"Alexandru Ioan Cuza" University of Iași
Faculty of Computer Science

Bachelor's Thesis

Resource Streaming using a Peer-to-Peer Architecture

proposed by
Marian-Sergiu Nistor

Session: *June - July, 2021*

Scientific coordinator
Prof. Dr. Lenuța Alboaie

"Alexandru Ioan Cuza" University of Iași
Faculty of Computer Science

Resource Streaming using a Peer-to-Peer Architecture

Marian-Sergiu Nistor

Session: *June - July, 2021*

Scientific coordinator
Prof. Dr. Lenuța Alboaie

Abstract

There were several historical instances in which breaches in centralized systems put user data confidentiality at risk. Therefore, there is an acute need for transitioning towards decentralized platforms which ensure fairness in the created ecosystem and prevent the occurrence of privacy-related violations. This paper proposes an architecture for a Peer-to-Peer resource streaming framework, which does not impose a hierarchy over its users, encouraging the establishment of an unbiased network in which the clients collaborate in the decision-making processes. Security-wise, the projected architecture's lack of centrality reduces the possibility of a total outage, due to the absence of any Single Points of Failure. The proposed approach considers locality-aware distributed hash tables (LDHT's) for storing the network state on each member node, and makes use of a custom implementation of the epidemic protocol for propagating information throughout the network, granting that each client's network state copy is consistent and up-to-date. This promises to address the majority of the issues that were stated previously.

Keywords: decentralized network, Peer-to-Peer network, resource streaming, locality-aware distributed hash tables (LDHT), epidemic protocol

Table of Contents

List of Figures

Introduction

1.1 Problem Statement

Up until now, business models that target the field of resource streaming were mostly implemented using centralized approaches, due to the ease with which the data could be maintained and transferred. This method proved to be facile, as a decentralized alternative would require a more complicated workflow and a set of advanced security rules, resulting in a lower development velocity.

However, in the past, several downsides surfaced over a number of episodes revolving around the concept of centrality. An important factor to consider is the vulnerability to cyber-attacks. The idea of a primary decision-making authority determines the occurrence of a Single-Point-of-Failure (SPoF), making the entire system prone to collapse when being targeted by malicious entities. According to Cisco's Annual Internet Report (2018–2023) [1], the total number of Distributed-Denial-of-Service (DDoS) attacks is projected to reach about 15.4 million in 2023, almost twice as many as there were reported in 2018 (7.9 million), as shown in Figure 1.1. If a system that is heavily reliant on the availability of an individual component becomes the target of such intents, there is a high chance of complete outages occurring. If the subject is a large-scale, high-traffic platform, this event could possibly result in enormous financial losses, affecting both the company that is maintaining the service and its users.

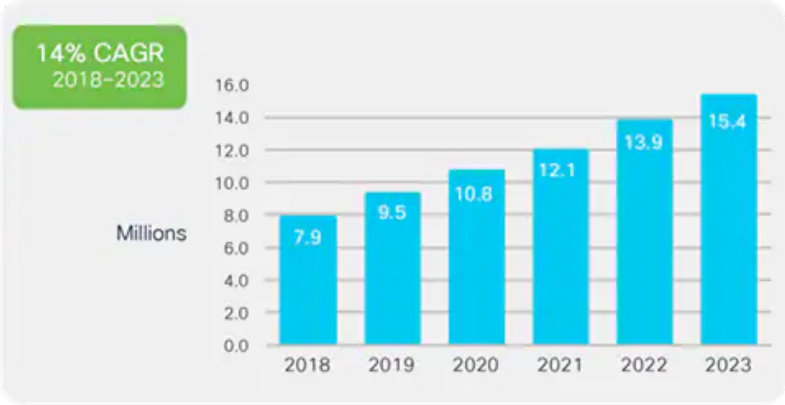

Figure 1.1: Projected number of DDoS attacks during 2018-2023 [1].

Another concern is represented by the provider's capability of intentionally disclosing private information to a third-party, defying specific confidentiality considerations. Empowering a singular agent to control the entirety of the data comes at the risk of that agent violating privacy regulations. There were multiple instances when companies acquired or handled user data in unjust ways. Google faced multiple legal penalties, as a consequence of the Google Street View Privacy Case (2007), the Google Privacy Policy Case (2012), the Google Buzz/Safari Case (2012), and the Google Spain Case (2014) [2]. Facebook was also held accountable during the Safe Harbor Case (2015) and the Facebook Cookies Cases (2014, 2017, 2018) [2]. In order to prevent such incidents from recurring, in 2016, the European Parliament and the Council of the European Union developed the General Data Protection Regulation (GDPR), replacing the Data Protection Directive. The GDPR acts as a legal framework, covering the subject of personal data protection in the European Union and the European Economic Area and it is active since May 25, 2018 [3].

Finally, a central power would be able to deliberately make biased decisions, inclining towards unfairly favoring a certain group. This goes against the idea of equity in a network-based ecosystem, as every participant is supposed to have an equal chance and priority to the services. One instance in which this matter emerged was during 2010-2017 when the European Commission conducted an investigation into Google Shopping. The inquiry concluded that Google abused its dominance as a general search engine by giving an illegal advantage to its own shopping platform over its competitors. Google was also accused of removing the UK-based search engine Foundem from the Google Search results for three years [4] [5].

1.2 Project Motivation

Considering the aforementioned facts, there is a significant need for a resource streaming service model that can fulfill the security and impartiality necessities, while being scalable and highly available. By decentralizing the business logic, the SPoF issue is eliminated, thus dismissing the possibility of a DDoS attack potentially disrupting the system. Scalability is granted by the fact that the data maintenance and transfer tasks are being handled by several actors, enabling the network to handle greater computational loads, as it can grow indefinitely. As a result of the fact that all the members share the governance responsibility, no singular participant or group possesses the ability to bypass or interrupt the decision-making process, since every action taken occurs subsequent to an agreement between a set of peers, therefore assuring fairness when settling a resource allocation conflict.

However, an issue to consider is the higher request processing time implied by the voting process. This concern can be addressed by conveniently choosing voters, based on geographical localization and latency, if the business rules allow such selection. Decentralized systems can also benefit from adopting edge computing. By placing the compute nodes closer to the end-user, the response time can be drastically reduced. While most devices used in edge data centers are of IoT nature, the introduction of 5G enhances the capabilities of this particular computing paradigm, by providing the fastest cellular network data rate with very low latency [6]. As indicated in Figure 1.2, by 2023, 5G speeds are expected to reach an average value of about 575 Mbps, 13 times higher than the average mobile connection [1].

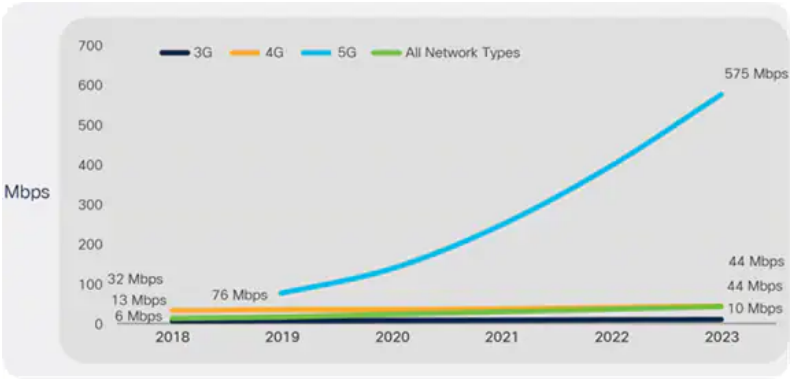

Figure 1.2: Projected global mobile average speeds by network type during 2018-2023 [1].

1.3 Project Goals

The goal that the project is trying to achieve consists of implementing a proper resource streaming platform model which does not rely on a central authority, therefore meeting the security and equity criteria stated in section 1.2. No actor is supposed to benefit from any privileges or depend on the choices made by a sole member. Instead, the decisions are meant to be made collectively, each node having the responsibility to vote for a specific outcome. Additionally, pure system decentralization is achieved using a Peer-to-Peer architecture. Also, the network can prevail without any external intervention or guidance. Ideally, the platform would not suffer from any performance issues, when compared to an equivalent centralized implementation.

So as to ensure that the network would not go inactive, the framework implements a particular type of infrastructure node, conveniently called, a "supernode", which acts as a network constituent that does not take part in any resource streaming processes. Its role is a passive one, preventing the ecosystem from becoming completely inactive. This entity type can only participate in the initial connection phase for a new node, redirecting the requester towards the desired subnetwork, or in the net graph DHT synchronization step, both announcing and receiving changes in the system's state. A more detailed overview of this distinct type of node is provided in subsection 4.1.1. Multiple supernode instances can be simultaneously active, at any given time.

Technical Prerequisites

This chapter focuses on explaining the technical aspects that are being implied by the proposed framework. The system implements the resource streaming mechanism in a fully decentralized manner, using a Peer-to-Peer network architecture, which is analyzed in section 3.2. In order to swarm the network with the required data, the system implements the epidemic protocol. An overview regarding the implemented method is provided in section 3.3. Each system constituent holds information regarding the network's arrangement, using a custom data structure. So as to keep the information consistent between each client's copy, the framework makes use of DHT's (distributed hash tables). The custom data structure and the means of synchronization are described in section 3.3.

2.1 Peer-to-Peer Data Streaming

A Peer-to-Peer data streaming system is a decentralized apparatus that serves the goal of distributing a provider's data to several receivers. The main difference between this system and a Peer-to-Peer file distribution system lies in the way that the data is being processed by the clients. The prior uses the $process-while-downloading$ approach, while the latter is based on the $process-after-downloading$ method. Therefore, each architecture type implies a distinct set of constraints, while a streaming service client must provide the information to its neighbors in real-time, frame by frame [7]. The data propagation algorithm must ensure that the frames arrive in chronological order, while also preventing data loss.

The system's topology is constructed based on the mesh network model [8] [9], which

enforces the direct connection between the network constituents, without the establishment of any hierarchical relationships or the use of any central authority.

There are several aspects that must be considered when designing a proper Peer-to-Peer data streaming framework. A feature to consider is the system's scaling capabilities [7], which is intrinsically implied by the mesh network model. A proper implementation of such system topology would ensure that the system is highly scalable [8]. Furthermore, the network model would also guarantee that no member would exhibit server-like behavior [7], due to the absence of hierarchical relationships. Another aspect in designing such system consists of the fact that, ideally, all the peers should have an equal bandwidth contribution to the ongoing network processes [7]. Therefore, the system must provide a load balancing mechanism, to avoid the discriminatory payload distribution between the network members.

2.2 Epidemic Protocol

The epidemic protocol provides a mechanism for sequentially flooding a Peer-to-Peer network with the required information, ensuring that the data reaches all of the constituent nodes [10]. A notable example of a tool that implements this protocol is the InterPlanetary File System (IPFS) [11], which is detailed in section A.1. The approach of the epidemic protocol implies the fact that each network member must propagate the messages received from its parent nodes to its child peers, after confirming its validity. As long as the information is legitimate and the process is not compromised by any errors, every node will eventually acquire the data, in a finite amount of time.

In Peer-to-Peer applications that require the maintenance and synchronization of large routing tables, the epidemic protocol is used to ensure the consistency of the routing table instance held by each client [10] [12]. This action is vital for such system, as every node is expected to have an up-to-date overview of the network state.

In the context of the proposed resource streaming system, the epidemic protocol is used for both synchronizing the distributed hash table, which holds information regarding the

network state, and for propagating the resource frames throughout a specific subnetwork.

2.3 Distributed Hash Tables

A distributed hash table (DHT) represents a table that is synchronized across all network members. In the context of decentralized applications, the DHT's provide a method for storing the network state on each client node, acting as a routing table [13] [12]. Consistency between each member's DHT instance is achieved with the use of network flooding protocols, such as the epidemic protocol, as specified in section 2.2. An example of a DHT implementation is the Kademlia DHT [14] and its variations (e.g. S/Kademlia DHT [15], Mainline DHT [16], Azureus DHT [16]), which are used by decentralized file systems, such as the InterPlanetary File System (section A.1) [11], or Peer-to-Peer content distribution frameworks, such as BitTorrent (section A.2) [17].

