CISC 322: Software Architecture

The Conceptual Architecture of Jami

Derived by Group 21

February 26, 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 conceptual architecture of Jami, a distributed and open source communication platform. The findings presented here are drawn from associated documentation and wiki pages, which are mainly found in the GitHub repository and on the Jami website. A conceptual software architecture spans many models and views which, in combination with often incomplete documentation, makes it difficult to extract and study. Despite this challenge, 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. This enables the relaying of data between users without the use of a server, which means increased autonomy and privacy for Jami users. In order to more thoroughly discuss its components, we have broken down the Jami architecture into a set of key subsystems including the Daemon, LRC, Clients, and OpenDHT; the OpenDHT is the primary component responsible for peer-to-peer communication and further consists of the Nameserver, Bootstrap, and TURN server. We provide additional insight into Jami's conceptual functionality through an extensive analysis of concurrency and data flow, which touches on the variety of APIs and connections Jami uses to operate. In order to illustrate the interaction between some of the key components previously mentioned, we lay out two use cases: logging in and adding a contact, and making a call. In the final sections, we discuss the derivation process and lessons learned.

Contents

1. Introduction

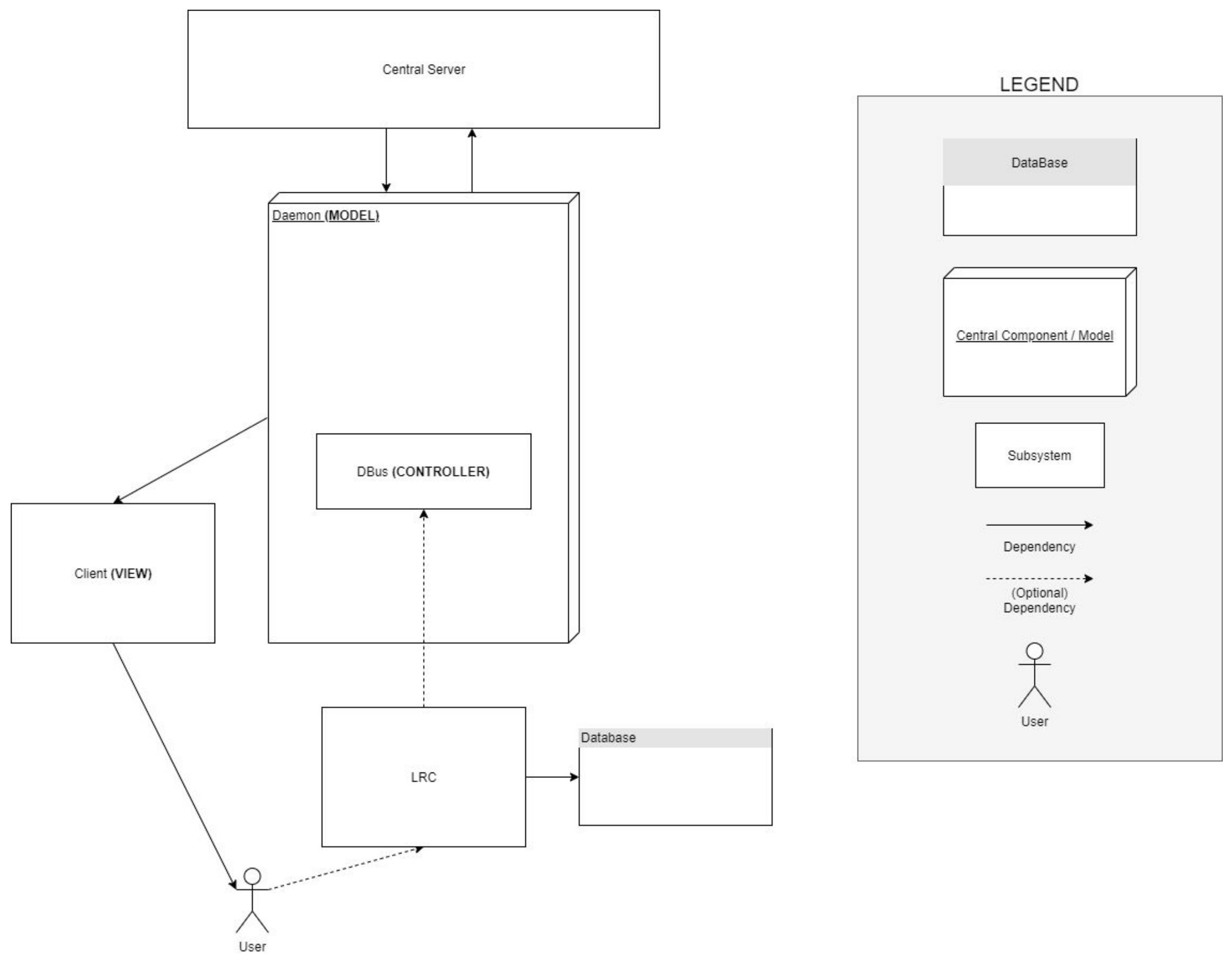
Developed and maintained by the Montreal based Savoir-faire Linux, Jami is a distributed peer-to-peer and SIP based communication platform that runs entirely on individual devices; it is currently supported by Linux, Microsoft Windows, OS X, iOS, and Android. The decision to make Jami fully distributed was ethically motivated by the inclination to protect user privacy and has a number of practical advantages including scalability, network resiliency, and autonomy. Although Jami's distributed architecture comes with its own challenges, the design proves to effectively support instant messaging, call, and video chat features.

