android, and Android TV. From analyzing the high level

dependency graph through *Understand*, we can see that 5 client subsystems exist. *client-gnome*, has dependencies on *client-qt* and *client-uwp*, one of which is a component of Linux architecture, and the other of Windows architecture. *client-qt* deals with the Linux component Qt (pronounced "cute"), which is a widget toolkit used for creating Graphical User Interfaces. *client-uwp*, deals with the Universal Windows Platform (UWP for short) which helps run apps on various Windows clients such as Windows 10, Windows 10 Mobile, Xbox, and HoloLens without having to rewrite the code for each OS. The high level dependency graph actually corresponds to our proposed conceptual architecture discussion about clients, where the Android client does not interact with the LRC. This is visible in the high level dependency graph as there is no connection between *client-uwp* and the LRC component. *client-qt* actually has 1 dependency on *client-uwp,* although both have dependencies on the Daemon. In addition to having dependencies on *client-uwp* and *client-qt* the *client-gnome* also has dependencies on install and the Daemon. *client-macosx,* similar to *client-qt* also has 1 dependency on *client-uwp*. The iOS client (*client-ios*) appears to be the only client without connections to any other client - only has 2 dependencies to Install and the Daemon. The high level dependency graph visually explains what we had discussed in our first report about the Clients. A user can implement its own client on a variety of operating systems while still maintaining all the features.

Servers

The main subcomponent of the server component is the OpenDHT, whose source code was found in *daemon/contrib/native/opendht*. Because the OpenDHT source was found in the contrib folder, we know that it may not be maintained by *savoirefairelinux*. For this reason, similar copies exist in the *daemon/contrib/x86_64-linux-gnu* folder and as a compressed tarball file in *daemon/contrib/tarballs/.* The purpose of OpenDHT is to provide the Distributed Hash Table method of networking on which the entire peer-to-peer network is built. It has a single two-way dependency on the Daemon.

To get a clear understanding of the servers, we tried to look at both the source code and our Understand file. We noticed that some files were not included in Understand that we believed to contain information about the servers. These included much of the contents of *daemon/contrib/src/* and *daemon/contrib/bootstrap.* The bootstrap file may not have been processed by Understand because it has no connections to any other files, and should simply be used to connect to the DHT network as per the documentation instructions. According to Jami documentation, “The bootstrap server is simply an OpenDHT node that we maintain and keep online at all times. Jami is configured to join the network through it when it connects for the first time. You can see this parameter in Jami’s advanced settings. The default value is ‘bootstrap.jami.net’.” Instructions for making your own DHT node are available, as well as how to keep it running (https://github.com/savoirfairelinux/opendht/wiki/Build-the-library , https://github.com/savoirfairelinux/opendht/wiki/Running-a-node-with-dhtnode).

In contrast, the TURN and nameserver are optional and we could not find code for them in the source folder. This could be because the code does not exist (since it does not need to exist) on the device, the code has cryptic naming, or we were analyzing an imperfect Understand build.

There are no discrepancies between our original conceptual architecture and our concrete architecture. It makes sense, as originally identified, that the Server component needs to have a

two-way dependency on the Daemon, in order to both send and receive information about the Jami network. For this reason, no changes were made to this dependency while updating our conceptual architecture and a full reflexion analysis is not required for this component.

Plugins

With the provided source code, we were able to use Understand to look at some of the generic plugins provided and see how they actually connect to other subsystems. Across all plugins, including the plugin SDK, we traced connections to the *plugins/lib* and *daemon/src/plugin* folders, and also to the *daemon/src/observer.h* file. We then took a closer look within these folders to find file-to-file connections. In *plugins/lib*, we found that most plugins connected to different sets of files found within. We noticed that all plugins connected to *pluglog.h*, which is used for logging errors, warnings, etc.

After taking a look at the contents of the *plugins* folder, we then briefly turned our attention to *daemon/src/plugin*. This folder contains plugin service managers, utils, and the media/chat handlers that were discussed in the conceptual architecture report. Plugins connect to the *chathandler*, *mediahandler*, *streamdata*, and *jamiplugin* files found here as necessary, with the *mediahandler.h* and and *jamiplugin.h* files being the key components used by all plugins. *chathandler.h* and *streamdata.h* were only accessed by the SDK and AutoAnswer plugins.