Traditionally, the DHT does not hold any information regarding locality for the network constituents. Therefore, from a topological point of view, it provides no mechanisms for latency minimization, as new clients are not being peered with optimal neighbors [13] [18]. The locality-aware distributed hash table (LDHT) represents an extension for the DHT, that stores information regarding the relationships between the network peers, therefore providing means of locating each node, relative to all the network constituents [13]. As a result, each new member can be peered with a set of favorable neighbors [13].

The proposed system takes node locality into consideration, storing information regarding inter-node relationships, each member's appartenance to a particular subnetwork, and the connections formed between the network subcomponents. The data stored in each net graph DHT instance is described in section 3.3.

Network Design and Implementation

This chapter will go over the technical details of the resource streaming framework, providing an overview of the network architecture, the framework structure, and the implementation aspects.

3.1 Analysis

As stated in section 1.3, the objective for the proposed system consists of providing a Peer-to-Peer resource streaming platform, with high scalability capabilities, which does not rely on any central authority and does not favor any particular members of the network. All constituents are expected to participate and collaborate in the decision-making processes. One of the system's most crucial goals is to maintain a high degree of fairness. Therefore, in addition to the idea of avoiding privileging certain nodes over others, the system is also expected to make sure that each member has an equal contribution, by distributing the payload as uniform as possible, as mentioned in section 2.1. This is achieved using load balancing mechanisms based on heuristics, which are described in section 3.3.

The routing mechanism is achieved using a custom LDHT [13] [18] implementation which considers a node's relative location in the network graph. This design is inspired by the Kademlia DHT [14]. Variations of the Kademlia DHT were previously used with success in the industry, specifically for Peer-to-Peer content-addressed file transfer applications. As mentioned in section A.1, the InterPlanetary File System [11] uses a secure extension for the Kademlia DHT, called the S/Kademlia DHT [15]. BitTorrent uses two extensions for the initial model, called Mainline DHT and Azureus DHT [16],

which are described in section A.2. The proposed framework's routing table provides means for identifying optimal peers for new members, based on the client's requested resource. Further details are provided in section 3.3.

The epidemic protocol [10] [12], whose theoretical implications are stated in section 2.2, is used for both synchronizing the DHT instances between the network constituents, and for propagating the resource frames throughout each subnetwork, sequentially. A similar approach to achieving inter-node routing table synchronization using the epidemic protocol is used by the InterPlanetary File System [11], as described in section A.1. Each node has the responsibility of propagating a message received from a trusted parent node to its immediate peers, as long as the data is valid. Depending on the request type, the message can either flood the entire network, or just a particular subnetwork. The exact mechanism for information propagation through the network is described in section 3.3, and the implementation details are provided in section 3.5.

3.2 Network Architecture

The network architecture was designed in a way that is feasible for synchronizing the network state between all of the members. The system follows the mesh network model, meaning that the nodes are connected directly to as many peers as possible, without following any hierarchical mapping, as specified in Figure 3.1. Upon the addition of new constituents, the tracker nodes attempt to strengthen or reestablish the connectivity within the network and the subnetworks, redirecting the new members to appropriate peers and prioritizing those who have a lower number of neighbors.

Because of the fact that the system is decentralized, therefore lacking a central authority, each node plays both the role of a server, for a subset of its peers, and the role of a client, for the others. The only exceptions are the supernodes, which only act as servers. The nodes can be split into three categories, considering their role in the ecosystem:

- Infrastructure nodes (supernodes), which have the purpose of keeping the network active, even if there are no users connected. Their only role is to ensure that new nodes can connect to default trackers, in the absence of other members. These

specialized entities do not have any peers inside of their subnetwork, they do not stream any resource, and external nodes are forbidden from joining their sub-groups.

- Host nodes, which initiate a new subnetwork, reserved for a specific resource, and feed the data into the subnet.
- Default nodes, which retrieve and propagate the data that is being streamed on a subnet.

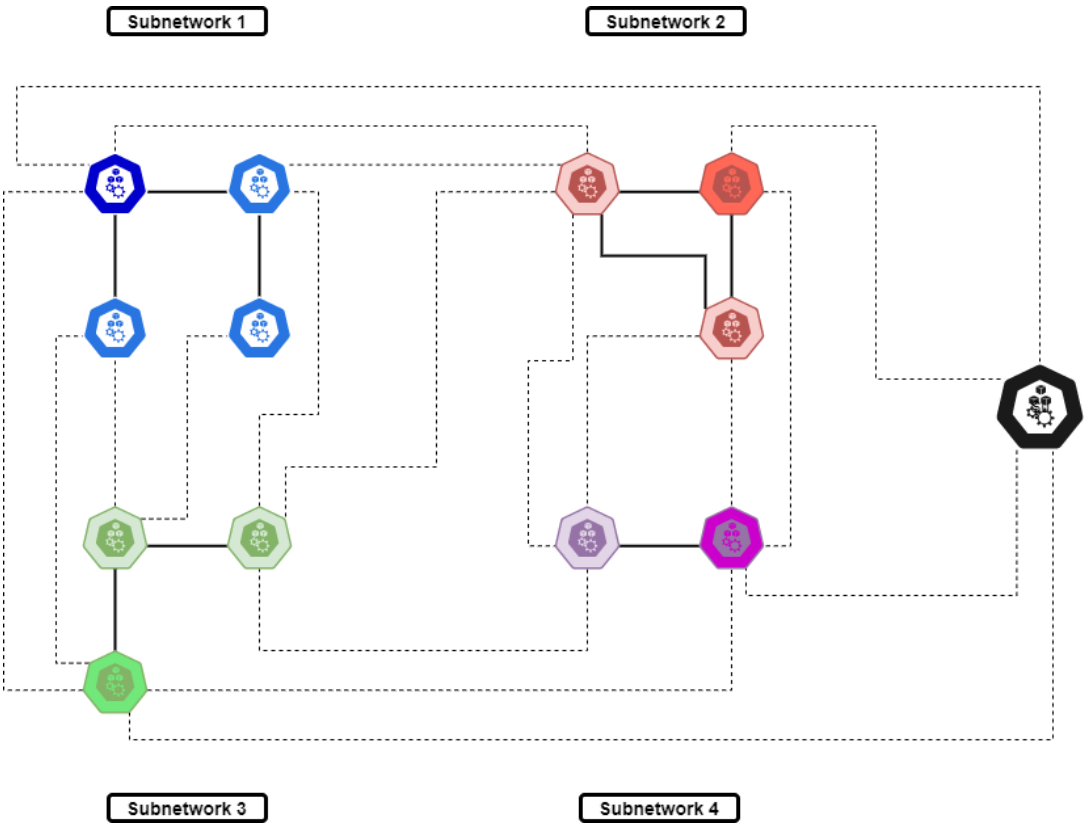

Figure 3.1: The architecture of the network, which is composed of multiple subnetworks (each represented with a distinct color). The black node is an infrastructure node.

Formally, the system can be represented as an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, in which the vertexes $(\mathcal{V})$ symbolize the nodes and the edges $(\mathcal{E})$ depict the connections between them. The structure is divided into multiple disjoint subgraphs $\mathcal{H}_i = (\mathcal{V}'_i, \mathcal{E}'_i)$, each corresponding to a subnetwork. Throughout the paper, the term "bridge" is used to denote an edge that connects nodes situated in distinct subgraphs. The mathematical

model for the network is the following:

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ network.

$\forall v_1, v_2 \in \mathcal{V}, v1 \neq v2, \exists$ path from v_1 to v_2

Let $\mathcal{C} = \{\mathcal{H} \subseteq \mathcal{G} \mid \forall \mathcal{H}_i, \mathcal{H}_j \in \mathcal{C}, \mathcal{H}_i \cap \mathcal{H}_j = \emptyset \text{ and } \cup_{\mathcal{H}_i \in \mathcal{C}} \mathcal{H}_i = \mathcal{G}\}$ network components set

Let $v_1 v_2 \in \mathcal{E}$ bridge $\iff v_1 \in \mathcal{H}_i, v_2 \in \mathcal{H}_j, \mathcal{H}_i, \mathcal{H}_j \in \mathcal{C}, \mathcal{H}_i \neq \mathcal{H}_j$

3.3 Framework Structure

The framework's structure was designed in a modular way, with the purpose of serving the maintenance process and further development. By separating the application logic into more components, errors can be isolated and identified with more ease. Fixes can be applied at module level, without affecting adjacent workflow sectors, as long as the updates do not affect the intermodular communication protocol. From a quality assurance perspective, this approach reduces unit test complexity implications, assuming that each component encloses just a part of the logic. The adoption of solid integration testing patterns is needed in order to ensure that the module synchronization flows are sound.

The functionality is split into 2 distinct modules:

- The Network State DHT Synchronization Module
- The Resource Streaming Module.

The Network State DHT Synchronization Module

The Network State DHT Synchronization Module (subsection 4.1.1) contains the logic for managing connections between peers, for redirecting new nodes based on their requested resource, and for assuring the fact that every participant holds a consistent, up-to-date network topology map, in the form of a distributed hash table (DHT). The DHT incorporates data related to network entities and its utility lies in its contribution

to decision-making processes. Whenever a new node attempts to acquire a certain resource, the tracker nodes have to decide upon two optimal subsets of nodes, based on heuristics. First, they determine the ideal peers from inside the subnetwork associated with a specific resource. Then, they choose the best candidates that are located outside of the subnetwork. The heuristics prioritize maintaining connectivity at graph-level and subgraph-level, while also considering the load that is being put on each node, therefore attempting to stabilize the network structure. The DHT's structure is comprised of information regarding the nodes, their connections, the way they are being clustered into subnetworks, and the bridges between the network components, as seen in Figure 3.2.

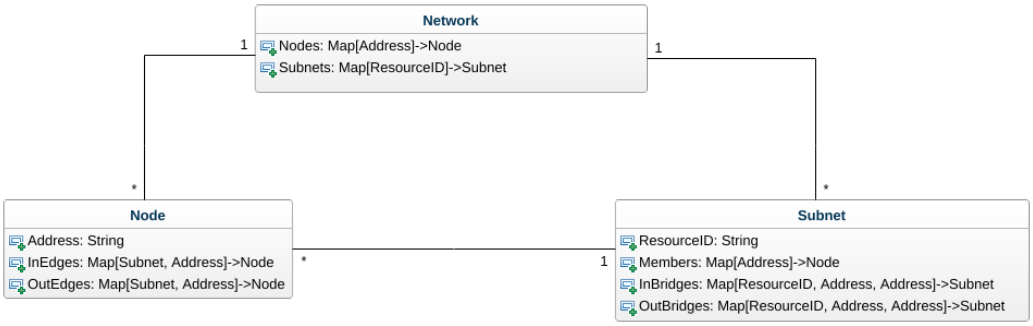

Figure 3.2: The custom network graph DHT data structure, which contains data about the network's nodes, the subcomponents and the relationships between them.

Whenever a node detects a change in the network, in the form of a new user connecting or leaving the establishment, it sends an update announcement to its immediate peers. Its neighbors further propagate the message, causing it to swarm through the entire system, in order to keep the DHT consistent for every participant. This behavior, from a single client's perspective, is depicted in Figure 3.3. Due to the fact that the network follows a mesh structure, there is a possibility that one node will receive the same DHT update message several times, from distinct peers, as each peer will detect that specific event at a different timestamp. So as to prevent information from endlessly swarming through the network, if a member receives an update request which cannot be reproduced on its copy of the net graph DHT, it will reject the request and will not further communicate it to its neighbors. In order to erase the possibility of valid events being dismissed due to the fact that they were announced in an erroneous order, the DHT alteration notices are sent several times, at set time intervals.

