TangleTunes – Design report

M11 Design Project
Evana Reuvers, s2360012
Daniel Melero, s2358379
Jasper van der Werf, s2615312
Jelte Koorstra, s2570408
Paul Blum, s2534444

Supervisor
Mohammed Elhajj

University of Twente

21st April 2023

Table of contents

1. Introduction 4
1.1. Goals 4
1.2. Scope 4
1.3. Report structure 5
2. Domain analysis 6
2.1. Domain 6
2.2. Target audience 6
2.3. System functionalities 7
2.4. Existing solutions 7
2.5. Terminology 8
3. Requirement specification and analysis 9
3.1. Stakeholders 9
3.2. Functional requirements 9
3.3. Non-functional requirements 11
4. Development methodology 13
4.1. Scrum 13
4.2. Central organization 14
4.3. Planning 14
5. Design Considerations 17
5.1. Incentives 17
5.2. Anonymity 18
5.3. Scalability 18
5.4. Security 19
5.5. Legality 19
6. System design 21
6.1. System architecture 21
6.2. Sequence diagrams 23
6.3. Use Case Diagram 26
6.4. TCP Protocol 28
7. Component Design 29
7.1. Listener 29
7.2. Distributor 32
7.3. Validator 36
7.4. Smart contract 38
8. Product 41
8.1. Listener GUI 41
8.2. Distributor CLI 43
8.3. Validator website 44
8.4. Smart contract 45
8.5. Deployment 46
9. Testing and evaluation 48
9.1. Listener Testing 48

9.2. Distributor Testing 53
9.3. Smart Contract Testing 54
9.4. Validator Testing 56
9.5. System testing 57
9.6. Evaluation 58
10. Discussion 61
10.1. IOTA Shimmer 61
10.2. Flutter and Just_Audio 61
10.3. Denial of service at smart contract 61
10.4. Security concerns 62
10.5. Streaming architecture 62
11. Conclusion 64
11.1. Summary 64
11.2. Achievements 64
11.3. Future work 65
12. References 66
13. Appendices 67
13.1. Project Proposal 67
13.2. Wireframes 70
13.3. Listener User Interface 72
13.4. Validator website UI 78
13.5. Distributor Commands and Configuration 80
13.6. Test results 82
13.7. Contribution log 84

1. Introduction

TangleTunes is a peer-to-peer (P2P) Music Streaming Service (MSS) built on the IOTA distributed ledger, enabling users to upload, distribute, and stream music while ensuring right-holders and distributors receive payment for each chunk of streamed music. It addresses the problem where independent music producers receive insufficient compensation due to their weak negotiating positions compared to larger right-holders and the MSSs [1]. In addition, MSSs can unilaterally change their terms-of-service, leaving independent producers with limited alternatives in a non-competitive market. Consequently, the power balance overwhelmingly favors the MSS.

This project expands upon the initial research and prototype developed by Daniel Melero [2], which demonstrated the feasibility of implementing a music streaming service using the IOTA distributed ledger. The original prototype was purely intended as a proof-of-concept, lacking a user-friendly interface for listening, a reliable method for users to upload songs, and overall system efficiency.

1.1. Goals

Building upon Melero's [2] research, we aim to design and implement a novel P2P MSS on the IOTA distributed ledger, attempting to improve both the design and implementation of Melero's prototype. Our implementation is a comprehensive system built from scratch that offers valuable insights into the feasibility of such a platform within a larger and more realistic environment.

In our project proposal, we outlined four essential goals: One goal for every major system component. These goals form the core of our concrete goals for this project.

1. **Smart contract:** To create and deploy a decentralized music streaming service on the IOTA distributed ledger, enabling users to upload, distribute and stream songs.
2. **Listener:** To create a user-friendly mobile music streaming application.
3. **Distributor:** To create and deploy a music distribution client for the desktop.
4. **Validator:** To create and deploy a validator with a basic web-interface for users to upload songs.

The project proposal is included as Appendix 1: Project Proposal. A more detailed description and analysis of the project requirements is given in chapter 3: Requirements Specification and Analysis.