By performing a reflexion analysis, we can try to answer the 4 W's to explain the discrepancies. Our original conceptual architecture diagram did not include a plugins subsystem. Since we learned through documentation that "the Daemon of Jami contains a JamiPluginManager class that can perform installs, uninstalls, loads, unloads, preference editing, and can control the usage of plugins"[14], we had originally grouped plugins as a part of the daemon, and due to the high level nature of conceptual architectures, did not mention this explicitly. Now through the creation of a conceptual architecture using Understand, we can see that all plugins connect with files within the daemon source code (*daemon/src*), but are themselves separate from the *daemon*. These plugins have been available since September 2020[15], and have been in development since May 2020. We can see that all plugin commits have been made by either Aline Gondim Santos or Sebastien Blin[16]. These are both experienced developers with extensive experience beyond Jami, so we can assume that their work is correct and the existing dependencies are valid[2, 14].

Control Flow

To understand the flow of information in Jami, we can look at the major subsystems individually, interpret their dependencies, and see how the combined subsystems work together as a whole.

First, looking at the Client, we see that every subsection of the client has elements that depend on other subsystems of the software such as the Daemon and LRC. This means that whenever an operation in the client must be performed where information is passed to or retrieved from elsewhere, the client must wait for the operation to be completed by another subsystem before displaying it. The majority of the dependencies lie with the LRC, which is expected, as the LRC contains a dbus api, connecting to the main dbus located in the Daemon. In general terms, the LRC is responsible for ensuring that the information being passed from the Daemon to the Client is correct. Thus, since the LRC processes information that is traveling from the Daemon to the

Client, the client's dependencies on the LRC can be interpreted as indirect dependencies on the Daemon. There are also internal dependencies in the client, which is expected as the client is something that the user directly interacts with. Thus, in addition to the client having generally low concurrency with the rest of the software, even functions within the client have overall low concurrency.

The LRC is the section which is partly responsible for the global control of the flow of information. Thus, it makes sense that the LRC does not operate concurrently with other subsystems. It's purpose is to take information from the Daemon through the dbus, ensure that it is correct, and pass it on to the Client. So, it makes sense that the LRC is very dependent on the Daemon. There are also many two-way interdependencies within the LRC. Since the LRC performs a limited number of functions, it makes sense that the subsections in the LRC have very little, if any, concurrency with each other.

The Client, LRC, Plugins, and central server all depend on the Daemon in order to perform their intended functions. The transfer of data that takes place here is largely facilitated by the dbus, which is contained in the Daemon. The REST API is also responsible for some data flow with the central server. However, we do not have access to the actual central server, so it is unknown just how much data is transferred as a result of the REST API. Still, we can see that there is an exchange of information, and we can assume that the server is communicating with other devices at the same time. So, the server is operating concurrently with other devices, and the Daemon's server API and dbus operate concurrently (data dependent). Other cases of data flow take place via simple function calls to files within the Daemon, and can bypass the dbus.

Finally, the Plugins section runs fairly concurrently with the rest of the software. The files in the plugins folder are dependent upon the Daemon's source code directly, though the amount of dependencies is fairly low. The plugins serve mainly as additional features for the program and are not essential for base software functions. It is for this reason that no subsystems in the Jami software depend on the plugins, allowing them to run concurrently with the other subsystems.

External Interfaces

Operating System & Plugins: Jami makes use of plugins which take advantage of frameworks and resources installed, and made available on the respective OS of the user. These plugins, such as the "Greenscreen" plugin, modify the user experience and interaction with Jami. Currently, plugins are only available for Windows and Linux operating systems. Some Android devices support plugin usage however the device must have IA computing in order to make use of them. MacOS plugins are soon to come.

Graphics Rendering & Encoding: Jami makes use of a common software technique known as hardware acceleration. Jami encodes the video into an efficient format through compressing the data[8], and then decoded into an RGB format on the client end of transmission. Jami is compatible with most codecs including H264, VP8, MP4V-ES and H263. This encoding and

decoding process puts a lot of pressure on the CPU, which is where the GPU comes into action. Most devices are able to delegate these processes to the GPU (hardware acceleration) which frees up processing power for the CPU.