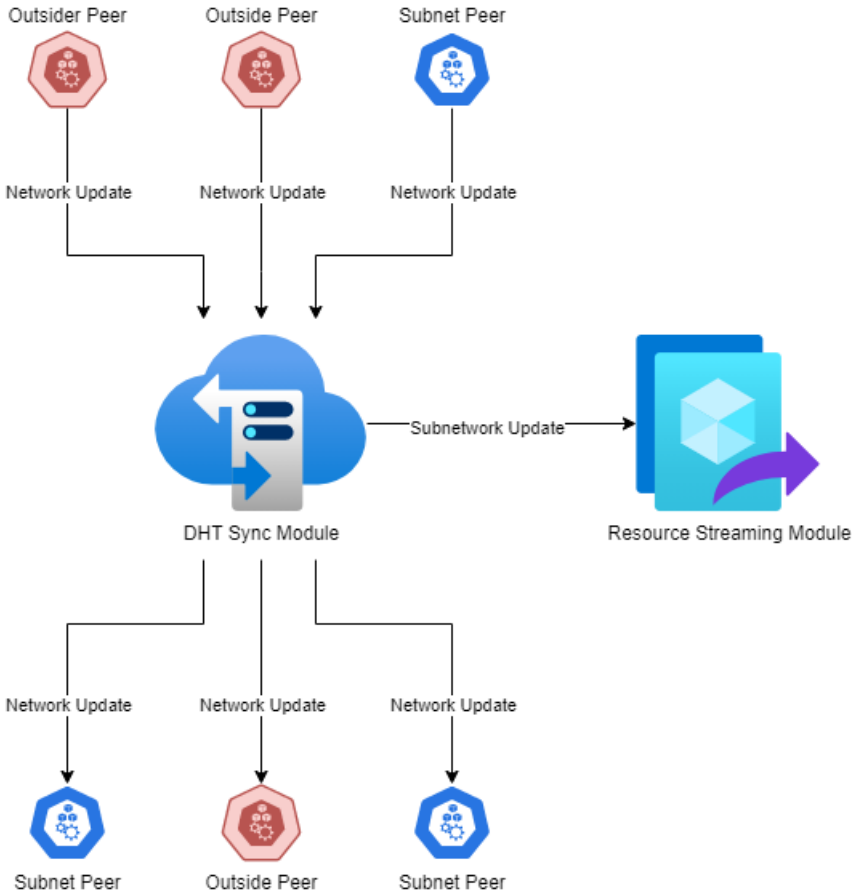

Figure 3.3: The process of propagating network updates, from the perspective of the client. Initially, the client receives a network update notification, which is processed inside the DHT Sync Module. Then, the Resource Streaming Module is notified. Finally, the update is propagated to the node's children.

The Resource Streaming Module

The Resource Streaming Module (subsection 3.5.2) handles the data transmission operations, based on the subnetwork structure provided by the Network state DHT synchronization module. In order to prevent corrupted or malicious data from being tunneled through the framework towards the user application, the module keeps track of the resource frames received over time. A frame is propagated towards the user application, via a local UDP server, after the current node becomes aware that the number of peers that voted for it surpassed a specific numeric threshold. The threshold is being computed as $P \cdot N$, where $P \in [0, 1]$ is a fixed percentage and N is the current number of subnetwork peers of the receiving node. The Resource Streaming Module does not communicate with nodes that do not belong to the same network component. When a

frame is received, the streaming controller distributes it to the node's peers, as shown in Figure 3.4.

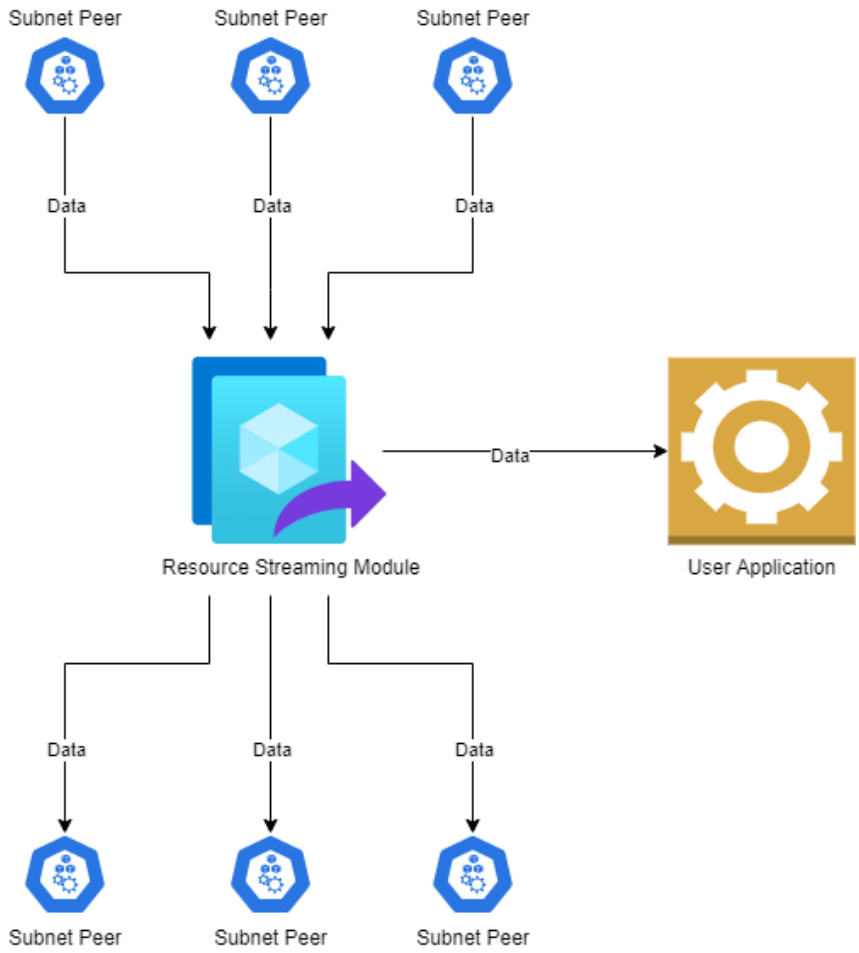

Figure 3.4: The resource data propagation process, on client level. The node receives a new frame from its subnetwork parents. If the data is valid and the vote counter for that specific frame surpassed the required threshold, the Resource Streaming Module instance sends the information towards the user application. Eventually, the frame is propagated to the client's subnetwork children.

In order to utilize the framework, one can either import the *Golang* package directly or use one of the following tools:

- The REST API (subsection 3.5.3), whose purpose is to facilitate the integration of the resource streaming functionality in applications written in different programming languages, without exposing any endpoints outside of the local network, to prevent security breaches
- The Python REST API wrapper (subsection 3.5.4), which represents a Python package that covers the HTTP calls towards the REST API and the communication with the Resource streaming module.

Additionally, the Network monitoring tool (subsection 3.5.5) serves as a dashboard meant to offer a visual overview of the network state and its evolution over time. Every super-node hosts an instance, which can be accessed via an HTTP call to a specific port.

3.4 Technologies

Several technologies were used during the implementation phase of the project, as it covers multiple software engineering topics, such as networking, system administration, or GUI.

Considering the fact that the widest part of the platform is centered around synchronization and data transmission, the communication aspects were the ones that contributed the most to deciding on a proper programming language. Even though most languages provide the required primitives for socket programming, there was a crucial need for a high-level language that is specifically designed for developing distributed systems, due to the fact that a lot of the logic is centered around the Peer-to-Peer network's structure. *Golang* [19] is an emerging, technology that meets these exact criteria. The TCP protocol was used for data transmission on the net graph DHT Synchronization Module, in order to ensure that the DHT updates arrive in chronological order, without any loss of information, as specified in subsection 4.1.1. UDP represents a quicker alternative, does not require any handshake methods or control flags, yet it is susceptible to error, due to the fact that it does not implement any mechanisms for preventing packet loss or duplicate packet delivery [20]. The UDP protocol is used for the sockets that operate in the Resource Streaming Module (described in subsection 3.5.2), considering the speed requirements implied by the high magnitude of data that the platform is supposed to handle.

The network monitor was implemented using *Python* 3 [21], due to the variety of 3rd-party libraries that it provides with the use of its package management system, *pip*. The GUI was realized using *Dash* [22], a framework that facilitates the creation of interactive plots and graphs, which was utilized for creating both the statistical data unit and the network graph unit. These components are described in subsection 3.5.5.

In order to test the system, an artificial ecosystem was created and populated with mock applications, which made use of the P2P resource streaming platform. The compute services provided by *Google Cloud* were used for this specific task, so as to simulate activity in the network, for benchmarking and verification purposes.

3.5 Implementation

This section will go over the implementation details for each module of the application, including the API's and the monitoring tool. The most emphasized aspects are those describing the Network State DHT Synchronization Module and the Resource Streaming Module, as those contain the actual logic for the system.

3.5.1 Network State DHT Synchronization Module

Due to the fact that every node in the network acts both as a client and as a server (the only exception being supernodes, which only act as servers), the logic for this module is split into two categories:

- The client logic, which is comprised of the actions taken in order to connect to the network and retrieve its state, from the other peers
- The server logic, which is constituted of the processes that take place when handling new connections, redirecting the new users and informing them of the network graph state

3.5.1.1 The Client Logic

In order for a new client to join the system, it has to go through several validation stages, after which he will be successfully connected to a subset of peers from the subnet associated with the requested resource and another subset of members that are situated outside of that subcomponent, so as to receive updates from outside sources.

The client stage differs based on the intent of the requester. A supernode will not go through this stage at all. A user that is willing to stream a resource will not connect to any subnetwork peers, considering that a new subnet will be created specifically for this data source. A client that requests access to an existing resource will have to go through the whole procedure, as it requires constant stream data updates from peers that already have access to the given resource.

There are three phases for the connection initialization stage:

- The redirection stage, during which the trackers are being solicited to provide the new user with an optimal set of peers addresses, depending on its request
- The subnetwork peers association stage, during which the node attempts to connect to peers that posses data frames from the requested resource
- The outside peers association stage, during which the client initiates the net graph synchronization flow with nodes that do not share the same resource.

Each of these will be described in the following paragraphs.

The Redirection Stage

This step is carried through by all client types, except for the infrastructure nodes. A visual representation for this stage is provided in Figure 3.5. Firstly, the requester launches a new streaming module instance, putting it in an inactive state. Then, a connection initialization message is sent to the certain tracker nodes, specified by the user. This message will contain the client's intent (it can either opt to join an existing subnet or host another) and an ID associated with that resource, if needed. After receiving responses from all the tracker peers, it will parse them. If the client's goal is to retrieve data from a stream, the responses will contain two sets of addresses, one set containing connection data for a subdivision of the subnetwork peers, and the other set, for peers that are located in other subcomponents, based on heuristics, as stated in section 3.3. Otherwise, if the new node requested privileges for hosting a new data source, each response will be comprised of a unique ID that represents a valid resource ID for the new stream, and a set of addresses for the outside peers. The client per-

forms an intersection on the lists of peer addresses received from the trackers. It will also pick a random resource ID from those previously received, if the objective is to create another subcomponent. Finally, the connections to the trackers are terminated and the subnetwork connection stage is set off.

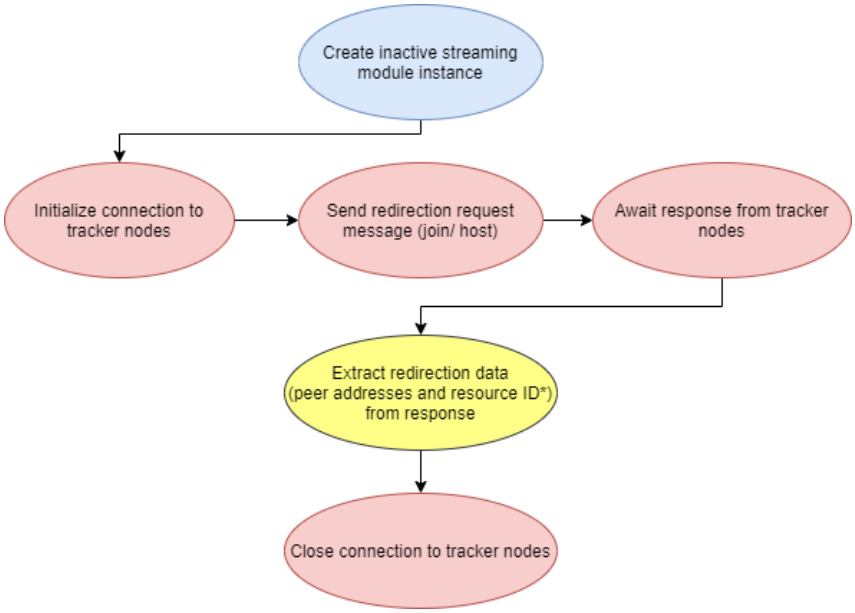

Figure 3.5: The redirection stage, meant to connect the new client to the optimal subnetwork and outside peers.