Before breaking down its conceptual architecture and subsystems, it is useful to address the stakeholders associated with Jami, which mainly consists of end users and the project build team. End users include individuals interested in Jami for personal use and organizations who use Jami internally. The project build team includes managers, company liaisons, and developers/engineers. The business model sustaining Jami as free and open source software relies mainly on selling professional consulting services to organizations who use Jami internally, thus eliminating the need to charge users, monetize user data, or rely on donations.

The conceptual architecture derived in this report is extracted from the documentation found on the Jami website and Github repository. 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. Users have the option to first access a central server to retrieve information about the user they wish to connect with, which would follow a client-server design pattern (however this is optional). The Dameon is the central component handling the business logic of Jami and peer-to-peer communication is supported by OpenDHT, which itself is composed of the Nameserver, Bootstrap, and TURN server. These subsystems are further discussed in subsequent sections, followed by a study of concurrency and data flow. The functionality and interaction between key components of Jami are illustrated in two use cases: logging in and adding a contact, and making a call. We conclude with a discussion of the derivation process and lessons learned.

2. Conceptual Architecture

In the previous section, we proposed the conceptual architecture of Jami follows a model-view-controller design pattern driven by a peer-to-peer communication network. The key components are illustrated in the box and arrow diagram below, where the central server (which includes the OpenDHT) represents the peer-to-peer network and the Daemon, DBus, and Client correspond to the MVC architecture. An intermediate component, the LRC, is also included in the diagram. The LRC acts as a middleware for all clients, with the exception of the Android client. It interacts with the Daemon through the DBus API, and is the only component having access to the local database.



The optional central server is excluded from the conceptual architecture above, as Jami is intended to be able to function as a serverless communication platform.

3. Subsystems

3.1. Daemon

The Dameon is the central component handling the business logic of GNU Jami; it implements the logic for Ring, interacts with OpenDHT, pjsip, ffmpeg and other libraries, and executes the application protocol. The Daemon does not interact with users, but is involved in every command. Clients are usually implemented on top of the daemon and interact with one of its many APIs, which consist of 4 managers and an instance file.

The CallManager interface manages calls and conference related actions; the Ring daemon supports multiple incoming/outgoing calls by generating a unique callID to identify specific calls, any actions addressing a call must address the method through this unique ID. The ConfigurationManager handles setting, preferences, and other configuration related tasks. The PresenceManager tracks the presence of contacts. The VideoManager handles renderers and manages video devices. A client being initialized is registered against the core through the Instance interface, which counts the number of such clients.

The daemon is broken down into four APIs. The DBus API (used by the LRC), the JNI API, the Node JS API, and the REST API (the Node and REST APIs are currently not up to date). A python wrapper which uses the DBus API is available to make interacting with Ring easier.

3.2. LRC

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; it does not, however, interact with the Android client. The LRC is written in QtCore, which is a C++ module used for developing GUIs and multiplatform applications. It consists of a number of major interfaces and classes.

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.

3.3. Clients

The purpose of the client layer is to formalize the use-case scenarios on GNU/Linux using Gnome, Windows, Mac OS, Android, Android TV, and iOS. It operates on top of the daemon, and it is possible to implement your own client using one of the many APIs, including REST, dbus, libwrap, and JNI. As per the high-level architecture diagram, all platforms interact with the controller through the LRC, with the exception of Android.

The purpose of the client is providing a UI through which users can interact with Jami and access its capabilities. Some of the key features of the client are text messaging and video calling across all 6 supported platforms, as well as audio calling capabilities for all except Android TV. All offer plugin support as well, with the exception of iOS and Mac OS.

3.4. Servers

3.4.1. OpenDHT

The OpenDHT is the component of Jami responsible for allowing the software to operate as a peer-to-peer communication platform. DHT stands for Distributed Hash Table, and its purpose is to build and maintain a hash table of key-value pairs which is saved across all the different peers of the Jami network. Built on the bones of the BitTorrent DHT, on a C++14 Kademlia implementation, which already has a very efficient method of distributing the hash table responsibilities across the participating peers, Jami adds additional functionalities, as follows.