Networks: As we've mentioned throughout the report, Jami depends on OpenDHT to establish connections between peers connected on the network[9]. Each node on the network contains a list of known nodes with their corresponding IP Address and Jami Identification. Making a call (as we discuss in depth below), requires the user node to identify the address of at minimum, one other node on the network. This node is configurable in the user's Jami settings and is known as the bootstrap node [Jami network article]. Jami is also designed to function offline, without connection to the internet. If the bootstrap node is located outside of the user's network then Jami cannot function. However, automatic peer discovery on OpenDHT allows users to connect to nodes on the same network without manual configuration and without internet connection.

Use Cases

Making A Call

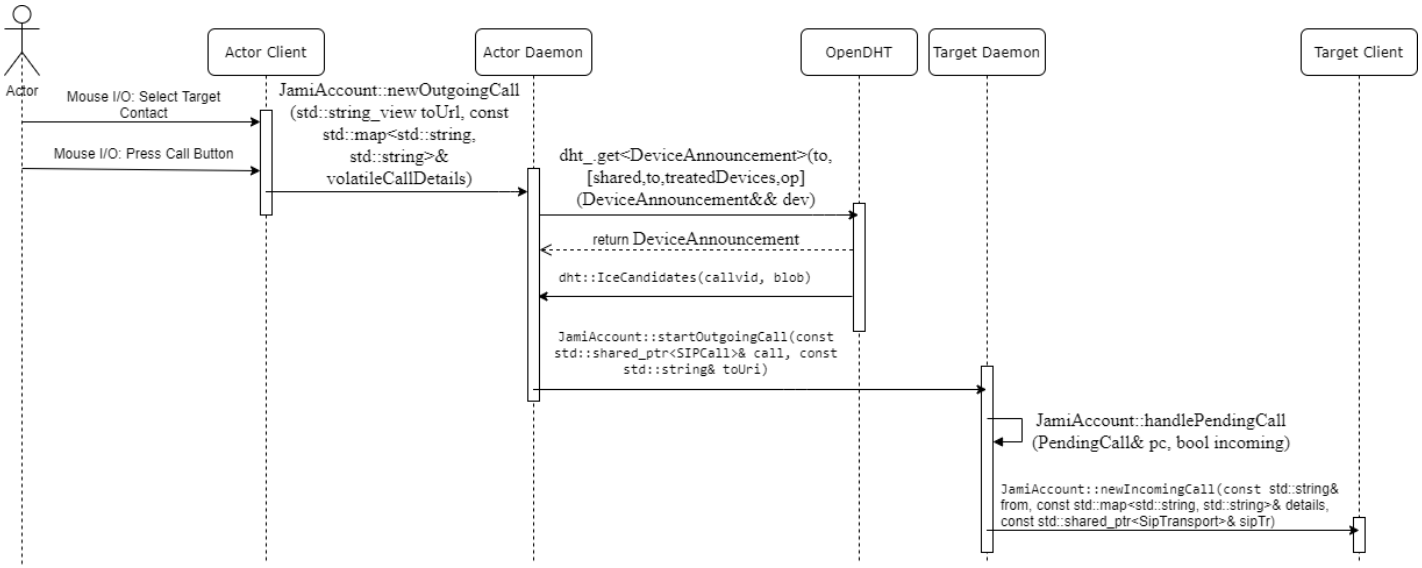

Figure 5. Use case: Making A Call

We will once again use the use case of making a call with Jami. This time, we made reference to the "Let's do a call" documentation[3], the "Manage Contacts" documentation[6], as well as the Jami source code found in *jamiaccount.cpp*[1]. We make the same assumptions as in our first iteration of this use case - assume that the user is logged in and already has the call recipient added as a friend. The process for this use case follows a similar structure to our first version: the call recipient's presence is searched for on the DHT, then a call request is sent to the contact and we wait on the return of necessary information from their device (IP, relay, and reflexive addresses).

The socket is then negotiated via ICE, and the socket is encrypted in a different way depending on which socket is used. The preferred method is TLS encryption with the TCP socket, but a fallback option of DTLS encryption with the UDP socket is available. The use case ends with the call being displayed on the target's client.