1.2. Scope

The primary objective of this project is to improve the design and implementation of Daniel Melero's initial prototype [2] across the four focus areas outlined in the *Goals* section. While our aim is not to deliver production-level software, we strive to advance distributed-ledger technology research by pushing its boundaries. Our focus is on enhancing the system's reliability, scalability, speed, and security given our limited time and resources for working on the project. By doing so, we provide valuable insights into which aspects of distributed-ledger technology are mature and which require further development.

1.3. Report structure

Ch. 2	**Domain analysis**	Provides an overview of the domains that TangleTunes is located in, as well as a summary of the terminology used throughout this report.
Ch. 3	**Requirements Specification and Analysis**	Details the functional and non-functional requirements established for the project.
Ch. 4	**Development Methodology**	Describes the methods employed by our team to collaborate and deliver the product.
Ch. 5	**Design Considerations**	Explores the rationale behind the design choices made throughout the project and discusses the trade-offs and constraints considered during the design process.
Ch. 6	**System Design**	Presents an overview of the overall system architecture, including its components and their interactions.
Ch. 7	**Component Design**	Delves into the individual design of each system component, outlining their functions, responsibilities, and relationships with other components.
Ch. 8	**Product**	Provides a comprehensive description of the final product delivered, detailing its features and functionalities.
Ch. 9	**Testing and Evaluation**	Includes a test plan, test results, and an evaluation of the requirements set for the product, as well as an assessment of its performance.
Ch. 10	**Discussion**	Reflects on the project's outcomes, exploring the implications of our findings, the limitations of our work, and potential areas for further investigation.
Ch. 11	**Conclusion**	Summarizes our project, highlighting lessons learned and recommendations for future work and research.

Table 1.3.1.

2. Domain analysis

For this project a domain analysis has been conducted. This analysis is intended to give a better understanding of the current domain of the TangleTunes system, its users and its tasks. Additionally, the analysis establishes the existing systems in these domains and their relevance is to our project.

2.1. Domain

TangleTunes is a decentralized music streaming application, intended to give users the ability to upload, stream and distribute music on a peer-to-peer basis. Because its core functionality is to stream music, it falls under the domain of music streaming applications. However, in contrast to traditional streaming services, TangleTunes has additional subdomains that differentiate our application.

TangleTunes makes use of distributed ledger technology; this provides a governance structure where users determine the terms of service of the application. Additionally, our platform differentiates itself by offering pay-per-play payments instead of requiring a subscription. Lastly, the usage of p2p streaming contrasts the standard way of using centralized servers. Hence, the subdomains are distributed ledger technology, pay-per-play payments and p2p streaming.

2.2. Target audience

The main function of our system is to allow users to stream music uploaded by right-holders while fairly compensating the right-holders. This functionality concerns our three core user-groups: The right-holders, the listeners and the distributors. Alongside these core users, we need an amount of trusted validators and node operators to successfully run the platform.

Core audience

The right-holders are users who own intellectual rights to music, in general these are independent musicians or record labels. Right-holders primarily want to be compensated when people listen to their music. Right-holders are essential by uploading music to the platform; without right-holders, no music is uploaded and no one can listen to music.

The listeners are regular users who wish to listen to music through an easy-to-use mobile interface. They may be concerned with fair compensation of artists, but many have not thought about this. Listeners are essential because without them, no one pays for the music and right-holders and distributors don’t receive payment.

The distributors are users who distribute the uploaded music, while receiving payments through their distribution fee. They are primarily concerned with receiving compensation for their work, however distributors may also work to help their favorite artists spread their music. These distributors are essential, because without them, music on the platform is unavailable for users to listen to.

Additional audience

The validators are users who validate music for intellectual rights before it is uploaded by the right-holders. The validators must be incentivized to protect intellectual rights in order for the platform to operate in a legal manner.

The node operators are users who have a Wasp-node that runs the TangleTunes smart contract. The node operators must be incentivized to run an honest node and protect the integrity of the platform together.