The Subnetwork Peers Association Stage

This stage is carried out only by the clients that solicit a specific data source. Figure 3.6 presents the steps that the client goes through in order to become a part of the requested subnetwork. The first action taken towards associating with the subnet peers implies initializing the connection with the previously described participants that share the required resource. Then, the user sends an initial message containing its intent to join the subnet, the data stream ID, and the port for the TCP server that it will use for accepting and exchanging data with nodes that will connect to the subnetwork in the future. Each peer will respond with a message containing the port for the UDP server that they're using for streaming data on the Resource Streaming Module, and the maximum resource size (in bytes), so that the client can further communicate these to his instance of the Streaming Module, which can now become active, receiving resource frames from its parent nodes of the subnet and propagating the information to its children. The new member will also receive net graph DHT update messages from its

immediate neighbors whenever they detect a change in the network state. The client's responsibility consists of updating its locally stored DHT instance and propagating the event further, to its other peers.

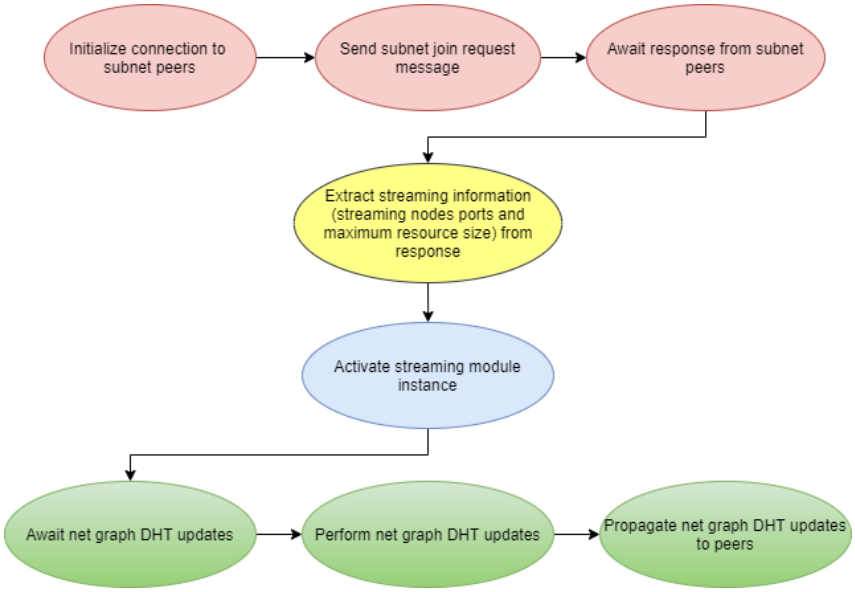

Figure 3.6: The subnetwork peers association stage, during which the new client attempts to join a given network subcomponent, based on the data that was previously received from the tracker nodes.

The Outside Peers Association Stage

This part of the logic is executed by all types of requesters, except for the supernodes. Figure 3.7 describes the actions executed throughout this stage. After connecting to the outside peers, the node sends them an affiliation solicitation message, containing the peering intent, the ID of the stream that the client has just joined, and the port for the TCP endpoint which will be used to serve incoming requests from new users. The outside peers' responses will contain the network state graph in a serialized, compressed form. After receiving all proposed net graphs, the client will choose the most voted one in order to eliminate the possibility of invalid net state data being retrieved from the peers. The reason why only the outside peers are being asked to provide this information lies in the fact that there is a possibility of all the nodes in a subnetwork being malicious, therefore providing the user with biased information. Similarly to the subnet peers, the outside neighbors will also announce the new node regarding network state changes, based on which the client will have to act as stated in paragraph "The

Subnetwork Peers Association Stage". After this step is executed, the server logic takes effect.

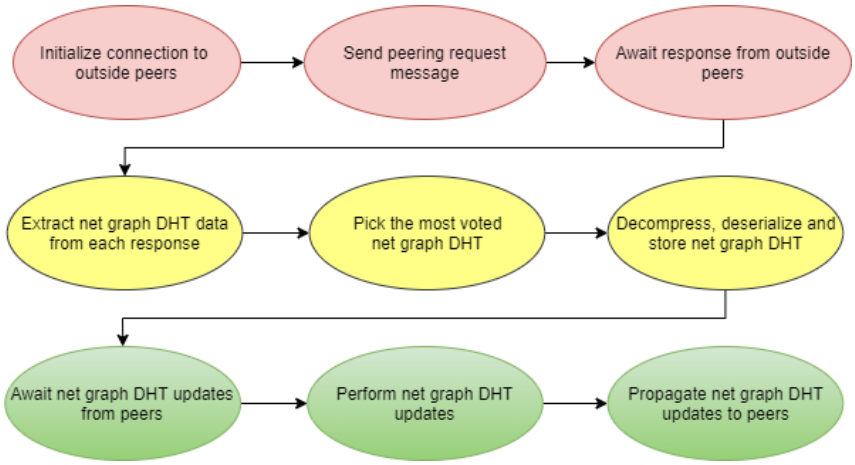

Figure 3.7: The outside peers association stage, meant to peer the new client to nodes situated outside of its subnetwork, to ensure that the net graph DHT instances stay consistent throughout all the system's subcomponents.

3.5.1.2 The Server Logic

After the client stage was successfully executed, the new node can officially be considered an active component of the network. Its responsibility consists of announcing the net graph DHT updates to its immediate neighbors, validating calls received from new clients, satisfying their requests, and redirecting them to the required location in the network, if needed.

Initially, the node initializes a TCP server, using the port that was reported to its peers during the connection phase, and begins waiting for new requests. After receiving an initial message, it extracts the needed data, based on the client's intent. There are three possible goals that the parent has to satisfy, each following a different flow:

- Redirection request, which implies locating the requested subnetwork (optionally, if the user aims to access an already existing resource) and providing the client with the needed peering information
- Subnetwork join request, which corresponds to the client's intent to access the data that is being streamed in the parent's subnet, requiring validation from the server node

- Outside peering request, which involves asking the parent for permission to form a direct connection, for the purpose of synchronizing net graph DHT instances.

Satisfying a Redirection Request

The redirection request message specifies whether the user desires to join or host a subnetwork. If its plans imply joining a subnet, the server will determine the given subunit's position in the network, using the state stored in the net graph DHT, and will respond with a set of addresses, belonging to members of that subnet. The members are selected based on a heuristic that attempts to reattach disconnected components and prevent the appearance of such structures. Otherwise, if the client's purpose is hosting a new resource, the parent generates a unique ID for the new stream. Afterwards, in both situations, the server will retrieve a batch of nodes that are situated outside of the requester's future subgroup, according to a set of rules, which prioritize the creation of relations between separated, unlinked subnetworks, in order to prevent the emergence of isolated subcomponents. After providing the client with the aforementioned information, the server stops the connection. This part of the logic is covered in Figure 3.8.

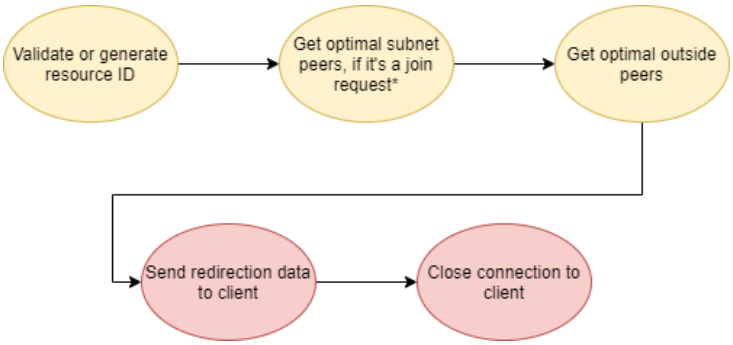

Figure 3.8: The process of satisfying a redirection request, from a tracker node's perspective. The node looks up the required subnetwork ID in the net graph DHT and retrieves optimal nodes from that subcomponent.

Satisfying a Subnetwork Join Request

The subnet join request message contains the ID that corresponds to the data source and the port used for the TCP endpoint that the new node will use in order to exchange

update messages with its peers. The parent will validate the resource ID. The port will later be used to store the node's entry in the net graph DHT. The response will contain the port that is being used by the server on its Resource Streaming Module instance and the maximum resource frame size (in bytes). Then, the parent will create a new instance in the net graph, associated with the child node. Finally, it will send a DHT update message to its peers, which will be propagated through the entire network. The logic for this stage is described in Figure 3.9.

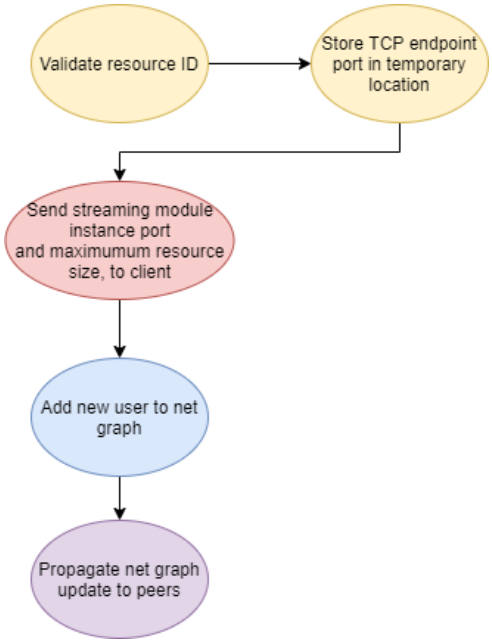

Figure 3.9: The process of satisfying a subnetwork join request. The receiver node validates the request, updates its own instance of the net graph DHT and notifies its neighbors of the respective update.

Satisfying an Outside Peering Request

The client's initial message incorporates the ID for the peer's subnet and its TCP endpoint's port. The server checks the existence of the resource ID and uses it in the process of associating the node with its corresponding subcomponent. Then, the parent uses the child's address and its TCP server port, to create a new net graph DHT entry. If required, it also creates a new subnetwork instance. Eventually, it announces the addition of these new DHT instances to its neighbors. The actions executed during this phase are represented in Figure 3.10.

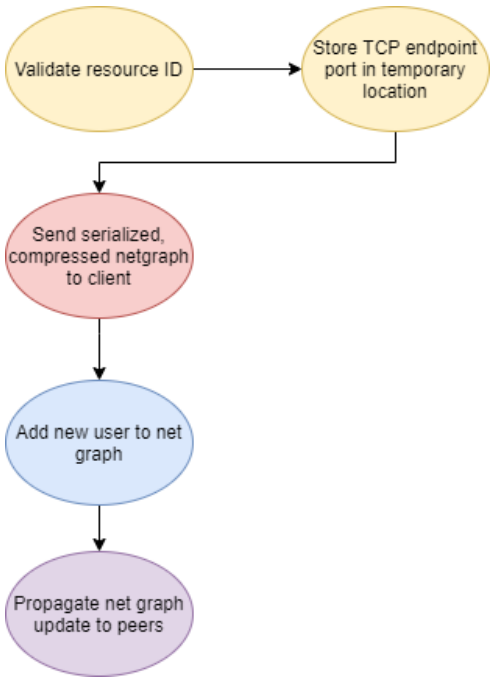

Figure 3.10: The process of satisfying an outside peering request. The receiver node validates the request and sends its copy of the net graph DHT to its new peer. Eventually, it updates its DHT instance and propagates the update to its neighbors.

3.5.2 Resource Streaming Module

The Resource Streaming Module provides a mechanism for continuously transmitting data frames. The data circulates based on the graph stored by the DHT Synchronization Module. The host feeds new frames into the subnetwork. Each member retrieves the information solely from its parent nodes. After receiving the same frame from a specific percentage of peers, it passes it to the application that consumes the services. Then, it propagates the data to its immediate neighbors.

A map is used to keep track of the received data, so that it can be shared with the node's children, or sent towards the user application. This structure holds the following fields:

- The frame's bytes
- The frame's hash, stored with the purpose of simplifying and optimizing the comparison between two frame instances
- An order number, which is used to order the entries in the map, based on the

time that they were received at, so that they are further propagated in chronological order

- The number of times it was received from a parent peer
- The last time it was received from a neighbor
- A list containing the parent nodes that issued the frame, to ensure that a single node cannot vote twice for the same entry
- A list of child nodes that have already received the frame, from the node in discussion.