Creating a DRing Account

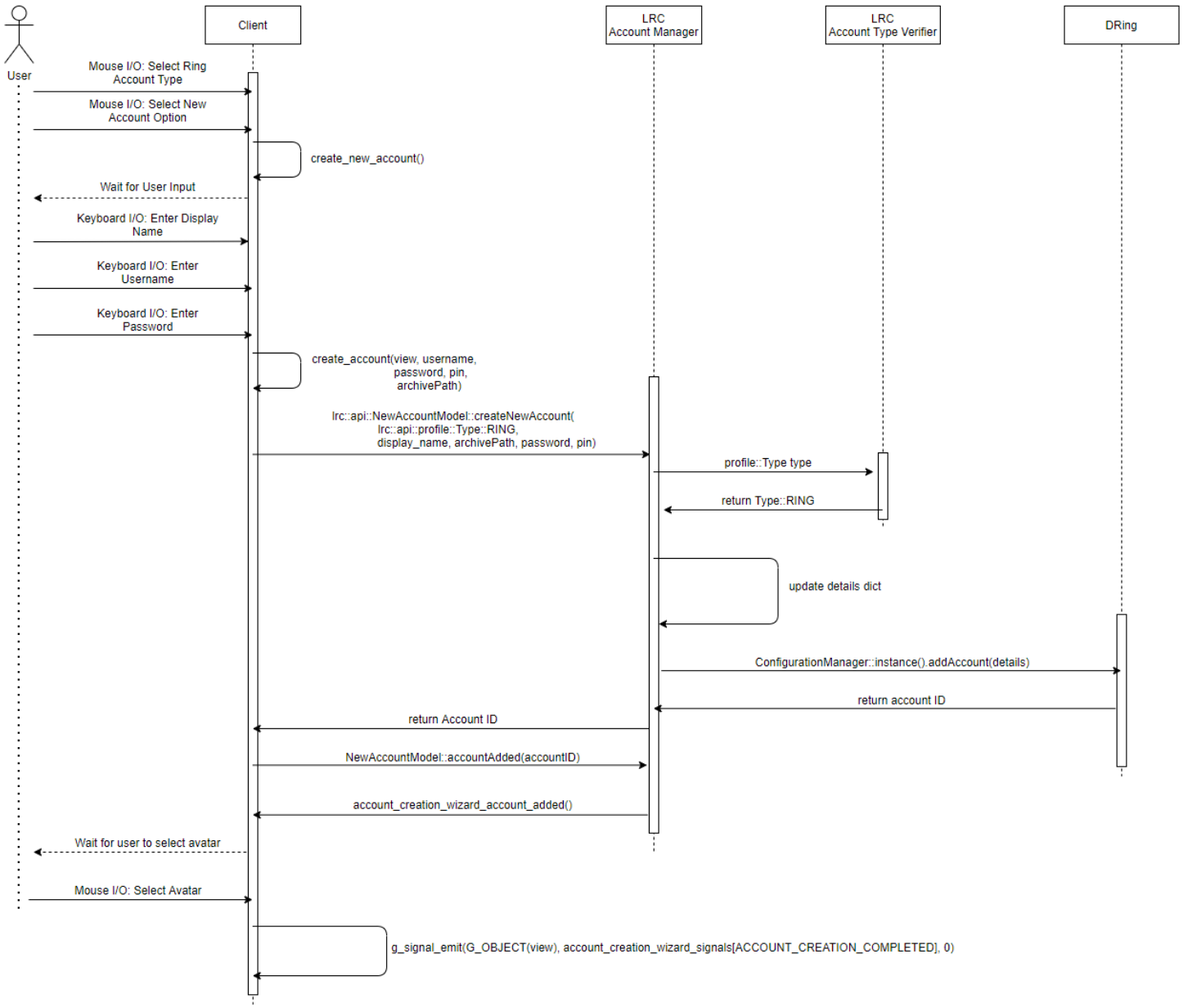

Figure 6. Use case: Creating a DRing Account

By reading the documentation available on the Jami GitLab page, we were able to determine that account creation begins in the accountcreationwizard files[5]. These were located to be in the client-gnome\src folder, and the method calls were followed for the whole account creation process.

