# SWAzam

*An architectural overview*

**TU Wien**
**184.159 Software Architecture, 2014W**

**Group 19**
Cismasiu Anca (0727280)
Gamerith Stefan (0925081)
Krapfenbauer Klaus (0926457)
Reithuber Manuel (0725031)
Schreiber Thomas (1054647)

# Table of contents

# Architecture

## Overview

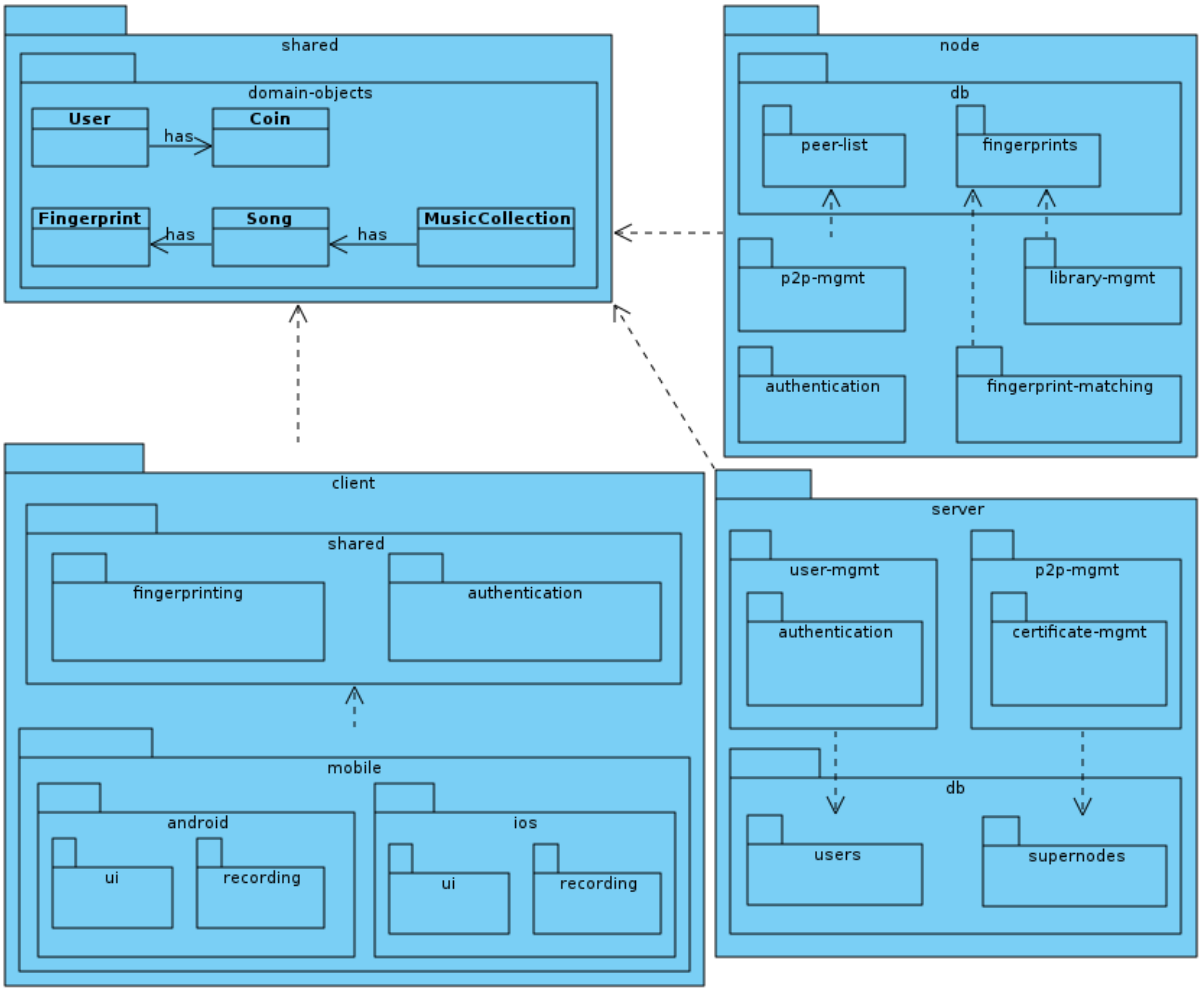
For the Swazam architecture, we chose a hybrid P2P/Client-Server architecture.

The clients, iOS and Android smartphones, must first register with the server to be able to use the service. After that, they use the microphone to record a music sample or choose a track from their local music collection. The app then runs a fingerprinting algorithm on this sample and sends it to one of the servers in the cluster through a load balancer.
The servers are only responsible for user registration, coin and certificate management.

After receiving the music recognition request, the server signs it with it’s own certificate, adds an ID, a callback IP and port and sends it to two supernodes with a TTL. These are nodes that we operate, as the Swazam creators. They run identical software as the peer nodes, but do not have a local music database. This means that the fingerprint matching part is inactive and they do not earn any coins. They are solely there to facilitate routing and try to balance the network.

The request is further routed into the actual P2P network, with each node running the fingerprint matching algorithm against their local music fingerprint database. When a match is found, the response is sent to the callback id and port of the request. The server handles the coin management and returns the result to the client.

## Module view




The module view is an important view for showing how the software source artifacts are structured and how they depend on each other. We structured our compilation units into four main modules which are the shared, node, client and server modules.

### shared.domain-objects

The shared module is used for sharing commonly needed classes among the other modules. In our case these are the domain objects of the application consisting of the User which has an amount of Coins and the MusicCollection which has a list of Songs which consists of slices stored as Fingerprints.

### client

The client module houses the functionality of the client devices which start a request for a song recognition. This module contains a submodule for every client platform type, which can be mobile (like in our case) or desktop or even web. For our module mobile we have two submodules for each specific platform namely Android and iOS. Each of these platform modules has two submodules for the implementation of the UI and for the code which is necessary for recording the audio sample (recording submodule).

#### client.shared

This module, like the global shared module, contains shared implementation units. These implementations contain common code to all platform types and platforms. On the one hand these are the algorithms for calculating the fingerprint of an audio sample and the logic for authentifying the user on the server.

#### node

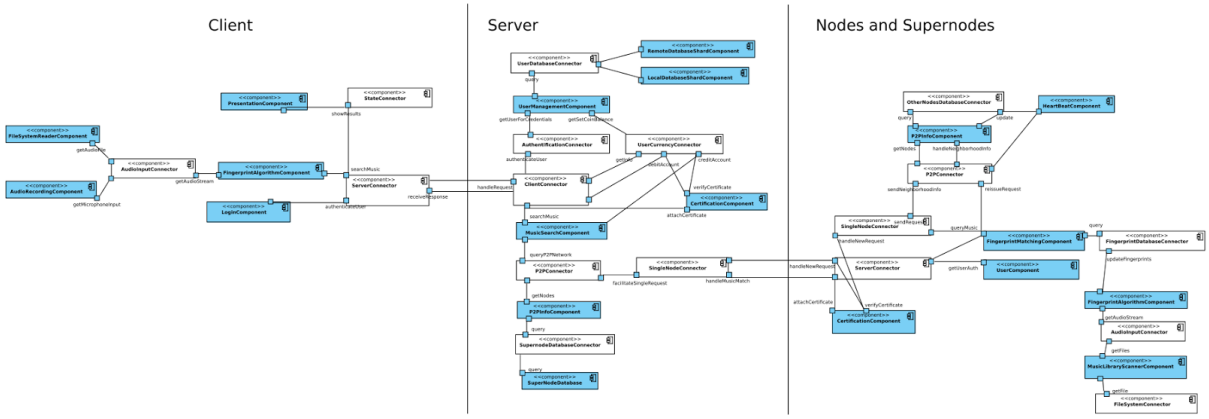
The module for the peer nodes contains the implementation units of the user-driven peers as well as the code base for the supernodes because these two types share the same code base and differ just in the configuration. This module contains a submodule for the authentication of the peer (for identifying which user runs this peer). Another submodule contains the search algorithm which searches through the fingerprints to find an answer to a request. This is the fingerprint-matching submodule which therefore has a dependency on the db.fingerprints module. Another module which depends on the db.fingerprints submodule is the library-mgmt module which contains the logic for tracking the users music collection and extracting the fingerprints of the songs. One of the most important modules is the P2P-mgmt module which handles the connection and traffic to and from the other peer nodes. Therefore it stores a list of connected peers by using the classes in the db.peer-list submodule. The module db and its submodules contain the database access objects for storing and retrieving the objects to and from the database.

#### server

The server module contains the implementation of the main server of the application which consists of a module for the management of the users (user-mgmt) including the registration, coin account management and the authentication which has its own submodule. Further more there is the P2P-mgmt module which handles the forwarding of the request which are signed using certificates from the certificate-mgmt module. It also maintains a list of the supernodes which are stored persistently in the database through the database access objects of the db.supernodes module. For storing the users and their coins the user-mgmt module also has a dependency on a submodule of the db module, namely the db.users module containing the database access objects.

## Component & Connector view

For SWAzam, we created a component & connector view. Since the meaning of each component and connector is maybe not obvious at the first glance, they are explained in alphabetical order underneath the figure. Since we treat connectors as first-class citizens, they are only distinguishable from connectors by their background color - white, while components have a blue background.




## Client

### AudioInputConnector

A user is allowed to use a music sample coming either from input via microphone or from a local mp3 database. The AudioInputConnector serves as a gateway for both of these methods, connecting the two respective components with the backend, the FingerprintAlgorithmComponent. For the FingerprintAlgorithmComponent, the audio input is only accessible via the one method “getAudioStream”.

### AudioRecordingComponent

This component handles the recording of audio for music recognition through the device’s microphone. It is only connected to the AudioInputConnector, which further connects it to the FingerprintAlgorithm.

### FileSystemReaderComponent

This component handles the recording of audio for music recognition from the devices’s filesystem. It is connected to the AudioInputConnector, which connects it to the FingerprintAlgorithm.

### FingerprintAlgorithmComponent

The FingerprintAlgorithmComponent handles the computation of the fingerprint of the raw audio file coming from the AudioInputConnector. The fingerprint is then forwarded to the ServerConnector.

### LoginComponent

The LoginComponent handles the login of the user with his credentials. It is only connected to the ServerConnector

### PresentationComponent

Since this client serves as our frontend, the PresentationComponent is responsible for displaying all the information it gets from the StateConnector to the user, through a GUI.

### ServerConnector

This connector handles all requests and responses of the server. Whenever a response is received, the StateConnector is updated and the user is presented with the results. Which physical server is actually connected is decided by DNS and the load balancing. This and

the behaviour of the connector when the server experiences a failure during the handling of a request is described later in this document.

StateConnector
The StateConnector is responsible for transmitting the current state of the program (e.g. login failed, fingerprint was recognized, etc.) from the server and all other components to the User Interfest manifested in the PresentationComponent.

### Server

AuthentificationConnector
The AuthentificationConnector is responsible for forwarding authentification requests of a user to the UserManagementComponent.