- 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. This avoids the need to poll for changes every few seconds.

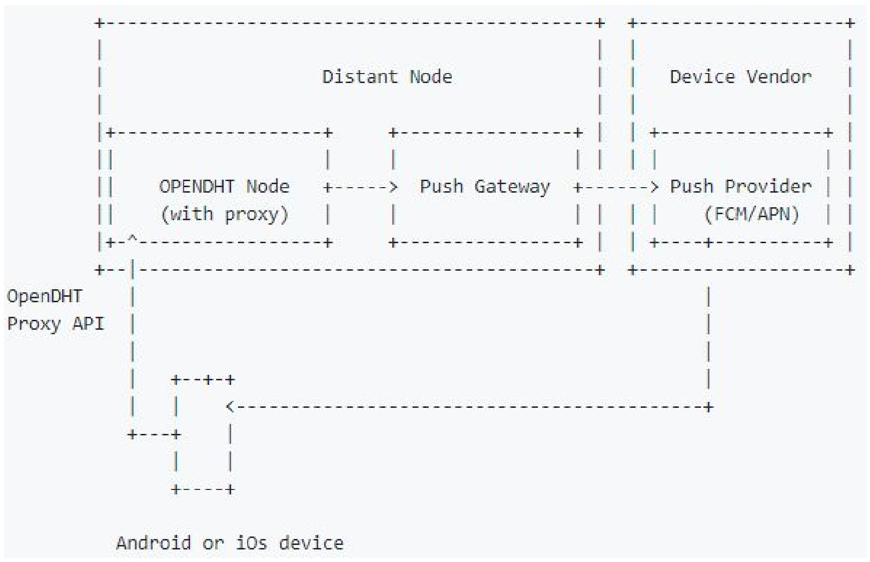
- A value ID to distinguish different values stored under the same key (which was not needed when values were just IP addresses).
- A value type, attached to every value, that allows different expiration, creation or editing policies to be applied to different values. For instance, a value type may specify that values expire after 5 minutes, or that they can only be accepted or edited if they are signed.

OpenDHT provides a couple of key functional abilities which give Jami its advantages. First, it is designed to work well for very large networks, where each peer can be a very small device. Similarly to BitTorrent, this allows for small devices to collaborate and build a very resilient network, which has high resilience to disruption. This achieves the basic purpose of Jami, which is to allow for a stable, scalable peer-to-peer communication platform that has no reliance on a central server.

Another purpose of the OpenDHT is to allow Jami devices to find each other on the Jami network without the need of IP addresses, which are usually dynamic and can change over time. Replacing the need for IP addresses, OpenDHT provides a way to identify devices on the network with a unique ID and to retrieve their IP address without knowing it beforehand, allowing them to communicate reliably even if the IP address changes. However, it needs to run continuously in the background to keep connections to the network open and listen to changes for alerting the user of a new call or message. This can be draining on Jami device batteries. Helping to alleviate this problem are aforementioned listen and get operations, which allow peers to 'listen' less often, and use up less battery life.

Next, let us cover contact management on the DHT. Announcing the presence on the DHT can be done by announcing the device on the network using a value containing the device hash. In order for a peer to check if another peer is present, the reverse process happens - you get the value at the hash corresponding to the Ring ID and retrieve the DeviceAnnouncement on such hash.

The OpenDHT proxy is a server that listens for changes on the OpenDHT network and sends notifications to Jami devices through Apple's or Google's push notification servers. No personal data is ever sent this way, not even in encrypted form. The proxy only alerts your devices that something new happened and the rest is done by retrieving the information through the OpenDHT network.



An optional public-key cryptography ("identity") layer exists on top of the DHT and uses the value type system. When used, this layer allows one to put signed data on the DHT. Signed values can then only be edited by their owner and are retrieved from the DHT and automatically checked. They are then presented to the user if the signature verification succeeds.

The identity layer also publishes a (usually self-signed) certificate on the DHT that can be used to encrypt data for other nodes. Encrypted values are always signed, and the signature is part of the encrypted data, which means that only the recipient can know who signed a value. For this reason, like standard non-signed values, encrypted values can't be edited.