There are 4 daemons that operate on this module, each one of them being discussed in the following paragraphs:

- The data retrieval daemon, which is active only on the non-root nodes, as the host is only supposed to feed the data to the other subnetwork members
- The frames propagation daemon
- The user application data transmission daemon
- The frames map cleanup daemon.

The Data Retrieval Daemon

The data retrieval thread encloses the logic for retrieving stream data from the parent nodes. The root node does not execute this process, as it represents the origin for the given resource, therefore being exclusively responsible for sharing frames with the other subnet constituents. Each parent is pinged periodically, as a form of requesting new data, if any is available. When a frame is received from a neighbor, the node performs a lookup in the frames map, based on its hash. If it was already added to the map, the node increments the vote counter for that specific entry and renews the list holding the parents that have sent it. Otherwise, a new instance is added to the frames map. Then, it updates the field holding information regarding the last time that the frame was received. If the number of votes exceeded a certain percentage of the total parent count, the entry goes into "accepted" state. An "accepted" instance can be transmitted

to the user application, upon a data retrieval request. The actions performed by this daemon are depicted in Figure 3.11.

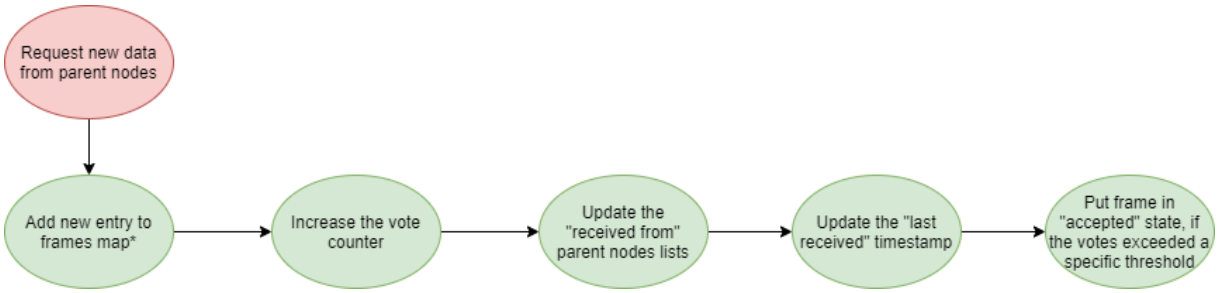

Figure 3.11: The data retrieval daemon, which continuously requests data from the client's parent nodes. Whenever new data is retrieved, it increases the vote counter for that specific resource frame. If the votes surpass a specific threshold, the frame is flagged as "accepted", so that it can be processed by the other Resource Streaming Module daemons.

The Frames Propagation Daemon

The frames propagation process satisfies data retrieval requests from child peers, in order to propagate the stream data through the subnetwork. Upon such request, the node creates a temporary list, based on the frames that were not sent to that specific child, prior to that timestamp. Then, it sorts the list, based on each instance's order number. Next, it answers the child's call by providing him the previously mentioned list of frames. Eventually, it updates the frames map instance, by appending the requester's identifier to the list of children that received each entry. The logic followed by this process is described in Figure 3.12.

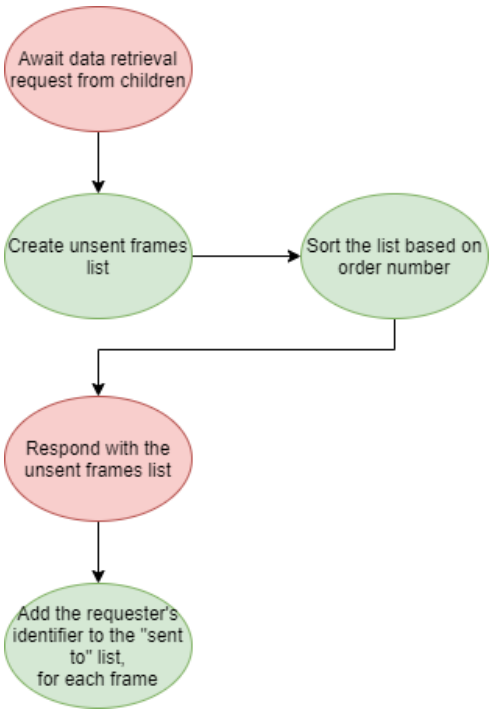

Figure 3.12: The frames propagation daemon, which continuously awaits for requests from the client's children, in order to propagate frames to the other subnetworks members.

The User Application Data Transmission Daemon

The user application data transmission daemon handles the exchange of data between the resource streaming module and the application that called the framework. Its scope depends on the type of node that was instantiated. If it's a host, then the daemon will be used to retrieve data from the application. Otherwise, its purpose will be to feed data towards the 3rd-party caller. If the node represents the root of the subnetwork, upon each call received on a specific UDP socket, it will create a new instance in the frames map, which will be streamed to the child peers, on future requests detected by the frames propagation daemon. If the member in discussion is a non-root constituent of the subnet, it will respond to the request with a list of frames that were not previously provided to the user application, similar to the one described in the "The Frames Propagation Daemon" paragraph. The actions performed by the user application data transmission thread are represented in Figure 3.13.

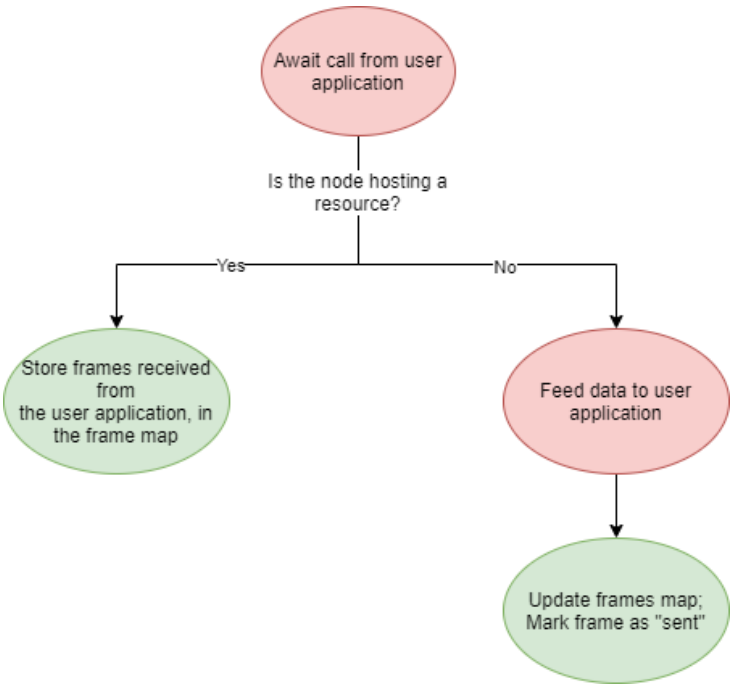

Figure 3.13: The user application data transmission daemon, which satisfies resource frame retrieval requests received from the user application.

The Frames Map Cleanup Daemon

In order to prevent the same frame from being propagated through the subnetwork multiple times, the module implements the frames map, which keeps track of the previously sent data. However, there are some cases when the host could intentionally feed the same information repeatedly. This would result in data loss, as the frames map ignores calls that supply duplicate instances. The frames map cleanup thread provides a mechanism for minimizing the probability of such erroneous information dropout occurring. The daemon performs periodical queries on the resource data map, searching for instances that have been active for longer than a set period of time, so as to remove them from the map. The steps executed by the frames map cleanup daemon are illustrated in Figure 3.14.

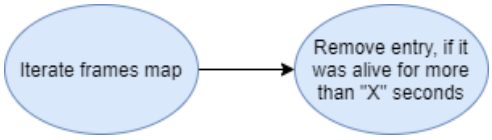

Figure 3.14: The frames map cleanup daemon, which removes entries from the resource frames map, that have been alive for more than a specific number of seconds.

3.5.3 REST API

The REST API associated with the resource streaming platform is a tool that facilitates creating and operating network nodes. While one could access the library directly from a *Golang* module, this tool enables interoperability with applications written in different programming languages, by simply performing HTTP requests. The server cannot be accessed from external sources and it only accepts calls from a local context, for security reasons. The resource streaming framework handler is an internal module, enclosed in this package, which performs the required actions on the resource streaming framework.

The REST API exposes 5 distinct paths:

- The "/ping" endpoint, used simply to confirm that the HTTP server is active, always returning an empty response body, with the 200 *OK* status code, regardless of the HTTP request method
- The "/host-super" endpoint (method: *POST*), which launches an infrastructure node
- The "/host" endpoint (method: *POST*), used to host a resource on the network; the caller needs to provide JSON-encoded dictionary, comprised of a list of strings, symbolizing IP:port pairs for the tracker nodes' synchronization servers used to join the network, and the maximum size (in bytes) of the resource that is going to be streamed
- The "/join" endpoint (method: *POST*), which instantiates a network member that will access a specific stream, based on a unique identifier, corresponding to an existing subnetwork; the request is expected to follow a certain format, consisting of a JSON-encoded map which contains a list of IP:port pairs, which act as connection information for each tracker node that is going to be contacted, and the unique identifier that belongs to the subnetwork that the caller intends to join
- The "/stop" endpoint (method: *POST*), which stops a node that was previously launched on that machine; the request body is supposed to contain a JSON-encoded string, which represents the local identifier chosen by the REST API for a specific entity that created using one of the "/host-super", "/host", or "/join" methods.

If the server receives an invalid call, due to an erroneous request body or the usage of a HTTP method that is not permitted for a specific endpoint, the response will consist of an empty body with a 400 *Bad Request* status code. If the call was valid, the API sets the status code value to 200 *OK* and populates the response body with a JSON string, which contains a series of fields, some of which are optional, depending on the initial request. The keys that are enclosed in the response dictionary are the following:

- *Success*, a *boolean* that indicates whether the action performed by the resource streaming framework completed accordingly
- *InstanceID* (default value: ""), a *string* which represents a locally unique identifier for the node that was launched, which is different from the ID stored in the network graph DHT, by the other members that are using the platform; this attribute is used in order to terminate an individual instance, at a later time; this field acts as a key in the instances map, which associates each identifier to its corresponding network node instance
- *TCPServerPort* (default value: 0), an *integer* that symbolizes the port of the TCP server launched by the DHT Synchronization Module
- *UDPServerPort* (default value: 0), an *integer* holding the port for the UDP server managed by the resource streaming module
- *ResourceID* (default value: ""), a *string* which stores the unique identifier for the resource that was created or accessed by the new node
- *MaxResourceSize* (default value: 0), an *integer* that represents the maximum size (in bytes) for the resource that was accessed by the new instance.

3.5.4 Python REST API Wrapper

The Python REST API wrapper is a library that acts as a client for the resource streaming REST API tool. Its purpose consists of handling the communication protocol between a Python application that imports this package, and the aforementioned HTTP server.

This module exposes 7 different methods:

- The *init* method, which launches a REST API executable, then waits for the service to become active, sending requests towards the *ping* endpoint, until it receives a response with the 200 *OK* status code; finally, it launches a non-blocking UDP socket, which will potentially be used for feeding data to the Resource Streaming Module, or, alternatively, for retrieving stream frames from it
- The *deinit* method, which closes the UDP socket that is communicating with the resource streaming module server and terminates the REST API instance
- The *host_super* function, which sends a *POST* request to the "/host-super" endpoint of the REST API, then parses the response body, return the success state of the operation, the instance ID generated by the HTTP server and the port for the TCP server used by the DHT synchronization module
- The *host* method, which sends a *POST* call to the "/host" endpoint on the HTTP server, then returns the success state for this action, the instance ID assigned by the REST API, the port for the server started by the net graph synchronization unit, the port for the endpoint that is running on the Resource Streaming Module, the unique identifier for the streamed resource and its the maximum stream frame size; this function requires a list of IP:port pairs, acting as connection information for the tracker nodes that are initially contacted when joining the network, and the maximum size (in bytes) for the resource that is going to be streamed in the subnetwork
- The *join* method, which sends a *POST* call to the "/join" endpoint on the REST API and returns the success state, the local instance identifier, the TCP server port for the DHT sync module, the UDP server port for the Resource Streaming Module, the resource ID and the maximum size of the resource; it requires the addresses and ports for the tracker nodes, the ID for the subnetwork, and a callback which gets triggered whenever new data is being received from the node's neighbors, in the streaming module
- The *stop* method, which sends a *POST* request to the "/stop" endpoint of the HTTP server, requiring a local instance ID; the function returns the success state resulted from attempting to terminate that specific node instance
- The *feed_data* method, which is used to feed data through the Resource Streaming Module, to the other subnetwork members, for a specific node instance; it

requires the local node instance ID, and the actual data, which it sends to the streaming unit, using the local UDP socket that was initialized during the call the *init* function.