CertificationComponent
Since all requests to or from the P2P network are signed, in order to provide information about the user issuing the request or the user that found a match (and is therefore awarded with a coin), as well as to guarantee the integrity of the request, a the CertificationComponent is needed. It handles the cryptography and is exposed to other components via its two methods “verifyCertificate”( for verifying certificates for incoming requests) and “attachCertificate”( for attaching certificates to outgoing requests) to the P2P network, respectively.

ClientConnector
This connector handles all incoming and outgoing requests and responses to and from a single client that is connected to this physical server.

LocalDatabaseShardComponent
This component represents the part of the database that is locally stored on the physical instance of the server (the “shard”). More of this is explained later in the document.

MusicSearchComponent
The MusicSearchComponent receives a fingerprint from a user via the ClientConnector, signs request with the server certificate and forwards it to the P2P network via the P2PConnector. For incoming responses to issued requests, it rewards the user with a coin (via calling “creditAccount” in the UserCurrencyConnector).

P2PConnector
The P2PConnector serves as a gateway to the P2P-network. It receives signed incoming requests from the ClientConnector, gets some nodes from the P2P-network from the P2PInfoComponent and forwards the request to each of these nodes via the SingleNodeConnector.

P2PInfoComponent
This component provides info about the supernodes in the P2P-Network, providing information about latency, usage, and other statistics.

RemoteDatabaseShardComponent
This component represents the part of the database that is not locally available on the concrete server but on some other server in the network. More of this is explained later in the document.

SingleNodeConnector
The SingleNodeConnector handles all requests and responses from and to a single node in the P2P-network.

SuperNodeDatabase
This database stores information (IP, port, etc.) about all supernodes. As explained later, it is distributed between all servers in the network.

SupernodeDatabaseConnector
This connector handles the connection from the P2PConnector to the distributed SuperNodeDatabase. It exposes the single method “getNodes” where a list of supernodes and their metadata is returned to the P2PInfoComponent.

UserCurrencyConnector
All actions that affect a user’s coin balance are handled via this connector that is responsible for answers from the P2P network (via the MusicSearchComponent, these are credited with a coin) as well as requests from a client (via the ClientConnector, these are debited with a coin). For these two actions, verification of the attached certificate is necessary to ensure integrity, the connector is therefore connected to the CertificationComponent.
Besides of this, it also handles simple requests for reading the balance of a user.

UserDatabaseConnector
This connector forwards requests for actions concerning the user (crediting, debiting, authentification) to the concrete shard of the distributed database where this information is stored.

UserManagementComponent
The UserManagementComponent is responsible for all actions concerning a user. It contains the business logic for coin accounting and user authentication.

### Nodes and Supernodes

Nodes and Supernodes are both described uns the same elements in the Component & Connector diagram, even though certain components are inactive for supernodes (fingerprint matching) and some configuration values differ in the details, because they both contain the same source code and only differ in their configuration.

AudioInputConnector
The AudioInputConnector, analogously to the AudioInputConnector in the Client handles the forwarding of single audio streams and metadata to the FingerprintAlgorithmComponent, where the fingerprint of the sample is generated.

CertificationComponent

Since every node that provided the correct match to a fingerprint is eligible for a coin reward, they all sign their callback to the server. In this way, the server is able to verify the identity of the node. The CertificationComponent also checks the attached server signature for each request so that it only handles valid requests.
The signing functionality of the component (“attachCertificate”) is only needed when sending a match response to the server (for being able to verify the validity of the sent user credentials) whereas verifying the certificate attached to a message (“verifyCertificate”) is needed when handling a request from the server via the ServerConnector as well as handling a request from another node via the SingleNodeConnector for being sure that only valid requests are handled.

FileSystemConnector

This connector handles the basic I/O operations via the filesystem. It is called by providing the name and location of a file from the MusicLibraryScannerComponent and returns the raw audio stream of the given song.

FingerprintAlgorithmComponent

This FingerprintAlgorithmComponent provides the same service as the FingerprintAlgorithmComponent already described in the client.

FingerprintDatabaseConnector

The FingerprintDatabaseConnector saves new fingerprints when the peer’s local music library changes (via the FingerprintAlgorithmComponent), and it is also used to query the database with samples from matching requests. The implementation description of this connector is given later in this document.

FingerprintMatchingComponent

This component provides the logic for querying the database to find a fingerprint provided by the server (via the ServerConnector) or another node (via the SingleNodeConnector). In case there is no match (or no database at all, when the concrete node is a supernode) the request is routed to another node in the P2P-network via the P2PConnector-connector.

HeartBeatComponent

The HeartBeatComponent sends heartbeats to connected nodes at regular intervals. In case the node is not reachable, the local node database is updated with this info.

MusicLibraryScannerComponent

A user’s music library is scanned during the initial configuration and also when files are changed in watched folders on the peer’s hard disk. This logic is encapsulated in the MusicLibraryScannerComponent - it accesses the raw audio files via the FileSystemConnector and then sends them for fingerprinting via the AudioInputConnector.

OtherNodesDatabaseConnector

Every node has information about his peer nodes. This information can be queried and updated. The list of nodes stored in the database is updated when receiving a signal from the P2PInfoComponent, when the connections time out or new connections are opened.

P2PConnector

This connector serves as a gateway from the different components trying to get information out to other nodes (like the FingerprintMatchingComponent when it didn't find a match).

P2PInfoComponent

As described later in this document, the peer connection logic, when to drop or open connections is quite complex. All these rules are encapsulated into the P2PInfoComponent. It handles newly received node-lists ("neighborhood info") from other nodes, provides its own information as well as nodes that specific requests should be sent to.

ServerConnector

This connector handles the direct connection to the server. In case of the node being a supernode, song requests are also received via this channel. Otherwise, the connector is used as a way to inform the server about a match. The connector also handles certificates (via the CertificateComponent) and has access to the user authentication when sending a match (via the UserComponent).
This Connector, analogous to the ServerConnector in the client, also handles load balancing and server failures during handling requests. How this is handled is described in detail later in this document.

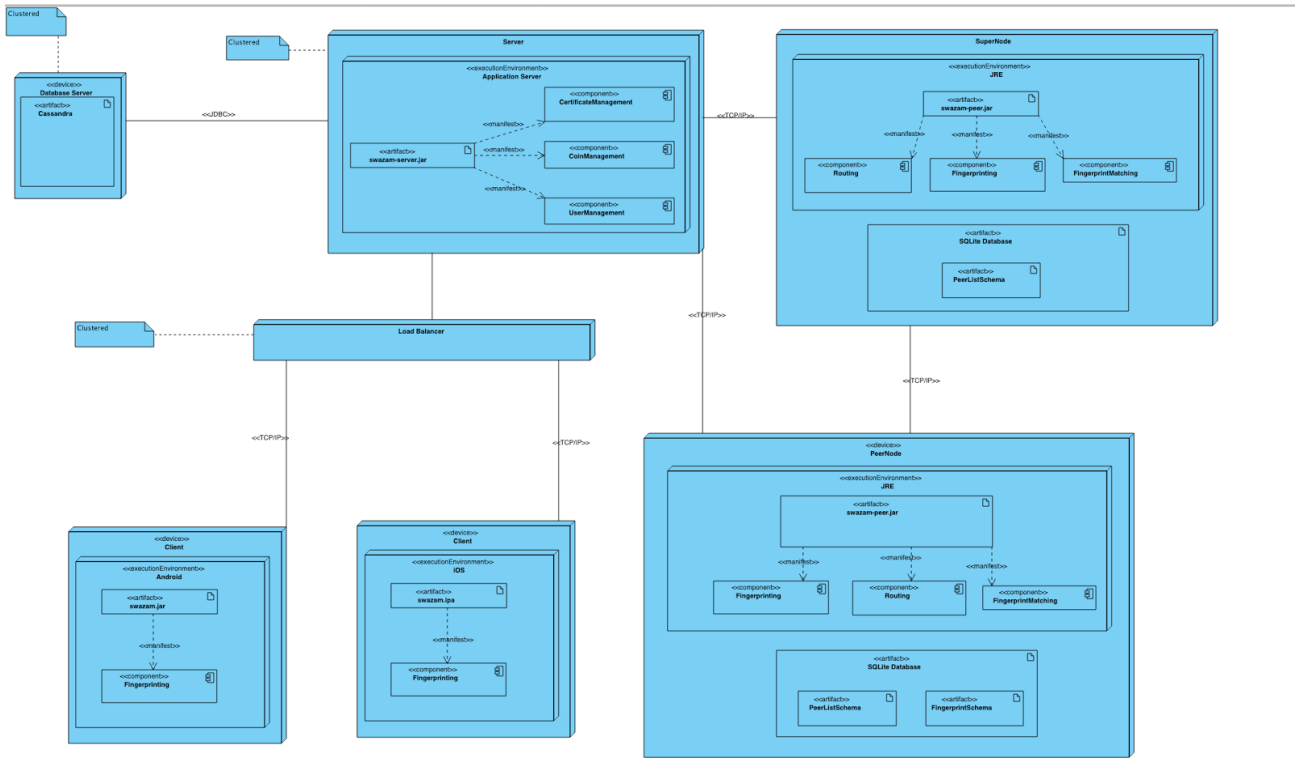
SingleNodeConnector

This connector handles the direct connection to another node. Requests for songs are received via this connector ("handleNewRequest") as well as providing metadata ("neighborhood info"), or the routing of song requests. The connector has access to signature verification (via the CertificationComponent), since only requests with a valid signature should be handled.

UserComponent

The UserComponent stores information about the user associated with the node as well as the user's certificate. It is only connected to the ServerConnector since the user's authentication is only necessary for login and the certificate is used when sending a match back for crediting a coin to the user's account.

## Deployment view




### Consistency

The artifacts in the deployment diagram correspond to the components and connectors in each of the 3 parts of the C&C view. The swazam.ipa and swazam.jre contain the elements labeled Client in the C&C view, swazam-server.jre contains the ones labeled Server, and swazam-peer.jre contains the ones under Nodes and Supernodes.

## Behavior view

To illustrate the dynamic functioning of SWAzam we created some sequence diagrams as shown below. These type of UML diagrams are well suited for showing the interaction between the different components and connectors.