3.4.2. Nameserver

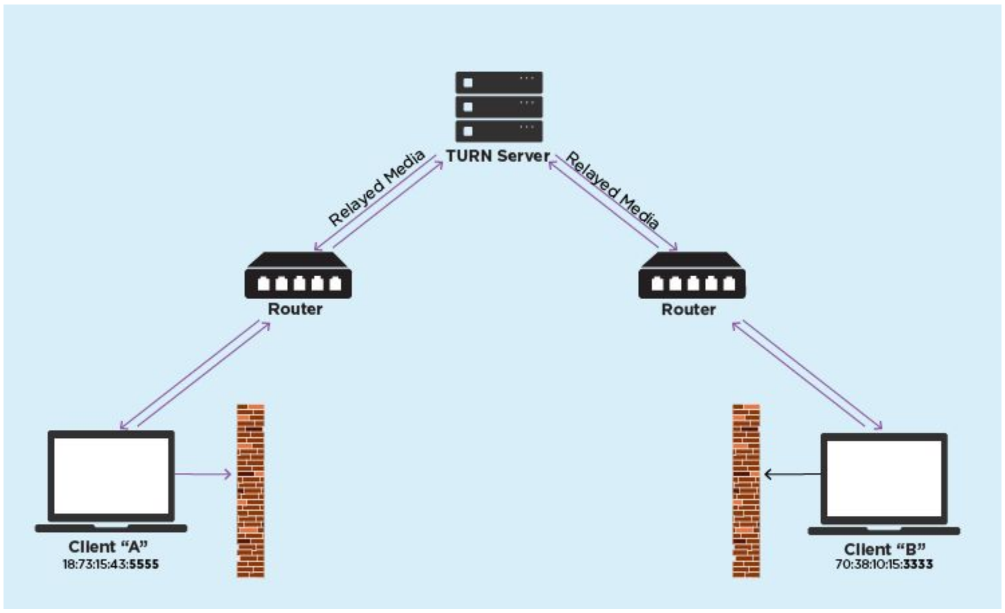
All Jami accounts are given a 40 character long unique ID upon creation. Since this ID is impractical for humans to remember, the nameserver allows a user's ID to be associated with a personal username. This makes it possible to search for and befriend another person by their username as opposed to their ID, although you could still search for the same user by searching their ID. For this username system to work through the Jami application, the nameserver is necessary. No personal data is ever sent through the nameserver, and Ethereum blockchain smart contracts are used to handle Jami usernames while keeping Jami as distributed as possible. Users can also configure the nameserver in the advanced Jami parameters to use their own if they wish.

3.4.3. Bootstrap

Jami's architecture is a distributed, peer-to-peer network. Each device running Jami acts as a node in this network. However, in order for devices to become a part of this network and make connections to other peers, they need to know at least one other peer beforehand. New devices are configured to join the network through the bootstrap when connecting for the first time. The bootstrap is an OpenDHT node that is kept online and maintained by Jami at all times, whose default value is bootstrap.jami.net. No personal data is ever sent through the bootstrap, and Jami gives users the option to connect to and run their own bootstrap server in the advanced settings of the application.

3.4.4. TURN

The TURN server is only used when a peer-to-peer connection cannot be established to relay data between users. Jami uses a technology called ICE to connect devices when users communicate. If the connection through ICE fails, a TURN server is created. One common example of when this might occur is when one or both users involved in a data relay are behind a firewall. The firewall makes it difficult to establish a direct link to the user, so a TURN server comes into play as a relay. All personal data that is sent through the TURN server is encrypted from end to end and is never stored elsewhere than on the end-users' devices.