2.3. System functionalities

As mentioned in the introduction, our system implementation consists of four components, each with their own unique set of functionalities. These four components exist in relation to the user-groups.

The first component is the listener mobile application. Its main function is to allow the listener to stream music from other distributors through a simple and attractive user interface. The second component is the distributor CLI. It allows distributors to distribute music on the TangleTunes network all over the globe. The third component is the validator website. Validators can use it to set up a website that allows them to easily validate and upload music requested by right-holders. The final component is the smart contract and it is used by the node operators to run their wasp-nodes.

Together, deployment and usage of all components results in a comprehensive system that can be used as described above in the target-audience paragraph.

2.4. Existing solutions

There exist quite a few projects that attempt to solve the music industry’s problems by using distributed ledger technology. All of these proposed solutions are still in early development, and only a few of them have a working alpha version at the time of writing this report. Even though these projects do not appear in relevant academic literature, most of them have an official whitepaper, as is custom in projects involving cryptocurrencies. In this section, we will broadly compare the most relevant solutions to our project.

First, the most successful platform, at least in terms of media coverage, is called Musicoin [3]. This project is built over the SKALE distributed ledger, which allows for the creation of private and feeless chains, secured by a Proof of Stake consensus mechanism. Similarly to our project, musicians are allowed to upload their music and set a price for each song. Listeners can enjoy the platform’s content on a pay-per-play basis and can tip their favorite artist if they wish to. Their proposed solution is to host the encrypted files on IPFS, a decentralized data storage network, and then upload the song’s metadata on a smart contract that will manage the relevant payments. A listener can download the file, pay for the given song and then decrypt the file using a key received in exchange.

There also exists a solution proposed by eMusic [4], a veteran company in the music industry with more than 20 years experience in the field of online music distribution. Their project is built on the Ethereum blockchain. The distribution of music on their platform is comparable to any standard centralized music streaming service like Spotify, but the use of distributed ledger technology allows the platform to be highly transparent about their finances with artists as well as with audiences. Right-holders upload their music through eMusic's website, files are hosted on their cloud infrastructure and metadata about the deal is recorded using a smart contract. Listeners can listen to the music through their eMusic's applications. A record of requested songs is kept on a second smart contract. After a fixed amount of time has passed, eMusic pays right-holders based on users' listening-time and the deal struck with the artist.

Even though many projects have attempted to solve the music industry's problems using distributed solutions, no proper, successful implementation exists at the time of writing. Additionally, the existing solutions tackle the problem in a different manner than we do. Musicoin is completely decentralized and therefore has problems regarding intellectual rights, while eMusic has them into consideration but does not take full advantage of distributed ledger technology. We think it is worth our time to design and implement a novel solution that balances both of these aspects.

2.5. Terminology

The following list includes the most important terms relevant to understanding the report. It includes general technological terms, abbreviations and terms unique to our platform.

- **MSS:** Music Streaming Service.
- **P2P:** Peer-to-peer; a distributed architecture that divides tasks or workloads among and between peers directly instead of through a central system.
- **IOTA:** An open, feeless and scalable data and value transfer protocol built on distributed ledger technology.
- **Distributed ledger technology:** Technologies which agree on a central ledger state using distributed consensus algorithms.
- **L1 ledger / The Tangle:** The primary distributed ledger that IOTA uses**.**
- **L2 ledger:** A distributed ledger that uses the security of the L1-ledger. In our case this uses IOTA Shimmer technology.
- **EVM:** Ethereum Virtual Machine.
- **Gas fee**: The fee that users have to pay to node operators per transaction
- **Smart contract:** A program on the L2-ledger that executes events in a deterministic manner. This program is spread across multiple nodes who run the program.
- **Chunk:** Up to 32500 bytes of an mp3-file.
- **Distributor fee**: The fee that a listener has to pay to a distributor per chunk.

3. Requirement specification and analysis

Before implementation started, we specified a list of all requirements and stakeholders. This was done to specify the needs of our project in detail. By defining the exact requirements we can determine the most important requirements for each of our stakeholders. In this definition, we include both the functional and non-functional requirements. The functional requirements define what we wish to achieve, while the non-functional requirements describe quality attributes of these goals.