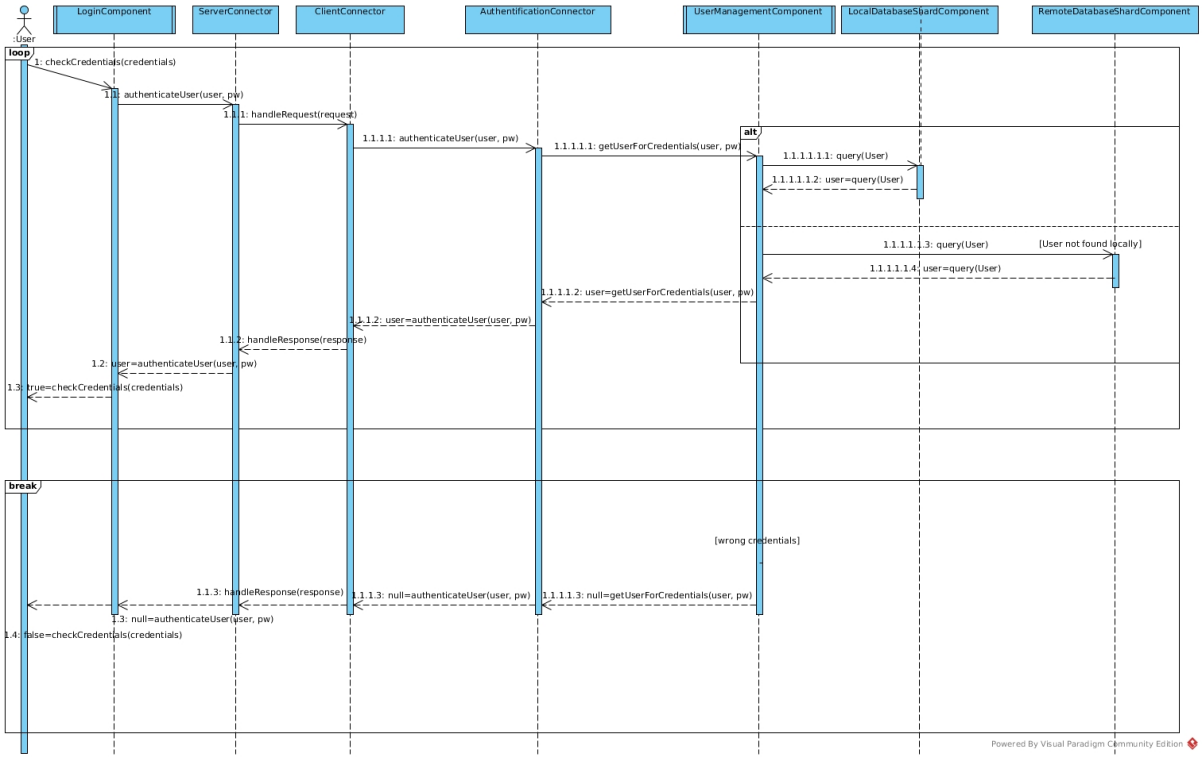
### Consistency

We strongly put emphasis on consistent diagrams. First, all interaction partners (represented as boxes in the sequence diagram) basically represent Components (passive) and Connectors (active) in the C&C diagram. The names are therefore identical. Second, as with Visual Paradigm, we did not found a way to implement IN/OUT ports in the C&C diagram to indicate the direction of control flow. Instead, we strongly adhered to the convention in our C&C diagram that control only flows from unnamed ports to named ports. All sequence diagrams follow this convention.

To get a better understanding of the details of the behavior of SWAzam we decided to divide the control flow requesting a song for recognition into *Authentication* and *Request music*.

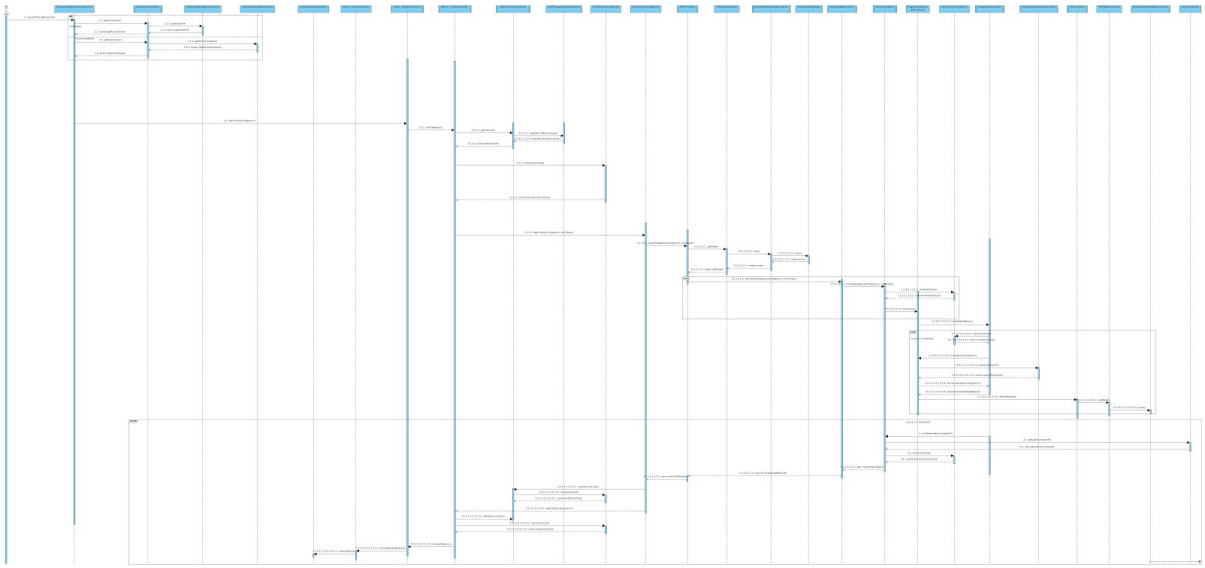
## Authentication

In order to map each request to a specific user account it is necessary for each client to authenticate the user. Since security is not of concern for this assignment we omitted sophisticated security controls.



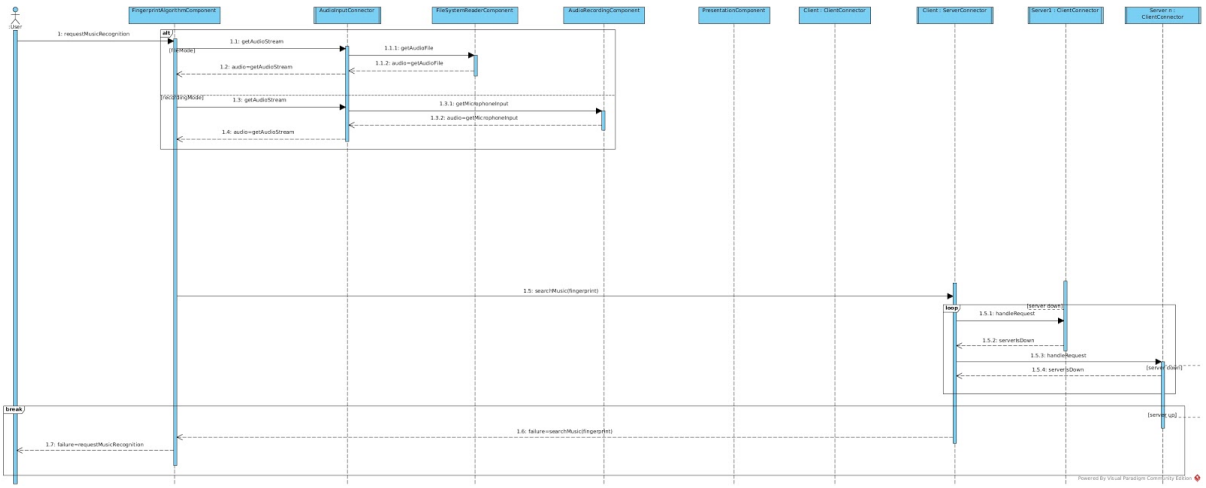

## Request music

The diagram shown below shows the bulk of the main use case for SWAzam, namely requesting music information for recognition. This diagram assumes that the execution conditions are met (enough coins, enough peers, and so on).



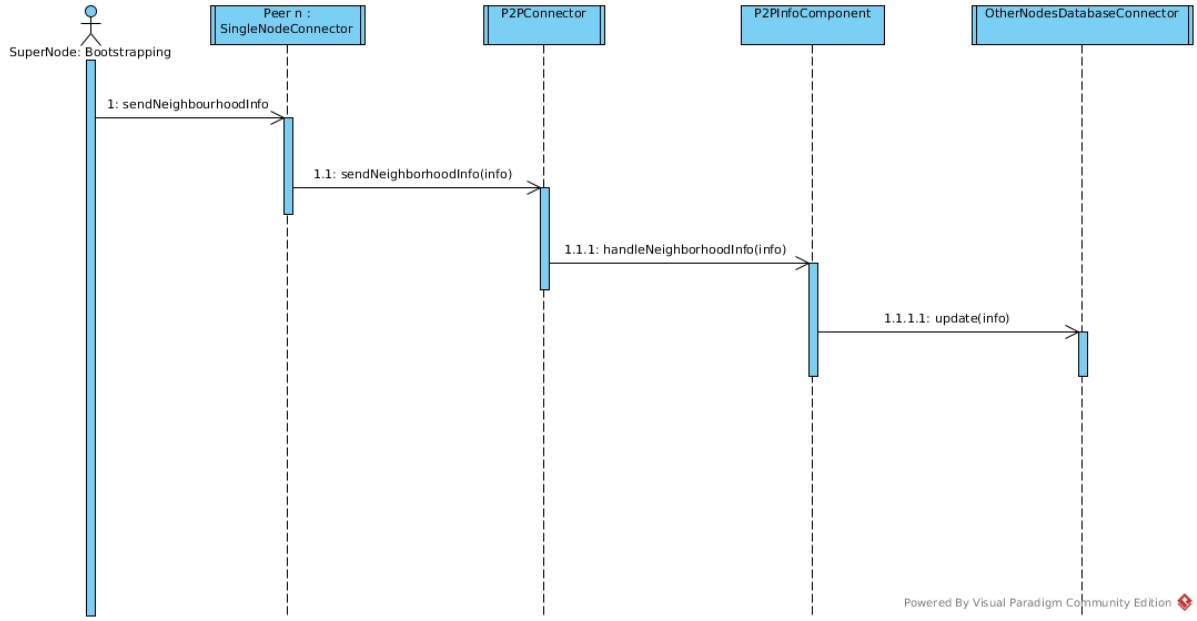
### Request Music - Server Failure

Since there are always 4 instances of the Server running in parallel the Load Balancer (Server:ClientConnector) balances requests according to some predefined load criteria and since all server instances are running in the cloud the load balancer tries to find an active Server instance. If no active instance is found, an appropriate failure message (as shown below) is sent back to the client.



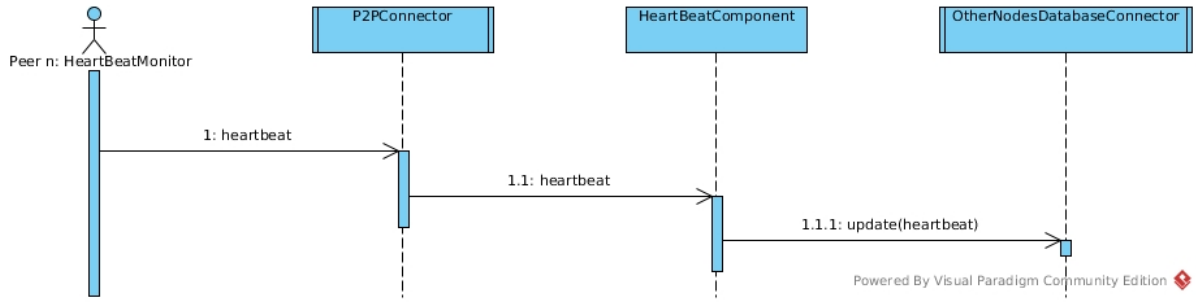

### Initial population of Peers

During the startup phase of the P2P network it is necessary to populate each node's database of peers. To accomplish this, the supernodes share a list of initial peers and contact them to update their knowledge of nearby nodes. These, in turn, contact other nodes and so on. The connections in the network of peer nodes therefore will grow exponentially.