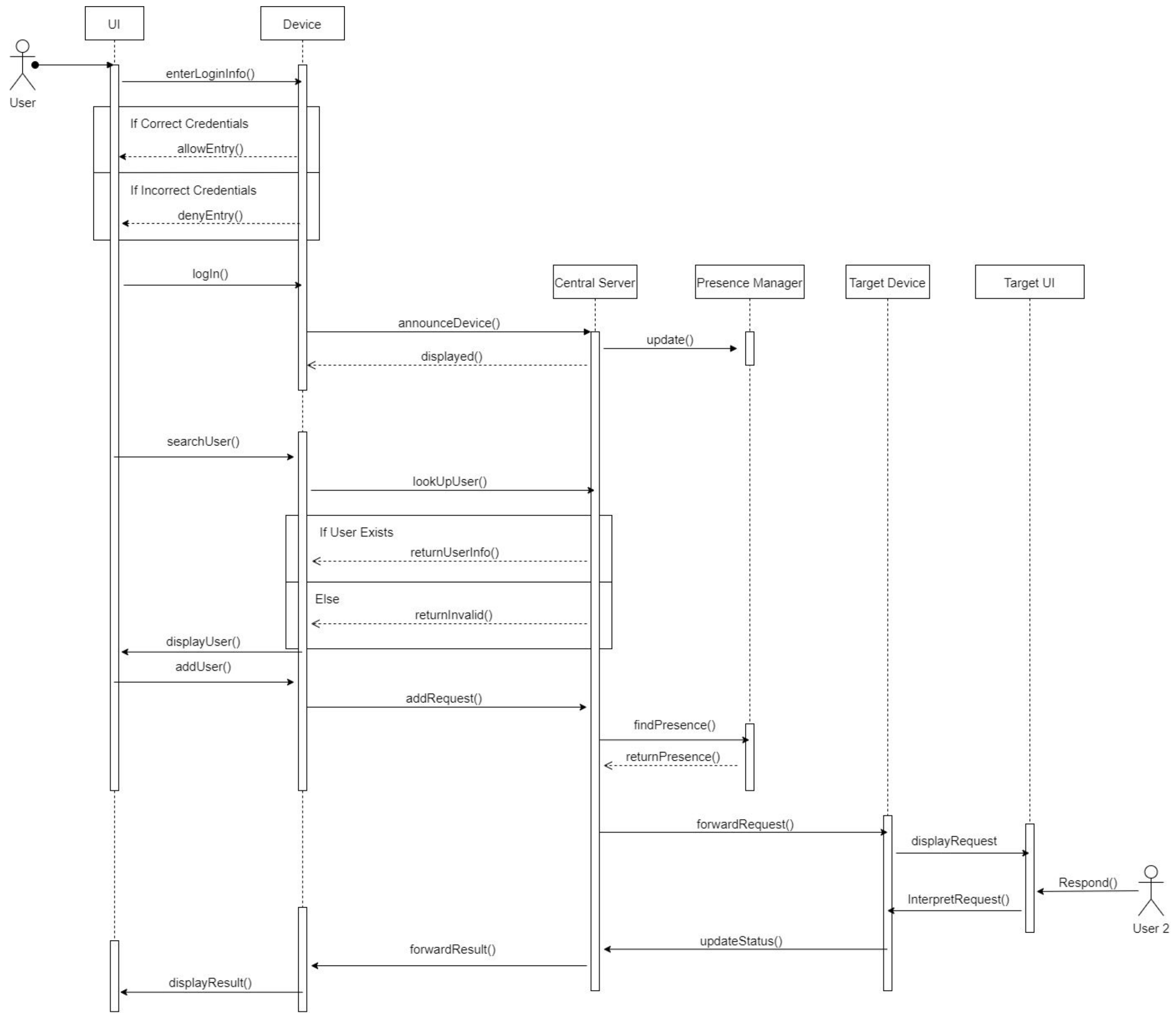
3.5. Plugins

The Daemon of Jami contains a JamiPluginManager class that can perform installs, uninstalls, loads, unloads, preference editing, and can control the usage of plugins. The plugin system inside of Jami exposes different APIs that can be used by plugins. Examples of these include the Chat Handler API and the Media Handler API. The Media Handler API allows for modifications to audio and video streams within Jami, and can be used to build green-screen and face-filter plugins to list a few examples. Jami plugins always run directly on the user's devices. Plugins can be composed by one or multiple media and chat handlers that are responsible for attaching/detaching a data stream from Jami and a data process. Each handler represents a functionality that can be totally different between them or can be a modified version of the same core process. Similar to how the Daemon does not know what is being done by the LRC or to the clients interfaces, the effects of a plugin on video, audio, and chat messages are unknown to the plugin itself. A plugin SDK is available to help encourage the building and packaging of native, cross-platform, third-party Jami plugins.