3.5.5 Network Monitoring Tool

The Network monitoring application acts as a tool that provides a visual representation for the network state, by depicting the member nodes, considering the way that they're distributed in each subnetwork. The interface also provides statistical data related to the evolution of the net graph, for a set number of epochs.

The monitor process is managed by each infrastructure node in the mesh during the DHT synchronization module initialization phase, and it is accessible from outside the node's local network. In order to retrieve data related to the system's state, it maintains an active TCP connection with the DHT sync unit. The information transfer is triggered periodically and the update messages are comprised of data regarding the network nodes, more specifically, their addresses, the relations between each other, and their corresponding subnetwork. Based on these, the monitoring application extracts a series of stats. Also, the GUI is composed of two components: the statistical data unit and the net state graph unit.

As seen in Figure 3.15, the analytical part of this module encloses 8 plots, each of them providing information regarding the evolution of a specific aspect, throughout the network's lifetime:

- The number of nodes that are active in the network
- The number of connections between the network members
- The number of operating subnetworks
- The number of internal relations formed between members, for each network subdivision
- The number of external connections, for each network subcomponent
- The number of inner connections, per member, for each subnetwork

- The number of outer links, per member, for each subcomponent
- The number of outside connections, per each neighboring subnetwork, for each subdivision.

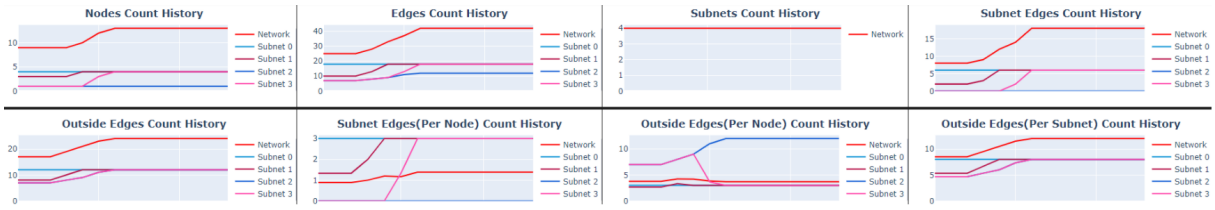

Figure 3.15: The statistical unit of the network monitoring tool, which provides historical stats regarding the network's state.

The second component of the monitoring tool is the net state graph unit, which mirrors the graph provided by the DHT Synchronization Module. This component plots the network mesh, as shown in Figure 3.16. It serves as a way of associating the data shown in the statistical unit with actual platform members, so that the system can potentially act in that direction.

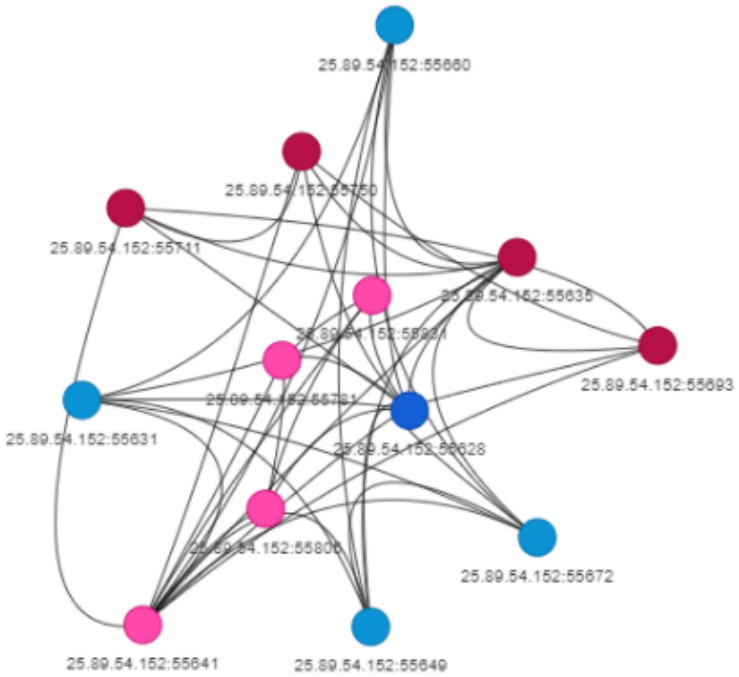

Figure 3.16: The net graph unit of the network monitoring tool, which provides a visual representation of the network. Each subnetwork is represented using a distinct color.

Critical Evaluation and Benchmarking

4.1 Formal Proofs using Petri Nets

This section focuses on providing formal arguments, in order to prove that each module of the proposed system behaves in a sound manner. The actions in the network were modeled using marked Petri Nets and the properties are demonstrated using reachability graphs associated with the respective nets.

A Petri Net is defined as a tuple $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{W})$, where $\mathcal{P}$ represents the set of places, or states, for the given process, $\mathcal{T}$ symbolizes the set of transitions between places, $\mathcal{F}$ is the set of arcs, each connecting a place with a transition ($\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$), and $\mathcal{W}$ is the set describing the weight of each arc ($\mathcal{W}\colon \mathcal{F} \to \mathbb{Z}$).

A marked Petri Net is a tuple $\gamma = (\mathcal{N}, \mathcal{M}_0)$, where $\mathcal{N}$ is a Petri Net and $\mathcal{M}_0$ ($\mathcal{M}_0\colon \mathcal{P} \to \mathbb{N}$) symbolizes the initial marking of the net. The execution of a transition (t) from a generic marking ($\mathcal{M}'$) results in the occurrence of another marking ($\mathcal{M}''$). The existence of such transition (t) implies the fact that $\mathcal{M}''$ is directly reachable from $\mathcal{M}'$ ($\mathcal{M}'[t > \mathcal{M}''$). This idea can be formalized as such:

$$\mathcal{M}'[t > \mathcal{M}'' \iff (\exists t.\mathcal{M}''(p) = \mathcal{M}'(p) - \mathcal{W}(p,t) + \mathcal{W}(t,p), \forall p \in \mathcal{P}))$$

The $\mathcal{M}'[* > \mathcal{M}''$ notation indicates that marking $\mathcal{M}''$ is reachable from marking $\mathcal{M}'$, occurring as a result of the execution of a transition sequence ($u = t_1 t_2 t_3 ...$), starting from $\mathcal{M}'$. $[\mathcal{M} >_\gamma$ represents the set of markings that are reachable from $\mathcal{M}$, in a marked Petri Net (γ).

The reachability graph of a marked Petri Net $\gamma = (\mathcal{N}, \mathcal{M}_0)$ represents a directed graph which contains all the markings reachable from the initial marking, including the initial one, and is defined as described in the following block:

$$\mathcal{G}_{\mathcal{R}} = (\mathcal{V}, \mathcal{E}), \mathcal{V} = \{\mathcal{M} | \mathcal{M} \in \{\mathcal{M}_0\} \cup [\mathcal{M}_0 >_{\gamma}\}, \mathcal{E} = \{t \in \mathcal{T} | \exists \mathcal{M}', \mathcal{M}'' \in \mathcal{V}.\mathcal{M}'[t > \mathcal{M}''\}$$

The aspects that the following subsections are demonstrating are the reachability and the boundedness properties [23], which prove that the required actions are being executed in a finite number of steps, according to the intended system logic, as long as every marking that occurs during an arbitrary execution sequence only appears once.

4.1.1 Network Graph DHT Synchronization Module

This subsection will focus on proving that the synchronization process takes place in a sound manner, completing its execution in a finite number of steps. Figure 4.1 depicts the network that was considered in order to demonstrate the stated properties. The network consists of three nodes, each one of them being the sole members in their corresponding subnetwork. The goal for the proposed architecture is to be able to announce every participant of a specific change in the net graph, by swarming the network with the required information. Each member has to perform the necessary operations on its distributed hash table instance exactly once.

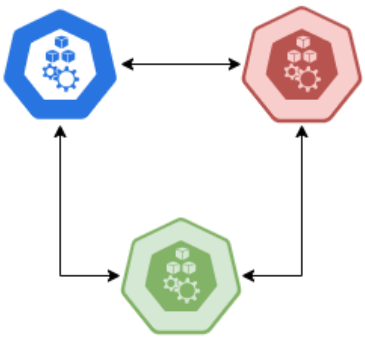

Figure 4.1: Network example, considering three isolated nodes, each situated in a distinct subnetwork.

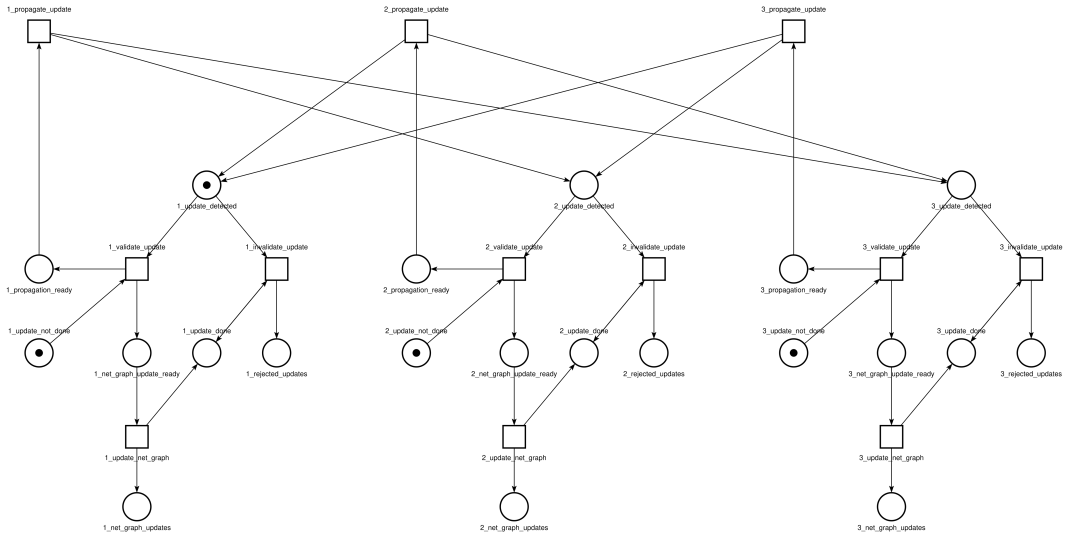

Figure 4.2: Petri Net describing the DHT synchronization process, in a network comprised of three nodes (initial marking).

Figure 4.2 describes the Petri Net used in order to model the previously described behavior. The initial marking indicates the fact that the first node detected a change in the network state, therefore having the responsibility of updating its net graph DHT instance and informing its direct neighbors (the second and the third node) of the recently occurred changes.

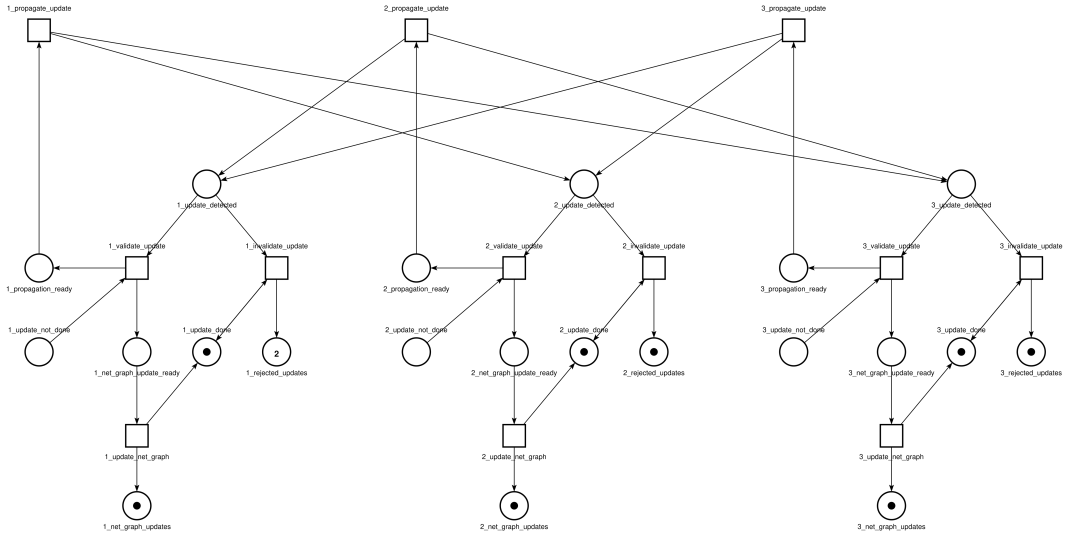

Figure 4.3: Petri Net describing the DHT synchronization process, in a network comprised of three nodes (final marking).