For a user to create an account, they must first navigate to the “create new account” section, presented by the client. Then, the user must declare the type of account they wish to create. After an internal method call by the client in client-gnome\src\accountcreationwizard.cpp, the user is prompted to input their new display name, username, and password. After another internal method call, the client makes a call directly to the LRC, where the method createNewAccount() in lrc\src\newaccountmodel.cpp is called. One of the parameters passed to this function relies on the output from the lrc\src\api\profile.h file to return the proper value type, which was decided when the user selected the type of profile they began creating. A dictionary is then created, which stores the user’s account type (RING), URI (unique identifier), username, and password. This details dictionary is then sent to the DRing, where it is stored for later use by the Jami software. Inside the DRing, the user’s account ID is created, which is sent back to the LRC, and then to the client. After verifying the information, the client then declares the account to be added in the LRC, which causes the account_cretion_wizard_account_added() method to be called in the accountcreationwizard.cpp file. This method prompts the user to select their avatar, which is then received by the client and information is updated accordingly. Finally, a signal is emitted by the client indicating that the account is active, and that the new account creation is finished.

Derivation Process

Our group decided to first split up and create our own versions of what each of us thought the concrete architecture would look like. We each did this by using the source code and applied it to the *Understand* software to create the concrete architecture. In addition, we each used the documentation provided for each of the components in Jami to figure what components in the architecture contain which files. By doing this we quickly realized that there were many differences within each of our *Understand* diagrams.

Once each member of the group had an attempt at creating the concrete architecture, we decided to have a meeting. During this meeting we compared each architecture and read through the documentation for each subsystem, and came to an agreement with what we thought was the correct concrete architecture. We came to the conclusion that the Plugins, Daemon, Client and LRC had the files that corresponded to folders that were in the source code that are labeled as those components. However, for the Server component we realized that under the Daemon/contrib/native/opendht was the same as the source code that corresponds to the server component on the github page. The following figure is what our group decided was the concrete architecture.

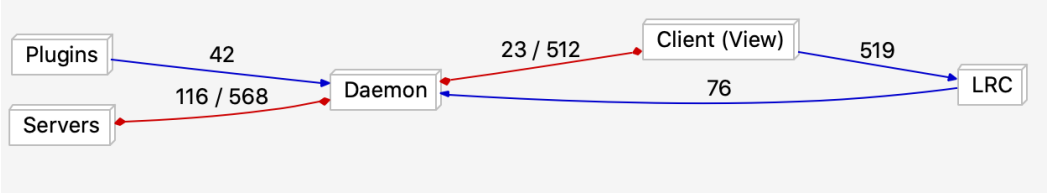

Figure 7: Our Concrete Architecture

Lessons Learned

The most significant learning curve throughout the derivation of this second report came from utilizing *Understand*. *Understand* added immense value to the team by providing a clear visualization of Jami's concrete architecture with quantitative descriptions of all dependencies. However, it was quite challenging to get the hang of using it, especially since the software was rather unstable and would frequently crash without saving our work. So, the first lesson learned was to be patient when working large software systems as it can often cause unexpected complications and setbacks. Additionally, we learned that working with large software systems have a significant amount of unexpected dependencies and are much more complicated than they appear from a conceptual architecture standpoint. We often found ourselves getting lost within *Understand* after diving down a specific component. Lastly, we learned how important communication and organization were when developing our concrete architecture. Being in a remote environment definitely increased the severity of this challenge as we were often spontaneously messaging one-another through Discord about how to alter our concrete architecture. We found that setting stricter deadlines and having more frequent team sprint calls helped us organize our process in finalizing our concrete architecture.

Conclusion

By recovering the source code and analyzing it in Understand for dependencies, in this report we have covered the concrete architecture of Jami, its architectural style and patterns, and how this was derived. Building on the conceptual architecture proposed in the previous report, we revised our conceptual model to better reflect the true structure of Jami, and using reflexion analysis, analyzed the discrepancies between the concrete and conceptual architectures to gain a deeper understanding of its implementation. Additionally, we focused on the Daemon subsystem in more detail, studying its interactions, in order to get a sense of what the low-level workings of a software of this scale look like.

In the next report, our task will be to come up with an improvement that can be made to the Jami platform that does not currently exist, and doing so will require all the knowledge we have gained in researching Jami in these two reports, both from a source code and documentation perspective.