4. Use Cases

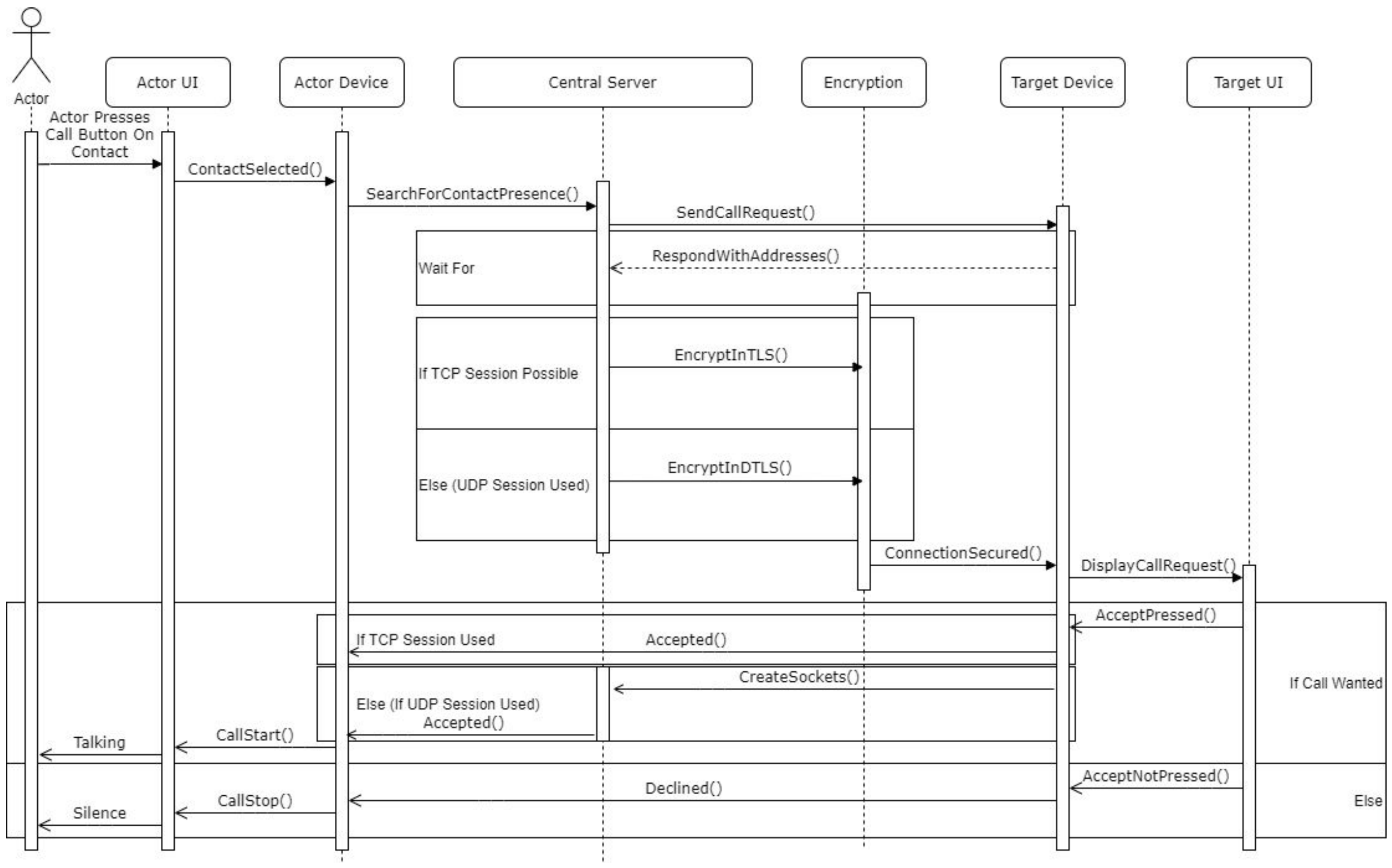
4.1. Logging In And Adding A Contact

For our first use case, we decided to show the process where a user logs in to Jami and adds a contact. First, the user must log in by entering their correct credentials. If their information is incorrect, then they are denied entry and must try re-entering their username and password. Then, after the user successfully logs in, the device automatically sends a signal to the central server indicating that the device is online, and can be found and contacted by other devices. When a user is searched for, the central server is checked to see if there exists a user matching the query. If there is a match, then that user's information is displayed, and a friend request can be sent. The central server checks for the presence of the target user, and forwards the friend request to the target user's device. The target user can then block the user who sent the request, or accept or deny the friend request. The device sends the target user's choice back to the central server, which updates the relationship status between the two users and forwards the result back to the original user. The result is then displayed, which terminates this use case.



4.2. Making A Call

Our second use case is making a call with Jami. We assume the user is logged in and already has the call recipient added as a friend, since this is required to make a call with someone. Making reference to the "Let's do a call" documentation, 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 contact can then choose to answer or decline the call. In the case of accepting while using UDP, an ICE transport negotiates the creation of 4 new sockets - 2 for audio, and 2 for video. To keep this use case conceptual, the DHT and ICE have been combined into one "central server".



5. Concurrency

Jami uses concurrency to perform the main app functions, and also to improve performance and provide greater app reliability. By allowing many different sections of the software to be active at once, Jami as a whole can be performing several tasks at once, all independent of each other, speeding up the app and allowing for a multitude of different tools to be at the user's disposal.

Jami has many different subsystems that all perform different tasks concurrently, which are connected through the Daemon. The subsystems and their concurrency have been broken down as follows.

Jami operates on a combination of a peer-to-peer, model-view-controller, and (optionally) client-server architecture. While connections are made peer-to-peer, users have the option to first access a central server to retrieve information about the user they wish to connect with. In this central server is a very large number of nodes, which each store information about a single user. Users can access some of the information on any given node, based on their level of authority with respect to the node. These nodes do not just act as a tool to store information, but rather as entities themselves. They are interconnected and query information from each other, all while running a separate thread listening for requests from a user's device so that they can process the user's request in a timely fashion. Nodes also have another concurrent thread running, constantly listening for calls from other nodes or other users, enabling them to send the required information immediately and directly to a user in the form of a push notification. Obviously, the nodes themselves are running concurrently, as well as the sockets used to connect the nodes to each other. This structure is essential for app function, and having nodes that are interconnected greatly increases performance when a user accesses the central server.