Figure 4.3 shows the final marking for the considered Petri Net. After the synchronization process is completed, every participant is supposed to have the updated version

of the net graph DHT, based on the previously detected network changes. As the system implements a version of the epidemic protocol, each member receives the same DHT update message from multiple sources. It's the client's responsibility to discard the duplicate messages and avoid transmitting them further, as a method of avoiding the infinite propagation of such notices. In the example network, each member will be notified twice (once per neighbor node). Therefore, each one of them will detect one redundant update message. The only exception is the node that first detected the network change (the first node), which will discard all the update messages regarding the event in discussion.

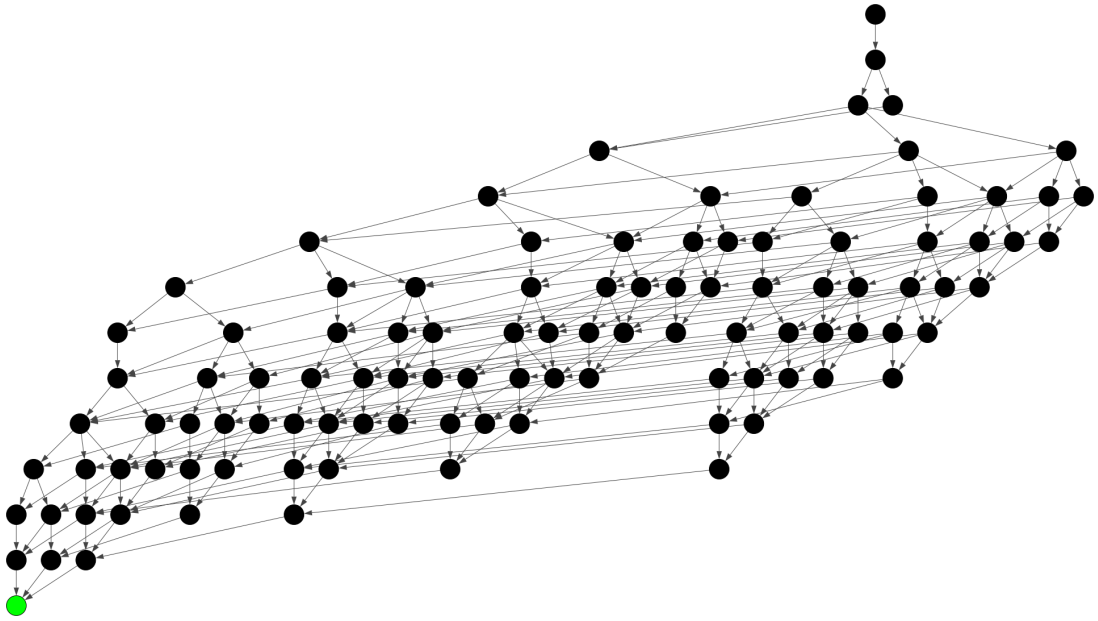

Figure 4.4: Reachability graph of the Petri Net that describes the DHT synchronization process.

The reachability graph for the Petri Net is depicted in Figure 4.4. A specific color code was used in order to differentiate between markings. The default color for the nodes is black. Green vertices represent sink nodes which indicate the presence of a marking in which all the network members performed the required update on their net graph DHT copy.

Based on the reachability graph, the final marking shown in Figure 4.3, corresponding to the only sink node in the structure, is **reachable**. Also, it is a **dead marking**, since the node's outdegree is 0. These arguments demonstrate the soundness property of the net graph DHT synchronization process. Furthermore, the Petri Net is **bounded**, due

to the fact that the reachability graph has a finite number of nodes [23]. Also, every marking resulted during an execution sequence only appears once. The previously mentioned ideas indicate the fact that the sync action completes in a finite number of steps, proving that the duplicate update avoidance mechanism also functions correctly.

4.1.2 Resource Streaming Module

The network example considered for the resource frame streaming process is similar to the one described in subsection 4.1.1. Due to the fact that the frames are only propagated internally, inside of each subnetwork, all of the members belong to the same subcomponent. As shown in Figure 4.5, the chosen network hierarchy implies an orphan node that has two children, acting as the host for the given resource, a node that has one parent and one child, and, respectively, a node that has two parents and no children. The objective for the streaming process implies the successful frame distribution to all the subnetwork constituents, in a finite number of steps. Each frame is supposed to be processed by a client exactly once.

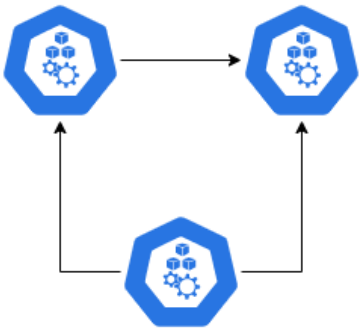

Figure 4.5: Network example, considering three isolated nodes, belonging to the same subnetwork.

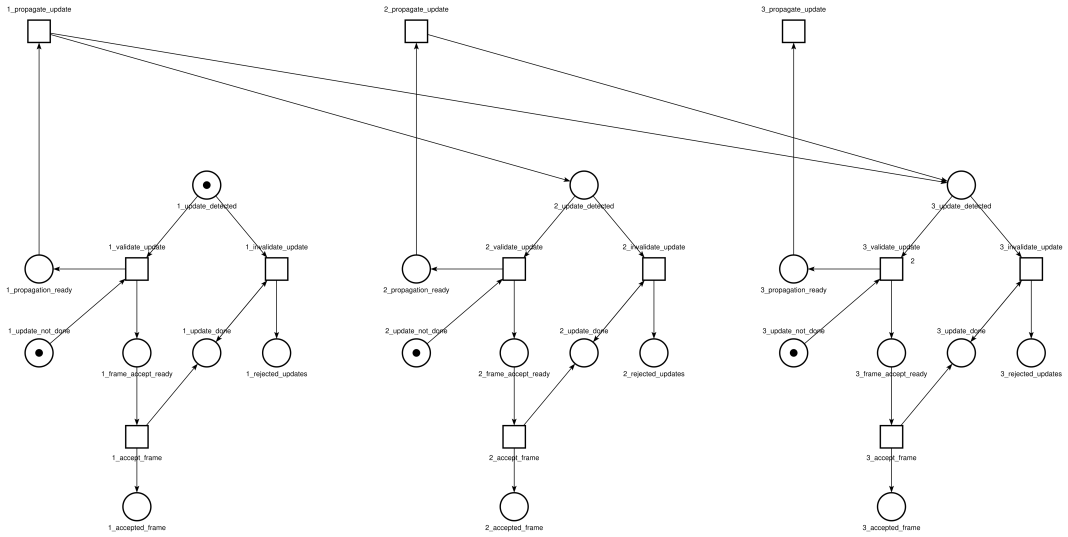

Figure 4.6: Petri Net describing the resource frame streaming process, in a subnetwork comprised of three nodes (initial marking).

The Petri Net that models the desired streaming process behavior for the aforementioned network example is depicted in Figure 4.6. Due to the constraints implied by the streaming module, no subnetwork member is able to transmit data to a parent node. Therefore, the second node cannot propagate frames to the third node, while the third one can only receive data. In the considered example, the last node must receive two update notifications for that specific resource frame, before processing it. The reason for that matter is that, according to the description provided in Figure 3.3, a frame is handled by the client only after its vote count surpassed a specific numeric threshold. For this particular model, all of the nodes' parents must confirm the validity of the frame. In the initial state, the first node, which represents the data stream host, receives a new frame on the user application data transmission bridge, which is supposed to be transmitted to all other subnetwork members.

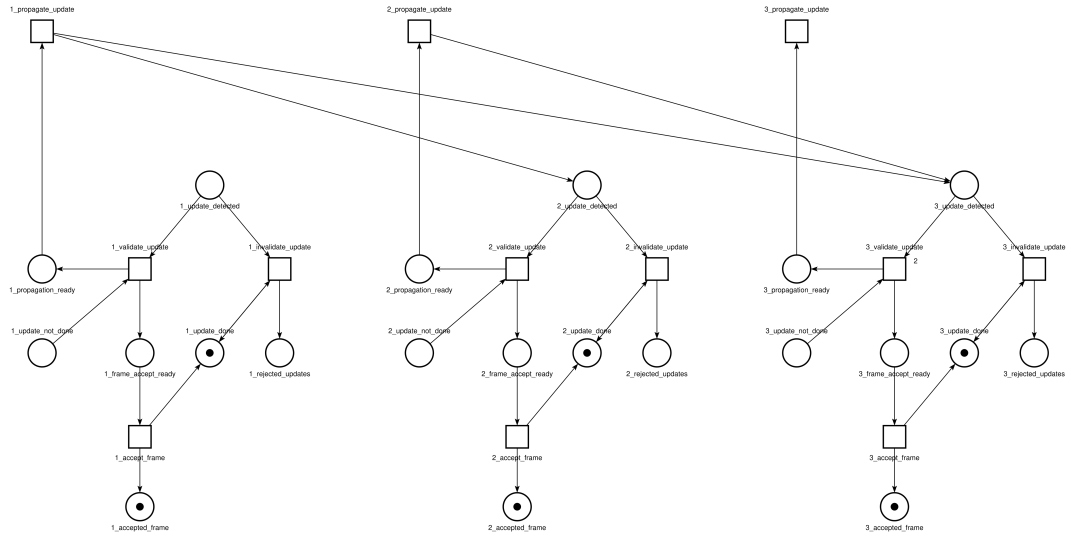

Figure 4.7: Petri Net describing the resource frame streaming process, in a subnetwork comprised of three nodes (final marking).

Figure 4.7 depicts the final marking for the Petri Net that models the resource streaming process. Upon the successful completion of the frame propagation throughout the subnetwork, all nodes are expected to have processed the received data. The process should not result in any rejected data due to the subnetwork hierarchy and the logic of this particular module.

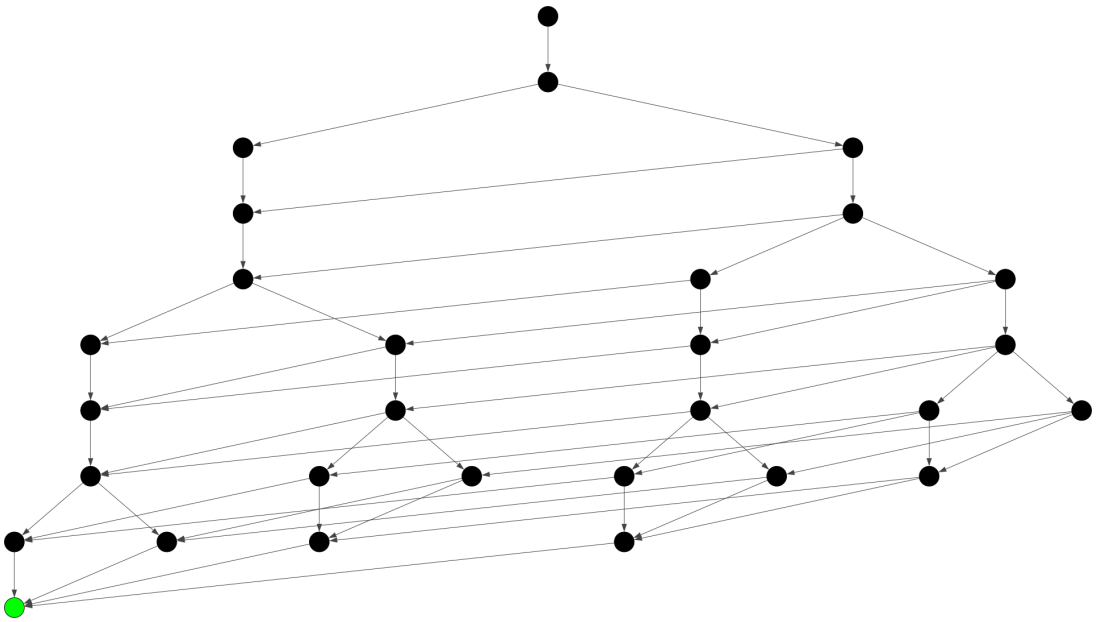

Figure 4.8: Reachability graph of the Petri Net that describes the frame streaming process.