### Heartbeat

To facilitate a self organizing network of peers all nodes will send regularly a heartbeat message to connected nodes. Each peer therefore only contacts available neighbors.




# Quality attributes

## Availability quality & tactics

Because of the high expected popularity of our system, a main goal was its high availability to the users. We therefore set our required availability rate for the whole system to no less than 99.99%. To achieve this value several architectural tactics and implementation decisions were made.

For the calculations we assume an availability of **95% for each physical server machine**. This means that we are allowed to have a **maximum downtime of 17 days and 19 hours**

per year of system runtime for the purpose of maintenance work and other unplanned disruptions of a single server.

### Database server

For ensuring the availability of the database server which is connected to the main server of the application we chose to use a database cluster with a minimum of **four** database servers running in the cluster. For the synchronization and replication of the databases on these servers we use Apache Cassandra[1], which is a database management software designed for high availability, replication among clusters and performance. We then have an availability of the database cluster of

$$A_{db-cluster}(A_{db}) = 1-(1-A_{db})^{n_{db}} = 1-(1-.95)^4 = 0.99999375 = 99.999375\%$$

where $A_{db}$ is the availability of one database server and $n_{db}$ is the minimum number of actively redundant servers in the cluster. This way the database cluster has an availability of approximate **99.999%**.

### Main server

The main server of our application is the component against which the clients authenticate, to which the clients send the requests for the searches and over which the P2P network sends the answers back to the clients. This server cluster also manages the user database and the coins account for each user. Therefore it is very important that this server has a very high availability together with the database cluster. To ensure this, we chose a similar approach to the one we took for the database cluster. We chose to run the server as a cloud based cluster of servers using Amazon AWS[2] cloud instances. The configuration of the dynamic cloud instances guarantees that there are always at least **four** main servers running concurrently. The traffic from the clients is distributed to the single servers of the cluster via a load balancer. Therefore we designed the cluster for active redundancy, just like the database cluster. So we can calculate the availability of the cluster for the main server to be

$$A_{server-cluser}(A_{server}) = 1-(1-A_{server})^{n_{server}} = 1-(1-.95)^4 = 0.99999375 = 99.999375\%$$

where again $A_{server}$ stands for the availability of the single server instances and $n_{server}$ stands for the minimum number of server running concurrently. Therefore our main server cluster also has an availability of approximate **99.999%**.

[1] http://cassandra.apache.org/
[2] http://aws.amazon.com/

### P2P network

Another very important and mandatory part of the system is of course the peer-to-peer network of the nodes with the music libraries. Due to the dynamic and user-dependant nature of our P2P network we cannot assure an exact availability rate for the user driven peer nodes. But for the stability of the network and for ensuring a controlled continuous growth of the P2P network we decided to run our own peer nodes, called supernodes, which are directly connected to the server and used as an entry point for the server to the P2P network. These supernodes don't have a music library database but ensure the stability of the network through their routing algorithms. To have the same level of availability we are running again at least **four** supernodes. In addition, for ensuring that no music requests get lost, the main server always forwards the music queries to **two** supernodes. Therefore our calculation is the same as above, namely

$$A_{supernodes}(A_{supernode}) = 1 - (1 - A_{supernode})^{n_{supernode}} = 1 - (1 - .95)^4 = 0.99999375 = 99.999375\%$$

where $A_{supernode}$ is the availability of one single supernode and $n_{supernode}$ is the minimum number of supernodes which we are running at any point of time.

### Peers

For establishing fault tolerance in the P2P system for an increasing reliability of the network we designed the peers in a way that they can detect when another connected peer went offline. This is implemented via a heartbeat functionality, i.e. that the peers periodically send an "i am alive" message to every other peer they are connected to. When a peer didn't receive such a message from a connected peer for a period of time he considers this peer to be offline and establishes a connection to another peer from its known-peers list.

### Overall availability

Because we cannot foresee the exact availability of the user-driven peers, we cannot include them into the calculation of the availability of the overall system and so the calculation is as follows:

$$A_{total}(A_{db-cluster}, A_{server-cluster}, A_{supernodes}) = A_{db-cluster} \cdot A_{server-cluster} \cdot A_{supernodes}$$
$$= 0.99999375^3 = 0.99998125 = 99.998125\%$$

So the overall availability of our system is approximately **99.998%**.

## Scalability quality & tactics

Scalability was one of our key considerations in our architectural process. The following subsections will describe the steps we took to avoid scalability problems.

### P2P

One of the main strengths of a peer to peer network (if designed properly) is scalability. We're crowdsourcing our need for processing power (and at the same time vastly increase the number of findable songs) by using peer to peer technology for music recognition.

To get the best possible performance out of the P2P network:

- Each peer maintains connections to 8-12 other peers at all times.
  If a connection gets dropped and the node has less than 8 connections, it will actively try to open more to reach 10 and will further accept more connections from other nodes until it reaches 12.
- For supernodes (described below) those numbers are considerably higher.
- Connections between peers time out after about 3 hours (plus/minus a random value up to half an hour). That way, the network constantly reorganizes itself which should help prevent peers from having an advantage by staying close to a supernode.
- When choosing which node(s) to connect to, peers will prefer nodes that showed good performance in the past (fast response time), but they will also try to pick at least some less-performing peers to avoid clustering.
  Also, the selection process discriminates against the peers that they were connected to too recently.
  Supernodes are only picked with a very low probability.
- When connecting to another node, the peers exchange a random subset of their known peers which they then add to their pool of known peers.
  Each entry in the list of known peers list has a timeout value (which will be used to clean up that list periodically) and a performance indicator (response time, availability, ...).
- Requests have a time to live (TTL) that decrements with each node it reaches and that works similar to the one in TCP/IP.
  Our servers are responsible for setting that value when issuing a request, but, in order to prevent abuse, the peer software decreases the TTL to 7 if incoming requests exceed that limit (The limit is set low because the number of involved nodes grows exponentially with each hop the request takes. The limit may be modified in future software versions if deemed necessary)
- Each request gets a random ID by the issuer (one of our servers). Peers cache the IDs of requests they already processed for five minutes (in RAM) to avoid requests being processed more than once by the same node.
  Request IDs are random 32bit integers. In the rare event they do collide, only nodes that already processed the other request will skip the new one.
- Requests are processed locally first and if no match was found are forwarded to 4 other peers (which are randomly selected from the list of currently active connections).
- Every Request stores a callback IP and port. If a peer finds a result locally, instead of forwarding the request it sends the response to that IP and port. The response also contains information on the account that peer belongs to (for coin billing reasons).
  The server that initiated that request listens at that IP/port and processes responses on a first come first serve base. If a request hasn't been answered after 5 seconds, it has failed and an empty result will be sent to the client.
- Supernodes are our entry points to the P2P network. They’re operated by us and placed close to important internet backbone nodes. They run the same software as

regular peers, but don't have a local fingerprint database and therefore only delegate requests. They maintain connections to about 500 (instead of 10) peers and get the requests directly from our application servers.
- Our servers maintain a persistent connection to the supernodes and issue each request to two of them (selecting them randomly) to avoid requests getting lost at unavailable supernodes.

### Servers

Our application servers provide the link between the client application and the P2P network.
They also take care of authentication and coin management.
Our servers are placed behind load balancers and the application servers are stateless (all non request-local data is persisted to the database).
And we've selected Apache Cassandra as server-side database engine. It's widely used amongst huge internet corporations[3] and seems to outperform most of its competitors (scaling almost linearly[4]).
Multiple load balancer instances are assigned to one DNS record to facilitate round robin DNS.
The load balancers themselves distribute the requests between the application servers taking application server load (and potentially offline servers) into account.

## Modifiability quality & tactics

The following subsections describe implementational and/or architectural details that might change in the future and the measures we took to to make those changes manageable (or the reasons we didn't. We tried our best to keep the different modules as loosely coupled as possible, but some core infrastructure can't be replaced easily.

### Product end of life

We make sure only users with positive coin balance can issue requests on the application servers.
And to make sure no one bypasses our servers, they add a signature to each P2P request. Only requests with a valid signature will be processed (and forwarded) by the nodes.
This also makes DOS attacks against the P2P network considerably more difficult.
But when the project comes to its end of life and we decide to stop our server infrastructure, we intend to publish the certificate details. From that moment every node can issue requests and our service becomes a pure self-organizing peer to peer network.
The implication of this is that the coin system becomes ineffective. But it allows peers to operate the network beyond the end of our involvement and might - if properly communicated - give an additional incentive to use our product instead of others.

[3] Apple, CERN, IBM, Netflix, Twitter, … - See http://planetcassandra.org/companies/
[4] Solving Big Data Challenges for Enterprise Application Performance Management - Rabl et. al.: http://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf

### Additional client support

Growing market share will require us to port our client software to additional platforms. This process should be straightforward. Apart from the UI, only audio recording and the fingerprinting algorithm need to be implemented.
If there are platforms where running the fingerprinting algorithm might not be suitable (e.g. due to performance/battery reasons or inside a web browser's javascript sandbox), the application servers would have to be extended to do optionally do the fingerprinting themselves and the client would send encoded audio in the request.

### Fingerprint database rebuild

We make a certain tradeoff between search accuracy and storage overhead when creating the fingerprinting database on the peers by only storing every 4th fingerprint created by the algorithm. This results in saving 75% storage overhead on the peers.
Peer node operators might however want to rebuild their fingerprint database to further decrease the disk space usage or to increase search accuracy in the future.
Decreasing fingerprint granularity to let's say 8 would be as simple as deleting every second fingerprint from the database.
Increasing the granularity however requires the application to run the fingerprinting algorithm on the whole database again and therefore will take some time to execute.
From an implementational standpoint however that feature won't add a lot of complexity.

### Change of the fingerprinting algorithm

Replacing the current fingerprinting algorithm with another one is probably the most severe change we could make to our software. It would certainly require us to support both algorithms in parallel (and therefore have both fingerprints in each request) as all peer and client installations needed to be updated and peers would need to rebuild their fingerprint database.
Switching to another algorithm should therefore only be done if the current algorithm turns out to show significant disadvantages.

### Expansion to additional markets

If the service turns out to be sufficiently popular, we might consider replicating the server stack to other data centers in other parts of the world. Client requests would then be processed by the nearest data center and overall response time should decrease.
Also, the application servers could be configured to only query supernodes close to them to further improve response times.

## Performance quality & tactics

### Server

The application server logic is pretty straightforward. Other than some reading/writing to the database (in our case the highly performant Apache Cassandra) its only real job is to transform HTTP search requests to P2P requests, sign them and wait for their response. None of these are resource intensive and except for the waiting part nothing really takes a long time.