References

[1] Béraud, A. (2021, March 19). *Src/jamidht/jamiaccount.cpp · master · savoirfairelinux / jami-daemon*. GitLab. https://git.jami.net/savoirfairelinux/ring-daemon/-/blob/master/src/jamidht/jamiaccount.cpp

[2] Blin, Sebastien. (n.d.). *Sebastien Blin*. LinkedIn. Retrieved March 24, 2021, from https://www.linkedin.com/in/sebastien-blin-25126716a/?originalSubdomain=ca

[3] Blin, Sebastien. (2019, December 8). *2.4. Let's do a call · Wiki · savoirfairelinux / jami-project*. GitLab. https://git.jami.net/savoirfairelinux/ring-project/-/wikis/technical/2.4.-Let%27s-do-a-call

[4] Blin, Sébastien. (2021, January 31). *Running a node with dhtnode*. GitHub. https://github.com/savoirfairelinux/opendht/wiki/Running-a-node-with-dhtnode

[5] Blin, Sébastien (2018, November 24). *2.1 Manage Accounts - Wiki - savoirfairelinux / jami-project.* GitLab. https://git.jami.net/savoirfairelinux/ring-project/-/wikis/technical/2.1.%20Manage%20Accounts

[6] Blin, Sébastien (2018, November 24). *2.2. Manage Contacts · Wiki · savoirfairelinux / jami-project*. GitLab. https://git.jami.net/savoirfairelinux/ring-project/-/wikis/technical/2.2.%20Manage%20contacts

[7] Duchemin, P. (2018, June 5). LRC documentation · Wiki · Savoirfairelinux / jami-project. Retrieved March 21, 2021, from https://git.jami.net/savoirfairelinux/ring-project/-/wikis/technical/LRC-documentation

[8] Jami. (2019, July 08). Hardware accelerated encoding and decoding in Jami. Retrieved March 25, 2021, from https://jami.net/hardware-accelerated-encoding-and-decoding-in-jami/

[9] Jami. (2019, May 03). Automatic peer discovery on local networks. Retrieved March 25, 2021, from https://jami.net/automatic-peer-discovery-on-local-networks/

[10] Krukova, A. (2020, July 07). Video conferencing systems architecture. Retrieved March 23, 2021, from https://trueconf.com/blog/reviews-comparisons/videoconferencing-systems-architecture.html#Types_of_Video_Conferencing_System_Deployment

[11] Lefeuvre, H. (2018, July 24). Install · master · savoirfairelinux / jami-lrc. Retrieved March 21, 2021, from https://git.jami.net/savoirfairelinux/ring-lrc/-/blob/master/INSTALL

[12] Oller, A. B. (2021, January 23). Build instructions · wiki · savoirfairelinux / jami-project. Retrieved March 21, 2021, from https://git.jami.net/savoirfairelinux/ring-project/-/wikis/technical/Build-instructions

[13] Ozinov, F. (2020, October 19). *Build the library*. GitHub. https://github.com/savoirfairelinux/opendht/wiki/Build-the-library

[14] Santos, A. G. (n.d.). *Aline Gondim Santos*. LinkedIn. Retrieved March 24, 2021, from https://www.linkedin.com/in/aline-gondim-santos-98ba87b0/

[15] Santos, A. G. (2021, January 25). *7. Jami plugins · Wiki · savoirfairelinux / jami-project*. GitLab. https://git.jami.net/savoirfairelinux/ring-project/-/wikis/technical/7.%20Jami%20plugins

[16] Santos, A. G., & Blin, S. (2021, March 19). *Commits · master · savoirfairelinux / jami-plugins*. GitLab. https://git.jami.net/savoirfairelinux/jami-plugins/-/commits/master

[17] Skype protocol. (2021, February 26). Retrieved March 23, 2021, from https://en.wikipedia.org/wiki/Skype_protocol#:~:text=9%20External%20links-,Peer%2Dto%2Dpeer%20architecture,port%20numbers%20of%20reachable%20supernodes.

[18] Universal Windows platform. (2021, March 09). Retrieved March 25, 2021, from https://en.wikipedia.org/wiki/Universal_Windows_Platform