Jami also uses Internet Chat Relay (IRC) to assist in connection and facilitate messaging between two users. Each device creates a local entity, which runs constantly in the background, listening for incoming notifications from nodes. Messages sent by a user go through the central server, through the nodes, and down to the recipient user, where the local IRC entity deals with the message accordingly (allows it or blocks it, displays it, etc.). This form of concurrency is an example of when having a subsystem running concurrently with the rest of the software is essential for the function of the software.

Group chats (two or more users involved) in Jami are called "Swarms", and operate in a peer-to-peer style. In order for users to connect, one user must act as the host, and others as clients connecting to the host. Users connect via sockets, which are all created as separate threads in the application that will run concurrently with the rest of the app as long as the swarm is active. There are also backup sockets that are created, which connect to the central server rather than the host in case a connection error occurs. These two types of sockets running concurrent to each other is an example of where performance is traded off for reliability and security, as information is stored locally (though the performance impact for something this small scale is negligible). The main drawback of peer-to-peer architectures many times is performance, as one or more users often rely on other user(s) for a stable connection. Adding in an extra type of socket solves this problem, and if the host disconnects, or there are other connection issues, the users will switch over to connection via the central server.

Similar to how swarms operate, when users call each other, sockets are created which facilitate a peer-to-peer connection between the two users. On each device, two sockets will be created: one for audio, and one for video. The data transfer between these two sockets must be synchronized (i.e. running in parallel), as it is important for audio to match what the video is showing.

Users must also be able to transfer files to each other, in addition to being able to chat with and call each other. Using Internet Connectivity Establishment (ICE), users are able to create special sockets that allow fast connection with each other. It is through these special sockets that users are able to share large files between each other. These sockets running concurrently speeds up performance, meaning fast file sharing. However, this does come at the cost of stability, because direct connection involving file transfer is likely to be blocked by some firewalls or Network Address Translators (NATs), which make it hard for users to connect with each other. In the case where connection fails, a Traversal Using Relays around NAT (TURN) server is created, which will run in the background, facilitating connection between the two users.

All of these subsystems employ the concept of concurrency in different ways (creating sockets that run concurrently, having threads within a structure running concurrently, or having an entity running constantly in the background) in order to ensure that the app runs as intended, while also increasing performance, reliability, and providing additional security.

6. Data Flow

Jami utilizes a variety of APIs to exchange data through the different interfaces used to power it. The first of which is the OpenDHT which offers the ability to store user keys up to 64KB. It is IPv4 and IPv6 compatible and offers a distributed key-value data store.

Data flow begins in the UI when the user opens the Jami application. The user has the ability to create an account, create a rendezvous point, import from another device, or import from an archived backup. Below is an outline of the data flow for the process of creating and deleting accounts.

Creation of an account involves sending a signal from accountsChanged. Following the emission of this signal, the respected client will update its internal structure along with other methods in the ConfigurationManager. Ring accounts are simply represented by files within a gzip archive. The archive will only be encrypted if a password is provided upon account creation. The contents of the archive are as follows: (i) The private key ringAccountKey and certificate chain ringAccountCert, (ii) generated CA key (ringCAKey), (iii) relocated devices (ringAccountCRL), (iv) the ethereal private key (ethKey) for the device which is only used when registered on ns.ring.cx , (v) the contacts, and (vi) the account settings.

Deletion of an account is rather straightforward as the keys are only on the device. Thus, deletion of the keys results in deletion of the account itself. Only the username is stored outside of the device (it is stored on the name server). The removal of this info is dependent on which nameserver is in use (e.g. it is not possible with https://ns.ring.cx). Similar to account creation, when an account is deleted the signal accountsChanged is emitted and the client will update its internal structure following this emission.

Announcing a client's presence on the network allows the device to be recognized as present. All this requires is the value containing the device hash on the hash corresponding to the Ring ID.

Checking if a device is present involves the reverse process just mentioned. We get the value at the hash and retrieve DeviceAnnouncement.

Next is the data flow for creating a swarm (group chat) in Jami. To create a swarm the user must first initialize a local Git repository. The user must then add their public key, device certificate, and CRL. The user then announces his presence to the other devices and sends an invite to join the swarm which is sent through the DHT.

7. Division of Responsibility Among Developers

Since Jami's conceptual architecture consists of a clear set of modules, assigning individual modules to participating developers would be an effective division of responsibility. Front-end developers may clearly focus on the clients, and back-end developers can be assigned to individual subsystems within the Daemon, the central server, and the LRC. Those with significant experience in the network domain, particularly peer-to-peer networks, would do well to focus on the OpenDHT and its associated components. In addition, a database engineer would be needed to build the SQLite database used by the LRC. A lead software architect would be needed to oversee the development of all components, and ensure proper design patterns are followed.