The time spent waiting for a response depends on how quick one is found. Popular songs will be found almost instantly in most cases. However, in the worst case no response is found and the server will wait for 5 seconds before responding to the client.
The cause for that issue is inherent to P2P networks and can only be addressed by fine-tuning the timeout value.

### P2P Network

The performance of a peer to peer network depends on the time it takes individual peers to process requests plus the connection delay between two nodes.
Therefore, we tried to focus on local search performance (as seen in the following subsections) as well as preferring locality (i.e. low response time) in peer selection.
As we also depend on supernodes as our only entry points to the P2P network (because we rely on a server infrastructure to issue P2P requests), it's also important to have them connected to as many other peers (as evenly distributed through the network as possible)

### Building the fingerprint database

Nodes have to be able to search considerable amounts of data in a very short time. The fingerprinting algorithm yields 256 subfingerprints for each 3 seconds of audio (a subfingerprint has 32 bits which results in 1kb of raw fingerprint data per 3 seconds of audio.
The research paper describing the algorithm[5] states 250 million subfingerprints for a database of 10000 songs. And each fingerprint has to reference the song (and its position within the song) it references to That's at least 3GB of extra data (using 32bit integers for each of the three values). But to be able to efficiently search that amount of data we need to add indexes (which likely doubles the size of the data again).
In a few small tests using SQLite as storage engine we ended up with a fingerprint db file of about 12GB (which isn't that big of additional overhead considering that we used some other indexes to be able to efficiently find and modify/delete fingerprints of specific songs). But 12GB of extra data might be considered a lot for a music database of maybe 50GB (at 128kbit/s MP3 or 25GB for 64kbit/s AAC[6]).
A fingerprint is basically a representation of the frequency spectrum at that position of the song. So fingerprints close to each other will most likely be similar.
As we can't rely on only matching exact fingerprints anyway (encoding errors, background noise when recording, ...), we decided to only store every 4th fingerprint (and therefore reducing the size of the fingerprint database by 75% without losing too much accuracy). However, to make sure we don't throw away the same data everywhere, each node randomly selects a 'skipOffset' when they're started for the first time and only stores the fingerprint if position % 4 == skipOffset.
This results in a fingerprint database size of about 3GB for 50GB of music which should be enough to convince more people to install the node software.
Another advantage of a small fingerprint database is that the operating system can cache the index more effectively, which further speeds up the search.

[5] Solving Big Data Challenges for Enterprise Application Performance Management - Rabl et. al.: http://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf
[6] The algorithm stores 256 fingerprints/3sec, we know we have 250mio fingerprints so we can calculate the music in that particular DB amounts to roughly 3mio seconds which is a little less than 30GB at 128kbit/s

### Local fingerprint search

Searching for matches is the other half of the fingerprinting problem. In the previous section we described how to effectively store the fingerprint data, this section will cover how to find it again.
When a search request comes in, the node first checks if it's properly authenticated (i.e. if the server has signed it) and then tries to find a match locally.
It picks a random subfingerprint and queries the database for exact or close matches (using the hamming distance to find possible neighbors).
We only accept a hamming distance of up to 1 for that subfingerprint because the number of possible neighbors grows almost exponentially for bigger numbers. If the chosen subfingerprint was too noisy to find a match, the request will simply be forwarded to other nodes which will select other subfingerprints as their candidates, thus increasing the likelihood to find a match while at the same time keeping resource consumption low on one node.
If matches are found, the surrounding fingerprints for each of them are queried from the database and the overall hamming distance of that match is calculated. If it's below a certain threshold, it's considered a match and the issuing server is notified with the results (using the callback IP/port mentioned in the search request). Results are processed in a first come first serve order, so once a match is found, all the other local matches are discarded. If no match was found, the request is passed on to other peers.
In our tests the algorithm was able perform the search described here in about 30ms on a database filled with 250mio subfingerprints.

### Passing on requests to other peers

When no matches were found after searching locally, a peer randomly picks 4 of the other nodes it's connected to, decrements the request's TTL and passes the request on (if the TTL is still greater than zero).

## Testability quality & tactics

For the sake of a reliable system another important quality is testability. A system is more unlikely to have bugs the more it is able to detect those bugs as soon as possible through automated testing during the development. One of the easiest ways to get a testable system is to limit its structural complexity during the architectural phase of the development and therefore we made a view decisions to ensure a modular, loosely coupled system. Another advantage of a modular system is, that the input and output of such modules can be observed and even modified or artificially produced which is necessary for testing purposes.

For getting a modular system we tried to split logically different functions into different modules or even different physical machines. Therefore we split the database from the main server functionality, in fact even to an entire different cluster of server machines. Another decision at design time was to use supernodes in the P2P system which are directly connected to the server and further forward and distribute the requests to the peer nodes. This decision enables us to monitor and/or simulate the incoming and outgoing traffic of the P2P network at a single point in the architecture. Furthermore it allows us to

monitor the scaling and growth of the P2P network on the supernodes and use this as an input to testing and evaluating the P2P network building algorithms.

Because the communication of the different physical devices is done over the network using a standard technology, namely TCP over IP, it is easy to monitor the traffic and analyze it for detecting faults of the individual units. It is also possible to use a recorded TCP stream from one to another component and replay this stream which enables the creation of automated tests of the different server and client systems.

In addition to those “low level” design decisions we also tried to incorporate the principle modularity in the structure of the implementation units (see Module view). We encapsulated the modules according to their functionality to control and observe their input, output and inner state for easing the creation of automated tests.

# Appendix A: Experience Report

Compared to the first assignment, designing a project from scratch is easier in our opinion. We didn't have to get acquainted with an existing code base (which was roughly 300k lines of code for MediaWiki) and we only needed to be consistent with our own documentation and not with actual code (which of course changes as soon as you start coding, but then you still have the advantage that the same team does both the coding and the design).
It was the first peer to peer project any of us had to design so we had to some research on possible pitfalls. Also no one had an idea of how audio fingerprinting worked in general (and the proposed algorithm specifically). But we managed to figure out those details and even wrote some code to test local lookup performance.

## Teamwork

Even though not all of us knew each other at the beginning of the semester, we managed to work together pretty well.
For this assignment, we first had a team meeting where we tried to find an agreement on a general architecture as well as some of the details.
After that, we split up the project into smaller subtasks which were assigned to individual team members (we used github issues for that as we use a private github repo for most of our documentation anyway).
And we started a Skype group chat back at the beginning of assignment one to discuss details that hadn't come up before (with the occasional conference call close to the end of the assignments).
For the first assignment we spent a lot of time composing a definitive list of components (which was a huge blocker as we had to be consistent to the actual source code and we weren't always sure about what qualified as component and what didn't). This time we didn't have that problem as we were starting from scratch and also there was more work to be done that didn't fully rely on the component relations.
All in all, this project allowed for quite efficient teamwork (much more so than assignment one).

## Tools

For modelling, we used Visual Paradigm. Some of us had used it before and even though is quite resource intensive, it proved to be a good asset in solving this (and the previous) assignment.
In January, there was a new major release (12.0) which creates project files that can't be read by older versions. That required all of us to update to the current version (which was just a minor issue but a little annoying nevertheless).
For the final documentation we used Google Docs as it's extremely simple for people to edit a document simultaneously. It has some drawbacks (no vector graphics support, …) but for our purposes it did the job reasonably well.
All the other documents, diagrams and other files ended up in a private github repository. Git is far less effective with managing binary files than it is with text formats, but we didn't expect a whole lot of changing binary data, so we picked it anyway.

In the end, the repository added roughly 80% to the raw data size (the whole repo - including the .git/ folder - was about 25MB), so git seems to be a valid tool for that kind of job.
One problem we had (mainly during the first assignment as there were some .vpp files that got edited by pretty much all of us) was update concurrency for Visual Paradigm files (which are essentially SQLite3 databases). We ended up using some form of manual locking by announcing changes to those files in our Skype group chat.

## Difficulties

### Mixed architecture

One problem we had was that we had to design a hybrid of a peer to peer network coupled with a server/client structure. The assignment description states that clients are not part of the P2P network. As most mobile devices don't have the most reliable network connectivity, it's probably better to let them talk to a reliable server infrastructure than to have the other end be unreliable as well.
That's why we've decided that clients only talk to our servers which then forward the requests to the P2P network. That also makes it easier to implement reliable coin management.
On the other side it takes away a lot of the advantages of a peer to peer network. We can only keep connections to a finite number of peers as entry points for our server infrastructure which in turn means that there are some peers closer to our servers (and therefore get hit by a lot more requests) than most of the others.
We tried to circumvent that issue by letting our supernodes (which are our entry points to the network) connect to as many peers as possible (currently we defined the value to be 500, but that can be changed easily - requests will be forwarded to 4 random ones of them), by setting a timeout to each peer connection (after which they will connect to another node instead) and by making sure processing on the peers is as light-weight as possible. They expect the first subfingerprint they compare to have a low hamming distance.
A consequence of this is that sometimes peers won't find a match even though they could if they were a little more tolerant. The request will simply be forwarded to other peers and eventually one of them will select a subfingerprint with little to no noise and will therefore find the match.

### Picking numbers

One thing we had to do during the design phase was to come up with some numbers to describe the behaviour of our peer to peer network (peers keep a connection to about 10 others, requests are forwarded to 4 peers if no local match was found, requests die after 7 hops, etc.).
Most of these numbers were educated guesses with some simple calculations to find out if the numbers add up (for example, given the numbers above, requests that find no match die after about 5000 nodes - in the worst case). Most of these numbers have yet to be tested in practice, but almost all of them can be changed with a simple software upgrade.

### Assignment description

We found some minor issues in the assignment description. For example, some sections still mention us having to implement the software. That seems to be a relic from previous semesters.
But other than that the assignment description was mostly clear.

## Conclusion

All in all the project was quite interesting. I think we've learned a lot about the inner workings of peer to peer networks and audio fingerprinting.
The size of the project seems to be fine for a team of five (especially because the workload is easy to split up), but I think all of us are glad we didn't have to actually implement the software.
Instead we focused on refining the architecture and to think of every possible pitfall and how to prevent them.

# Appendix B: Test scripts

While choosing the fingerprint database storage mechanism and designing a search algorithm we wrote two small test scripts to try out if what we designed would actually work (and the amount of resources it would need).
For simplicity we used Python for those scripts which means that it's quite likely that a Java implementation is a lot faster.

## Indexing script

The first script[7] creates a SQLite database and fills it with loads of random test data. We've tested the algorithm for up to 250mio subfingerprints (which is the number mentioned in the paper). It takes about an hour to fill the database and almost another hour to create the fingerprint index.
To speed things up, we decided to only store every 4th subfingerprint (which results in 4 times faster index creation as well as 75% size reduction). And it's easier for the OS to keep the index in the disk cache, which also results in faster searches.