Figure 4.8 represents the reachability graph for the aforementioned Petri Net. The coloring convention used in the graph is the same as the one described in subsection 4.1.1.

The only sink node present in the reachability graph corresponds to the final marking described in Figure 4.7. The marking's presence in the structure proves its **reachability** property. Additionally, it represents a **dead marking**, due to the fact that its outdegree is 0. Therefore, the behavior for the streaming process is sound. Furthermore, the fact that the reachability graph contains a finite number of vertices implies the idea that the considered Petri Net is **bounded** [23]. Considering all the paths from the source node to the singular sink vertex, each node that's part of the given path only appears once. This indicates that during an execution sequence no marking can occur multiple times. Therefore, the frame propagation process is completed successfully in a finite number of steps.

4.2 Benchmarks

A benchmarking plan was established in order to analyze the framework's performance on genuine tasks, focusing mainly on speed measurements. The plan is comprised of the following metrics: transfer speed (UDP packets per second, bytes per second), data loss percentage, and latency.

The benchmarks were realized using a minimal network example, on a custom video streaming application that implements the resource streaming framework. Each virtual node was running on a separate machine, to ensure that the data goes through each TCP/UDP layer, in order to simulate a proper usage example. Each machine had different hardware configurations. The average number of virtual processor cores was 4 and each computer had an average of 8 GB RAM. The upload speed was approximately 30 MBps and the download speed was about 115 MBps.

Under these circumstances, the results indicated that each receiver node processed an average of 14.68 UDP packets per second, equivalent to a transfer speed of 0.917 MBps. The recorded data loss percentage was 0%, meaning that no packets were lost during the execution of the resource frame propagation mechanism. Finally, the measured average latency was 3.6 milliseconds.

Conclusions

5.1 Summary

To conclude, the proposed platform meets the project's main objective of developing a decentralized system designed specifically for resource streaming. The framework's idea of encouraging clients to participate in the decision-making processes, instead of implementing a hierarchy-based solution that relies on a central authority, ensures that the mechanisms operate in a fair and secure manner, without favoring any particular members or groups, or endangering users' confidentiality. While the possibility of a partial outage exists, due to the fact that the subnetworks depend upon the host's availability, a total outage is unlikely, considering that the network stays active as long as there's at least one active entity.

Efficiency-wise, although the streaming tool requires a lot of memory resources, machines with more virtual processor cores generally obtain better performance when using the platform, as the bottleneck is represented by the fact that the framework makes use of several threads which run in parallel, in order to detect and propagate network changes, or process and transmit data.

5.2 Applications

The resource streaming framework can benefit several types of applications that require a method for data transfer. The main domains considered during the system design process were business, education, gaming, and entertainment.

When considering the field of business, the framework can be used to implement database replication and synchronization tools. The setup phase would consist of initializing a new subnetwork for each machine holding a database instance. Then, each member of the cluster is supposed to connect to each stream created in the previous phase. In order to keep the instance states consistent, the user application must place a trigger on each table that is meant to be synchronized across all servers. Whenever a query is executed on a machine, the trigger would fire, sending the current query string towards the framework's local REST API. Eventually, all the other members would receive the request, being able to replicate the query on each database instance.

The framework's safer and lighter approach to data distribution could also benefit media streaming services due to its fully decentralized nature, which eliminates any Points of Failure (PoF's) and disperses the higher payload that a central authority would have to endure, distributing it to all of the members of a specific subnetwork. Media streaming tools are heavily used in fields such as education, gaming, and entertainment. Specific implementations include virtual classroom applications and general use streaming platforms.

5.3 Future Work

Further releases of the proposed resource streaming platform should focus on providing additional security measures, in order to ensure user data confidentiality, and a proper mechanism for restricting client access to specific streams.

The most crucial feature that needs to be implemented in future framework versions is the security layer. Currently, the data is being sent in raw form, making the message propagation apparatus vulnerable to packet sniffing attacks. The suggested approach for improving safety during the information transfer act is the usage of asymmetric key encryption. This feature would also benefit the addition of an access restriction system. In addition to the encryption mechanism, the framework could make use of digital signatures, in order to authenticate a new client to a given subnetwork, therefore introducing the concept of a private subnetwork. There are several protocols that specialize

on routing messages signing (e.g. Chained One-Time Signature Protocol [24], Independent One-Time Signature Protocol [24]), therefore serving the exact purpose of the DHT Synchronization Module for the resource streaming platform.

Lastly, further releases could implement a method of streaming data from multiple sources, in the same subnetwork. There are certain cases in which the user could need to propagate information coming from multiple sources. For example, given the current resource streaming platform version, the database replication and synchronization tool mentioned in section 5.2 would imply the creation of multiple subnetworks, each server acting as a host for a different resource. Then, each machine is supposed to join all the other newly created network components. This would result in an increased payload on each member of the decentralized database cluster. Therefore, such applications would benefit substantially from the addition of multi-source streaming features.

References

[1] Cisco Annual Internet Report (2018–2023). 2018.

[2] Kimberly Houser and W. Voss. GDPR: The End of Google and Facebook or a New Paradigm in Data Privacy? *SSRN Electronic Journal*, 07 2018.

[3] Michelle Goddard. The EU General Data Protection Regulation (GDPR): European Regulation that has a Global Impact. *International Journal of Market Research*, 59(6):703–705, 2017.

[4] Antonio Buttà. Google Search (Shopping): an Overview of the European Commission's Antitrust Case. *Antitrust & Public Policies*, 5(1), 2018.

[5] Nathan Newman. Search, antitrust, and the economics of the control of user data. *Yale J. on Reg.*, 31:401, 2014.

[6] Hamed Rahimi, Ali Zibaeenejad, and Ali Akbar Safavi. A novel IoT architecture based on 5G-IoT and next generation technologies. In *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 81–88. IEEE, 2018.

[7] Dongyan Xu, Mohamed Hefeeda, Susanne Hambrusch, and Bharat Bhargava. On peer-to-peer media streaming. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 363–371. IEEE, 2002.

[8] K Hareesh and DH Manjaiah. Peer-to-peer live streaming and video on demand design issues and its challenges. *International Journal of Peer to Peer Networks*, 2(4):1, 2011.

[9] Nazanin Magharei and Reza Rejaie. PRIME: P2P Receiver-drIven MEsh-based Streaming. NOSSDAV, 2006.

[10] Halikul Lenando and Mohamad Alrfaay. Epsoc: Social-based epidemic-based routing protocol in opportunistic mobile social network. *Mobile Information Systems*, 2018, 2018.

[11] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System. *arXiv preprint arXiv:1407.3561*, 2014.

[12] Spyros Voulgaris and Maarten Van Steen. An epidemic protocol for managing routing tables in very large peer-to-peer networks. In *International Workshop on Distributed Systems: Operations and Management*, pages 41–54. Springer, 2003.

[13] Weiyu Wu, Yang Chen, Xinyi Zhang, Xiaohui Shi, Lin Cong, Beixing Deng, and Xing Li. LDHT: Locality-aware distributed hash tables. In *2008 International Conference on Information Networking*, pages 1–5. IEEE, 2008.

[14] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[15] Ingmar Baumgart and Sebastian Mies. S/Kademlia: A practicable approach towards secure key-based routing. In *2007 International Conference on Parallel and Distributed Systems*, pages 1–8. IEEE, 2007.

[16] Raul Jimenez, Flutra Osmani, and Bjorn Knutsson. Connectivity Properties of Mainline BitTorrent DHT Nodes. In *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, pages 262–270. IEEE, 2009.

[17] Vivek Rai, Swaminathan Sivasubramanian, Sandjai Bhulai, Pawel Garbacki, and Maarten Van Steen. A multiphased approach for modeling and analysis of the BitTorrent protocol. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 10–10. IEEE, 2007.

[18] Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. LAND: Locality aware networks for distributed hash tables. Technical report, Tech. Rep. TR 2003-75, Leibnitz Center, The Hebrew University, 2003.

[19] Jeff Meyerson. The Go programming language. *IEEE software*, 31(5):104–104, 2014.

[20] Santosh Kumar and Sonam Rai. Survey on transport layer protocols: TCP & UDP. *International Journal of Computer Applications*, 46(7):20–25, 2012.

[21] Guido VanRossum and Fred L Drake. The Python language reference. 2010.

[22] Shammamah Hossain, C Calloway, D Lippa, D Niederhut, and D Shupe. Visualization of Bioinformatics Data with Dash Bio. In *Proceedings of the 18th Python in Science Conference*, pages 126–133, 2019.

[23] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[24] Kan Zhang. Efficient Protocols for Signing Routing Messages. In *NDSS*. Citeseer, 1998.

[25] Alfonso De la Rocha, David Dias, and Yiannis Psaras. Accelerating Content Routing with Bitswap: A multi-path file transfer protocol in IPFS and Filecoin. 2021.

[26] Enrique Costa-Montenegro, Juan C Burguillo-Rial, F Gil-Castiñeira, and Francisco J González-Castaño. Implementation and analysis of the BitTorrent protocol with a multi-agent model. *Journal of network and computer applications*, 34(1):368–383, 2011.

Similar Systems

This chapter addresses the technical aspects of the previously implemented systems which present similar characteristics as the proposed resource streaming system. The considered applications are the InterPlanetary File System (IPFS) and BitTorrent.

A.1 InterPlanetary File System

The InterPlanetary File System (IPFS) is a Peer-to-Peer distributed file system that provides a content-addressed file storage model [11]. IPFS inherits the main idea behind Torrent, while also providing means of development on top of the main framework [11]. IPFS provides a mechanism for distributing data across several clients. While it also follows a decentralized network model, its scope differs from the proposed system's goal, which, unlike IPFS, serves as a data streaming framework, not as a persistent storage system.

IPFS makes use of the S/Kademlia DHT [15], a secure extension of the Kademlia DHT [14], in order to coordinate and maintain system metadata [11]. S/Kademlia uses the XOR metric to provide a topological ordering for the routing table entries, knowing their keys. Based on the same concept, it can identify the closest nodes to a specific entry [15] [14] [11]. The proposed system takes a different approach to this matter, the distance metric being computed based on a member's position in the network graph DHT. The IPFS keeps the routing tables consistent between each node's copy with the use of a custom epidemic protocol implementation [10] [12].

In order to handle the exchange of blocks between peers, IPFS utilizes the BitSwap

protocol [11]. BitSwap operates on explicit content naming, being designed especially for Peer-to-Peer content-addressable networks [25]. While the proposed data streaming framework also represents a content-addressable network, the information transmission method differs, considering its use of frames, instead of blocks, due to the continuous data flow required for properly maintaining the streaming process.

A.2 BitTorrent

BitTorrent represents a self-scalable decentralized protocol for content distribution [17]. The transfer method is based on fixed-size blocks, so as to split the seeding responsibility among multiple nodes, in order not to overload a specific seeder.

When a new node requests a specific file, the peers that will provide blocks from that specific resource will be retrieved from a tracker [17]. The original BitTorrent implementation made use of centralized trackers, which were later replaced by DHT's [16]. A similar mechanism is implemented in the proposed system. The tracker is replaced by multiple nodes that retrieve an optimal set of peers that hold the required data, based on heuristics meant to balance the load in the network. The proposed network does not make use of any trackers in order to eliminate any type of hierarchy, when it comes to making decisions in the ecosystem.

BitTorrent implements two custom variations of the Kademlia DHT [14]: Mainline DHT [16] and Azureus DHT [16]. These distributed hash tables are being used in the process of peer discovery, based on the needed resource, therefore acting as routing tables for the ecosystem.

The BitTorrent protocol uses the choking algorithm in order to normalize the load distribution across a specific resource's seeders [17] [26]. This method prefers peers with higher upload capabilities, to avoid stressing nodes that have lower upload potential [17] [26]. In contrast to this approach, the proposed framework assigns similar loads to all network constituents, balancing the number of neighbors for all members.