8. Derivation Process

Our group worked on this objective by first assigning each member to a viewpoint, keeping organized by using Google Drive to share our research with others. We individually researched the architecture of Jami, and each attempted to create a model of the conceptual architecture based on our understanding. We then had a meeting to discuss the points we agreed and disagreed on, construct a template for our conceptual architecture report, and split roles based on the viewpoints we researched. By splitting up the research component everyone was able to have access to all the findings we established as a group. Afterwards, we had a number of final meetings to compare and finalize our conceptual architecture and report.

9. Lessons Learned

Throughout the entire process of deriving a conceptual architecture for Jami, the team encountered numerous obstacles, the most significant having been the rather incomplete and out-of-date documentation on Jami. The documentation does a good job providing a brief overview of the features of Jami — contact management, account creation/deletion, and doing a call — but fails to provide a deeper description without showing the actual code. Some features of Jami such as call recording and group calls do not even have documentation yet. In order to combat this obstacle, the team needed to perform a significant amount of individual research and analysis in order to gain a complete understanding of the components and architecture of the system. This process taught each member of the team how to effectively manage their time between team meetings, individual research, and work sprints. Prior to beginning the project the team had expected to be provided with deep, detailed documentation. So, having faced a situation quite contrary to this enabled the team to take away a valuable lesson learned. Another lesson the team learned came from working in an academic team from a remote position. A vast majority of courses offered in the Queen's Computing program do not provide students a way to work collaboratively on projects. This, coupled with a remote semester, was quite a large change for the entire team. One of the largest challenges in this environment was communication. When working in a team, understanding each member's sentiment and feelings is critical to having an effective team dynamic. Communicating this remotely through Discord can be especially challenging. Fortunately, we had various team members step up as leaders to help communication between everyone in the team. This leadership allowed the team to function more effectively as we were ultimately able to overcome the initial communication obstacle.

10. Conclusion

Summarizing the core aspects of the conceptual architecture of Jami from existing documentation, we have found that its high-level structure is that of a model-view-controller style, with a peer-to-peer architecture style used for the communication network. With the help of the OpenDHT server, Jami offers users the ability to exchange text, audio, and video calls without the need for a centralized server, eliminating the need for the company to ever store users' personal data, and addressing a growing demand in the tech space for safer, private communication methods. The Jami architecture further has the benefits of scalability and network resiliency, allowing the Jami user network to expand easily and maintain its reliability.

In the following report, we will shift our focus from the conceptual architecture described in the Jami wiki pages, onto the concrete architecture that is to be found in the actual source code of the current Jami software. In doing so, we will gain a deeper understanding of how Jami was ultimately implemented in practice, and discover how the end product differed from the original plans.

11. References

Jami-ing out. (n.d.). Retrieved February 22, 2021, from https://cisc322.github.io/Jami-ing-Out/

Internet relay chat. (2021, January 16). Retrieved February 21, 2021, from https://en.wikipedia.org/wiki/Internet_Relay_Chat

Introduction · wiki · savoirfairelinux / jami-project. (n.d.). Retrieved February 22, 2021, from https://git.jami.net/savoirfairelinux/ring-project/-/wikis/technical/0.-Introduction

Naggar-Tremblay, F. (2019, October 09). Why is Jami truly distributed? [Web log post]. Retrieved February 21, 2021, from https://jami.net/why-is-jami-truly-distributed/

Naggar-Tremblay, F. (2019, July 15). Practical advantages of Jami's distributed architecture. Retrieved February 22, 2021, from https://jami.net/practical-advantages-of-jamis-distributed-architecture/

Naggar-Tremblay, F. (2019, June 07). Challenges associated with Jami's distributed architecture. Retrieved February 22, 2021, from https://jami.net/challenges-associated-with-jamis-distributed-architecture/

"Together", the new version of Jami and a new step forward [Web log post]. (2020, October 16). Retrieved February 21, 2021, from https://jami.net/together-the-new-version-of-jami-and-a-new-step-forward/

Savoir-faire Linux. (n.d.). Retrieved February 21, 2021, from https://github.com/savoirfairelinux

Savoirfairelinux. (n.d.). Savoirfairelinux/opendht. Retrieved February 22, 2021, from https://github.com/savoirfairelinux/opendht/wiki/Push-notifications-support

Steeves, T. (n.d.). WebRTC NAT Traversal methods: A case for Embedded turn. Retrieved February 22, 2021, from https://www.frozenmountain.com/developers/blog/webrtc-nat-traversal-methods-a-case-for-embedded-turn