Example command line:

```
$ ./createTestData.py fingerprints.sqlite3 1000
```

## Lookup script

The second script[8] does the lookup. It first selects a random subprint and tries to find it in the database (with a hamming distance of up to 1 which means that it actually does 33 lookups).
For each possible match it then calculates the hamming distance of the whole fingerprint. If it's below a certain threshold, we found our result.

Example command line (finds results for the given print with a distance of up to 10):

```
$ ./searchTestData.py fingerprints.sqlite3 \
      deadc0de7bef4d8a374d56d0f0faaa2d 10
```

[7] http://pastebin.com/yZ6TjqJv
[8] http://pastebin.com/EaFWSwgy

# Appendix C: Diagrams in vector format

Clustered

<<device>>
**Database Server**

<<artifact>>
**Cassandra**

<<JDBC>>

Clustered

**Server**

<<executionEnvironment>>
**Application Server**

<<component>>
**CertificateManagement**

<<artifact>>
**swazam-server.jar**

<<manifest>>

<<component>>
**CoinManagement**

<<manifest>>

<<component>>
**UserManagement**

<<TCP/IP>>

**SuperNode**

<<executionEnvironment>>
**JRE**

<<artifact>>
**swazam-peer.jar**

<<manifest>>

<<component>>
**Routing**

<<manifest>>

<<component>>
**Fingerprinting**

<<manifest>>

<<component>>
**FingerprintMatching**

<<artifact>>
**SQLite Database**

<<artifact>>
**PeerListSchema**

Clustered

**Load Balancer**

<<TCP/IP>>

<<TCP/IP>>

<<TCP/IP>>

<<TCP/IP>>

<<device>>
**Client**

<<executionEnvironment>>
**Android**

<<artifact>>
**swazam.jar**

<<manifest>>

<<component>>
**Fingerprinting**

<<device>>
**Client**

<<executionEnvironment>>
**iOS**

<<artifact>>
**swazam.ipa**

<<manifest>>

<<component>>
**Fingerprinting**

<<device>>
**PeerNode**

<<executionEnvironment>>
**JRE**

<<artifact>>
**swazam-peer.jar**

<<manifest>>

<<component>>
**Fingerprinting**

<<manifest>>

<<component>>
**Routing**

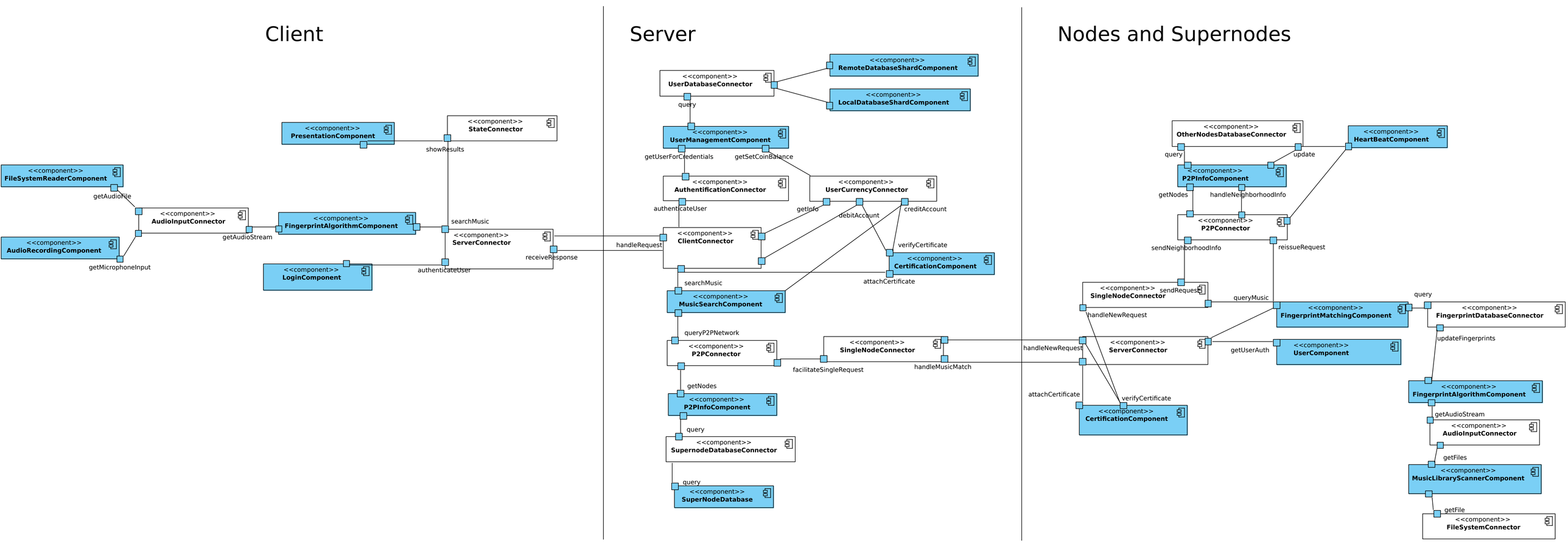
<<manifest>>

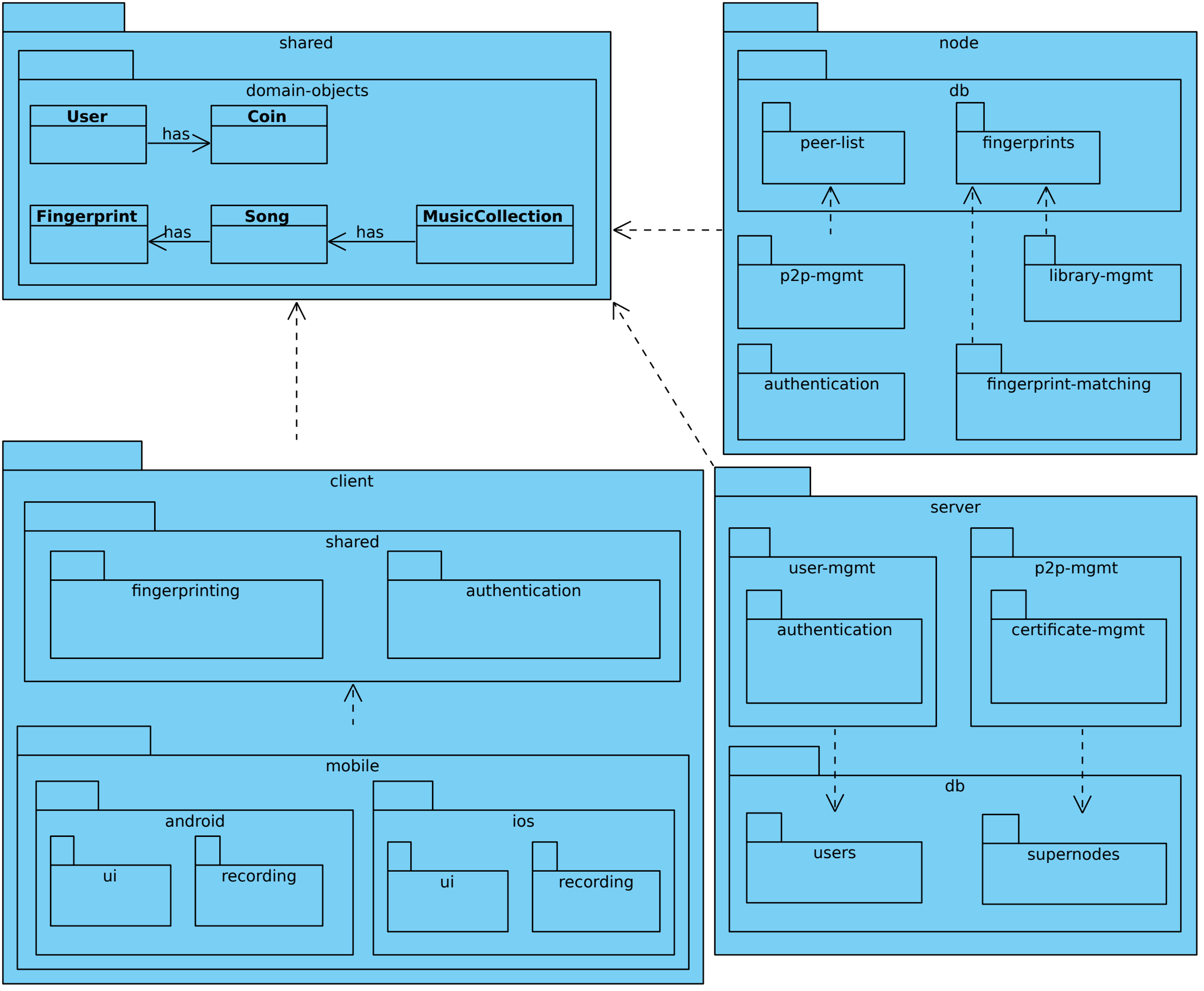
<<component>>
**FingerprintMatching**

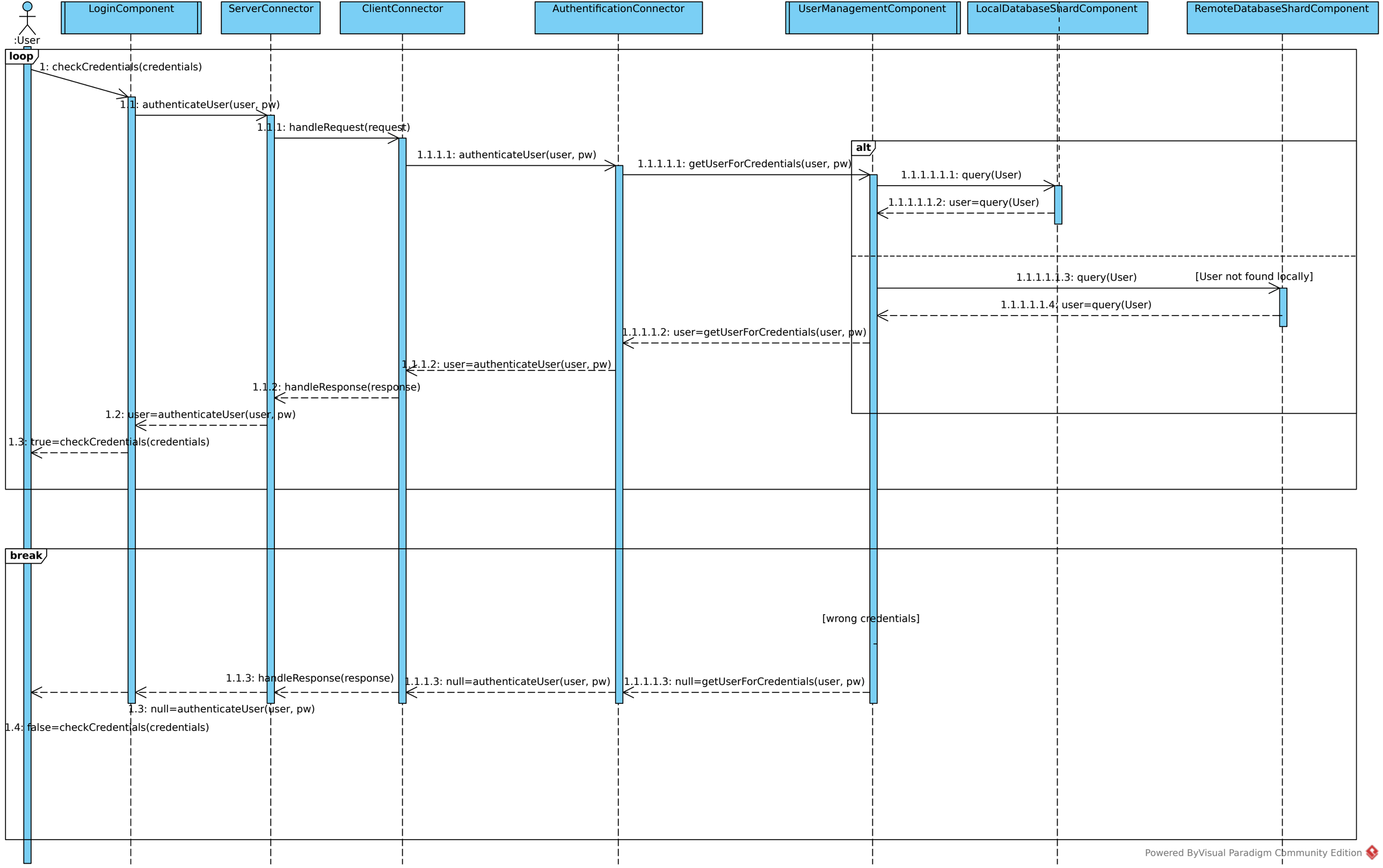
<<artifact>>
**SQLite Database**

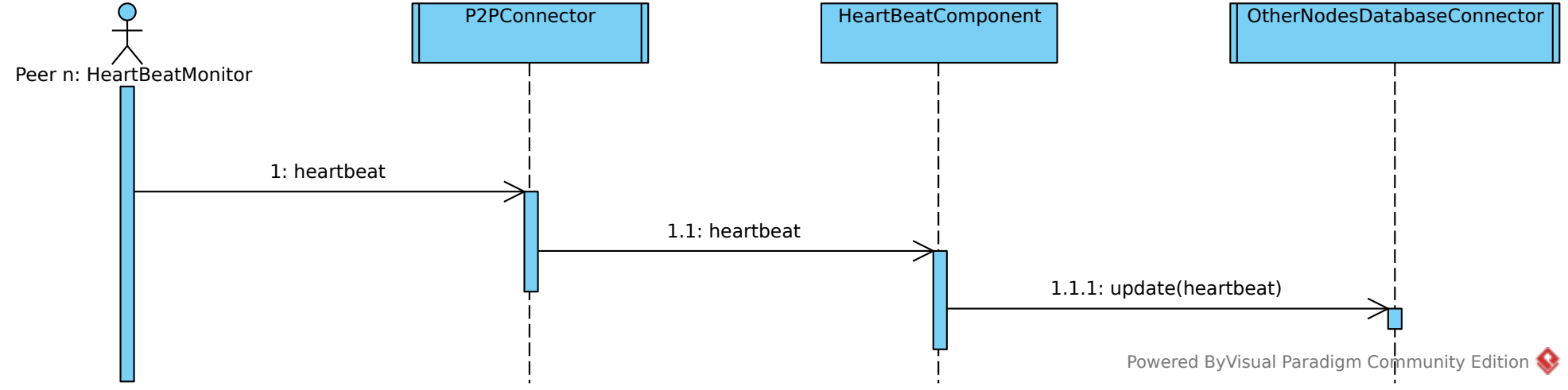
<<artifact>>
**PeerListSchema**

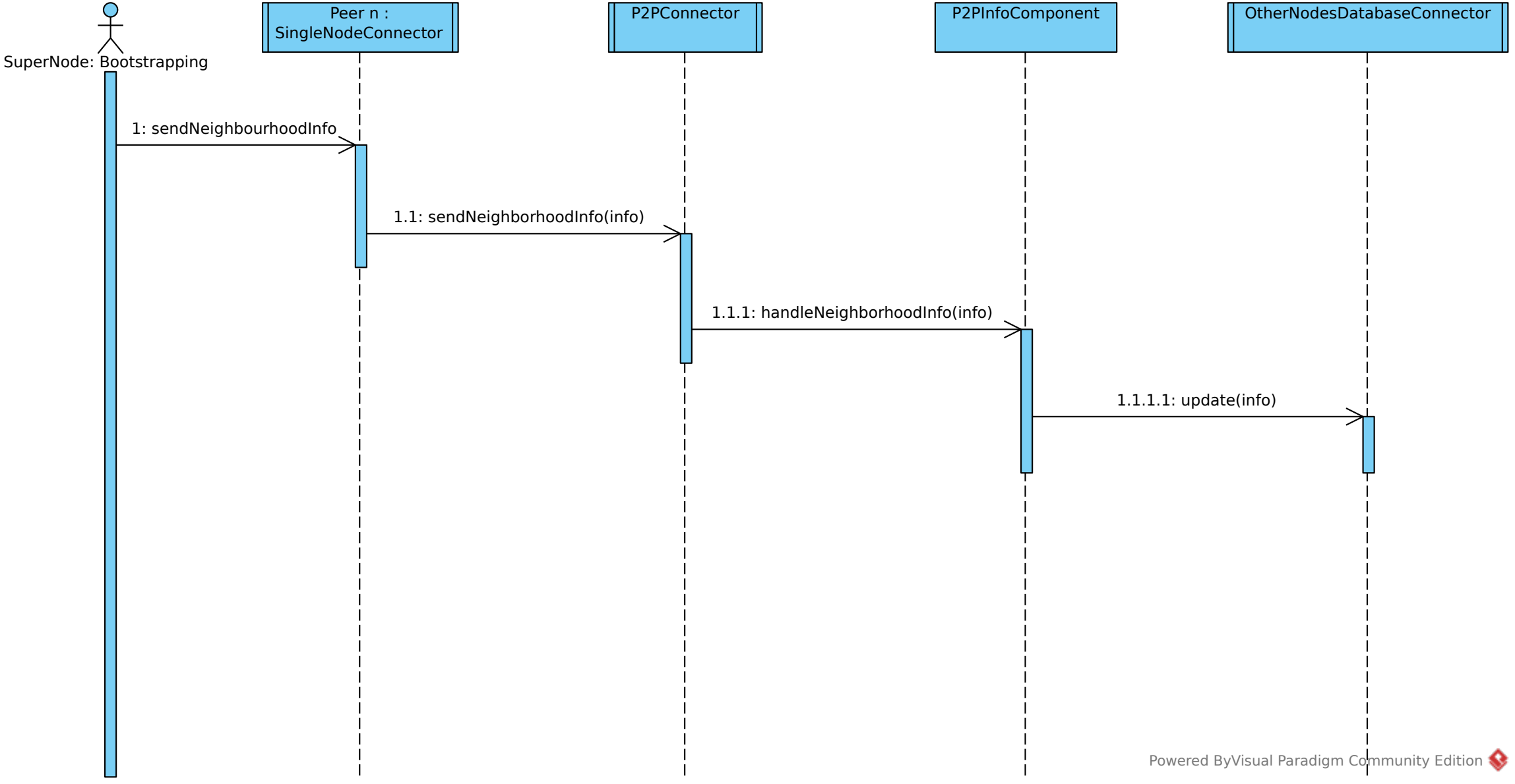
<<artifact>>
**FingerprintSchema**

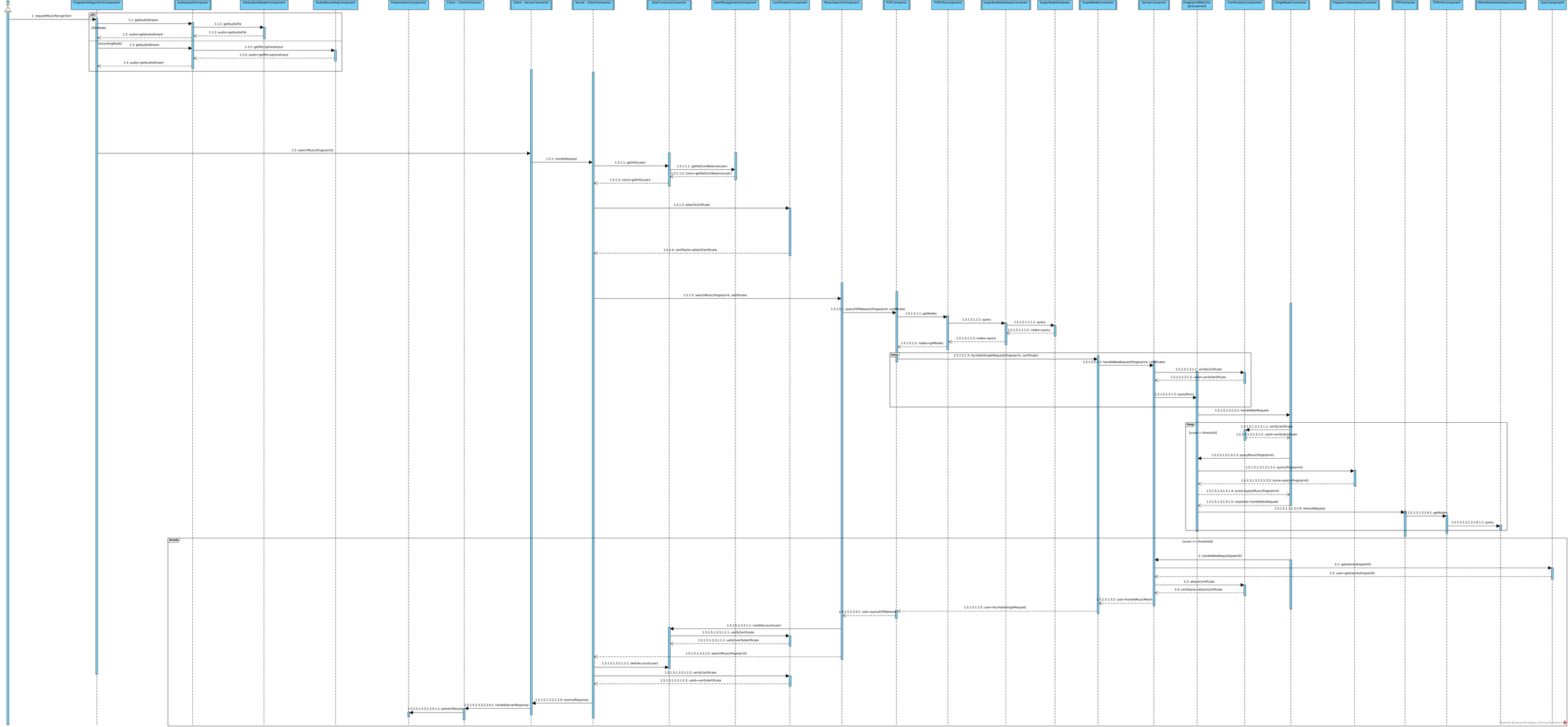

Client
Server
Nodes and Supernodes
<<component>>
PresentationComponent
<<component>>
StateConnector
showResults
<<component>>
FileSystemReaderComponent
getAudioFile
<<component>>
AudioInputConnector
getAudioStream
<<component>>
FingerprintAlgorithmComponent
searchMusic
<<component>>
AudioRecordingComponent
getMicrophoneInput
<<component>>
ServerConnector
receiveResponse
<<component>>
LoginComponent
authenticateUser
handleRequest
<<component>>
UserDatabaseConnector
<<component>>
RemoteDatabaseShardComponent
<<component>>
LocalDatabaseShardComponent
query
<<component>>
UserManagementComponent
getUserForCredentials
getSetCoinBalance
<<component>>
AuthentificationConnector
authenticateUser
<<component>>
UserCurrencyConnector
getInfo
debitAccount
creditAccount
<<component>>
ClientConnector
verifyCertificate
<<component>>
CertificationComponent
attachCertificate
searchMusic
<<component>>
MusicSearchComponent
queryP2PNetwork
<<component>>
P2PConnector
<<component>>
SingleNodeConnector
facilitateSingleRequest
handleMusicMatch
handleNewRequest
getNodes
<<component>>
P2PInfoComponent
query
<<component>>
SupernodeDatabaseConnector
query
<<component>>
SuperNodeDatabase
<<component>>
OtherNodesDatabaseConnector
query
update
<<component>>
HeartBeatComponent
<<component>>
P2PInfoComponent
getNodes
handleNeighborhoodInfo
<<component>>
P2PConnector
sendNeighborhoodInfo
reissueRequest
<<component>>
SingleNodeConnector
sendRequest
handleNewRequest
queryMusic
query
<<component>>
FingerprintMatchingComponent
<<component>>
FingerprintDatabaseConnector
updateFingerprints
<<component>>
ServerConnector
getUserAuth
<<component>>
UserComponent
handleNewRequest
attachCertificate
verifyCertificate
<<component>>
CertificationComponent
<<component>>
FingerprintAlgorithmComponent
getAudioStream
<<component>>
AudioInputConnector
getFiles
<<component>>
MusicLibraryScannerComponent
getFile
<<component>>
FileSystemConnector

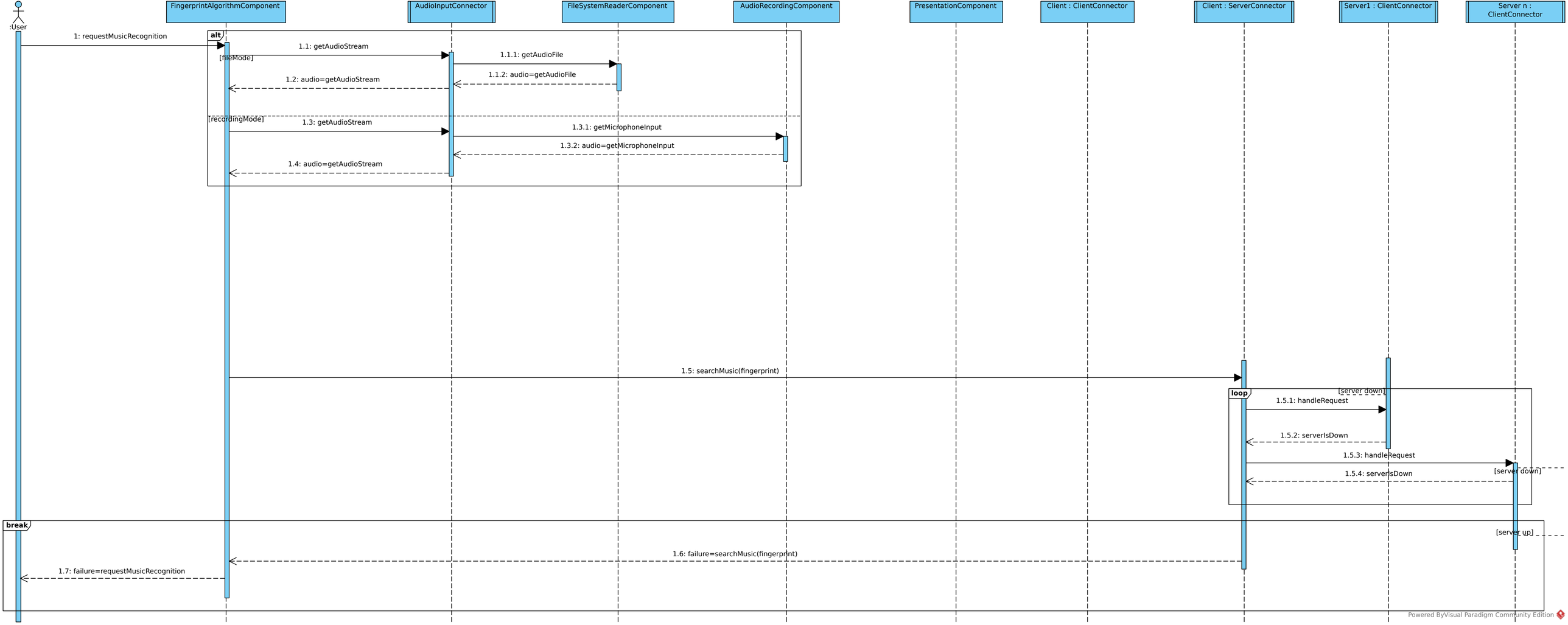

shared
domain-objects
User
has
Coin
Fingerprint
has
Song
has
MusicCollection
node
db
peer-list
fingerprints
p2p-mgmt
library-mgmt
authentication
fingerprint-matching
client
shared
fingerprinting
authentication
mobile
android
ui
recording
ios
ui
recording
server
user-mgmt
authentication
p2p-mgmt
certificate-mgmt
db
users
supernodes




Peer n: HeartBeatMonitor
P2PConnector
HeartBeatComponent
OtherNodesDatabaseConnector
1: heartbeat
1.1: heartbeat
1.1.1: update(heartbeat)
Powered ByVisual Paradigm Community Edition




FingerprintAlgorithmComponent
AudioInputConnector
FileSystemReaderComponent
AudioRecordingComponent
PresentationComponent
Client : ClientConnector
Client : ServerConnector
Server : ClientConnector
UserCurrencyConnector
UserManagementComponent
CertificationComponent
MusicSearchComponent
P2PConnector
P2PInfoComponent
SuperNodeDatabaseConnector
SuperNodeDatabase
SingleNodeConnector
ServerConnector
FingerprintMatchingComponent
CertificationComponent
SingleNodeConnector
FingerprintDatabaseConnector
P2PConnector
P2PInfoComponent
OtherNodesDatabaseConnector
UserComponent
1: requestMusicRecognition
alt
1.1: getAudioStream
1.1.1: getAudioFile
[fileMode]
1.1.2: audio=getAudioFile
1.2: audio=getAudioStream
[recordingMode]
1.3: getAudioStream
1.3.1: getMicrophoneInput
1.3.2: audio=getMicrophoneInput
1.4: audio=getAudioStream
1.5: searchMusic(fingerprint)
1.5.1: handleRequest
1.5.1.1: getInfo(user)
1.5.1.1.1: getSetCoinBalance(user)
1.5.1.1.2: coins=getSetCoinBalance(user)
1.5.1.2: coins=getInfo(user)
1.5.1.3: attachCertificate
1.5.2.4: certificate=attachCertificate
1.5.1.5: searchMusic(fingerprint, certificate)
1.5.1.5.1: queryP2PNetwork(fingerprint, certificate)
1.5.1.5.1.1: getNodes
1.5.1.5.1.1.1: query
1.5.1.5.1.1.1.1: query
1.5.1.5.1.1.2: nodes=query
1.5.1.5.1.2: nodes=query
1.5.1.5.1.2: nodes=getNodes
loop
1.5.1.5.1.3: facilitateSingleRequest(fingerprint, certificate)
1.5.1.5.1.3.1: handleNewRequest(fingerprint, certificate)
1.5.1.5.1.3.1.1: verifyCertificate
1.5.1.5.1.3.1.2: valid=verifyVertificate
1.5.1.5.1.3.1.3: queryMusic
1.5.1.5.1.3.1.3.1: handleNewRequest
loop
[score < threshold]
1.5.1.5.1.3.1.3.1.1: verifyCertificate
1.5.1.5.1.3.1.3.1.2: valid=verifyVertificate
1.5.1.5.1.3.1.3.1.3: queryMusic(fingerprint)
1.5.1.5.1.3.1.3.1.3.1: query(fingerprint)
1.5.1.5.1.3.1.3.1.3.2: score=query(fingerprint)
1.5.1.5.1.3.1.3.1.4: score=queryMusic(fingerprint)
1.5.1.5.1.3.1.3.1.5: response=handleNewRequest
1.5.1.5.1.3.1.3.1.6: reissueRequest
1.5.1.5.1.3.1.3.1.6.1: getNodes
1.5.1.5.1.3.1.3.1.6.1.1: query
break
[score >=threshold]
2: handleNewRequest(peerID)
2.1: getUserAuth(peerID)
2.2: user=getUserAuth(peerID)
2.3: attachCertificate
2.4: certificate=attachCertificate
1.5.1.5.1.3.2: user=handleMusicMatch
1.5.1.5.1.3.3: user=facilitateSingleRequest
1.5.1.5.1.3.3.1: user=queryP2PNetwork
1.5.1.5.1.3.3.1.1: creditAccount(user)
1.5.1.5.1.3.3.1.1.1: verifyCertificate
1.5.1.5.1.3.3.1.1.2: valid=verifyVertificate
1.5.1.5.1.3.3.1.2: searchMusic(fingerprint)
1.5.1.5.1.3.3.1.2.1: debitAccount(user)
1.5.1.5.1.3.3.1.2.2: verifyCertificate
1.5.1.5.1.3.3.1.2.3: valid=verifyVertificate
1.5.1.5.1.3.3.1.2.4: receiveResponse
1.5.1.5.1.3.3.1.2.4.1: handleServerResponse
1.5.1.5.1.3.3.1.2.4.1.1: presentResult
Powered By Visual Paradigm Community Edition


:User
FingerprintAlgorithmComponent
AudioInputConnector
FileSystemReaderComponent
AudioRecordingComponent
PresentationComponent
Client : ClientConnector
Client : ServerConnector
Server1 : ClientConnector
Server n : ClientConnector
1: requestMusicRecognition
alt
[fileMode]
1.1: getAudioStream
1.1.1: getAudioFile
1.1.2: audio=getAudioFile
1.2: audio=getAudioStream
[recordingMode]
1.3: getAudioStream
1.3.1: getMicrophoneInput
1.3.2: audio=getMicrophoneInput
1.4: audio=getAudioStream
1.5: searchMusic(fingerprint)
loop
[server down]
1.5.1: handleRequest
1.5.2: serverIsDown
1.5.3: handleRequest
[server down]
1.5.4: serverIsDown
[server up]
break
1.6: failure=searchMusic(fingerprint)
1.7: failure=requestMusicRecognition
Powered ByVisual Paradigm Community Edition