3.1. Stakeholders

The following stakeholders describe all parties with an interest in our system. To be as complete as possible, we distinguished between direct and indirect stakeholders. Direct stakeholders are involved with the system on a daily basis, whereas indirect stakeholders take a higher interest in the outcome of the project. Based on these definitions we constructed the list of stakeholders as mentioned below.

Direct stakeholders

- Listener: A user who listens to music by streaming it from a distributor.
- Distributor: A user who distributes music to listeners.
- Right-holder: A user who uploaded a song to which they have the intellectual rights.
- Validator: A specially-authorized user that can upload songs for right-holders.
- Node operator: A user who runs a wasp-node with the smart contract code.
- Contract deployer: The deployer of the smart contract.

Indirect stakeholders

- Copyright legislators
- Competitors
- The IOTA foundation
- Developers

3.2. Functional requirements

Functional requirements are the requirements related to the features that need to be implemented in order for the application to fulfill the user's needs. Once the stakeholders of our project had been determined, we started writing our functional requirements based on each specific stakeholder. In order to do this, we used use cases. Use cases are written descriptions of how the user will perform a certain task in the application. Giving a general idea of the system's behavior. In our case, the user was the intended stakeholder. By doing this we could determine a list of functional requirements per stakeholder. The following functional requirements are grouped according to stakeholder, and are ranked according to the MoSCoW requirements specification: (M) must-have, (S) should-have, (C) could-have, (W) won't-have.

Listener

- A listener should be able to listen to music by paying per chunk. (M)
- A listener should be able to play and pause a song. (M)
- A listener should be able to select a song to play by id. (M)
- A listener should be able to see their funds. (M)
- A listener should be able to see the metadata (price, duration) of a song. (M)
- A listener should be able to generate a key pair. (M)
- A listener should be able to seek within a song. (M)
- A listener should be able to set a password for decryption of their private key. (S)
- A listener should be able to enter their password upon opening the app. (S)
- A listener should be able to withdraw funds. (S)
- A listener should be able to view his public key. (S)
- A listener should be able to export his private key. (S)
- A listener should be able to import a private key. (S)
- A listener should be able to search the index of all songs by artist and song-name. (S)
- A listener should be able to add songs to their library. (S)
- A listener should be able to remove songs from their library. (S)
- A listener should be able to search in their library. (S)
- A listener should be able to listen to music when leaving the application. (C)
- A listener should be able to set a maximum price/chunk. (C)
- A listener should be able to add songs to the queue. (C)
- A listener should be able to remove songs from the queue.(C)
- A listener should be able to view their queue. (C)
- A listener should be able to select a listening-strategy. (W)
- A listener should be able to tip a right-holder. (W)

Right-holder

- A right-holder should be able to get paid the rights-fee for chunks streamed of their music. (M)
- A right-holder should be able to request registration at a validator for their music. (M)
- A right-holder should be able to change the rights-fee of a song. (S)
- A right-holder should be able to change the distribution-fee of a song. (S)
- A right-holder should be able to deregister their songs. (S)
- A right-holder should be able to lock their distribution-fee of a song for a set time. (C)
- A right-holder should be able to lock their rights-fee of a song for a set time. (C)

Distributor

- A distributor should be able to distribute any registered music if they have the music stored locally. (M)
- A distributor should be able to set the distribution fee for a song. (M)
- A distributor should be able to generate a key pair. (M)
- A distributor should be able to distribute music anonymously. (S)
- A distributor should be able to stop distributing music. (S)

User (Listener, Right-holder and Distributor)

- ❖ A user should be able to create an account anonymously. (M)
- ❖ A user should be able to deposit funds to their account. (M)
- ❖ A user should be able to withdraw funds from their account. (M)

Node operator

- ❖ A node-operator should be able to run the smart contract. (M)

Contract deployer

- ❖ A contract deployer should be able to deploy the smart contract. (M)
- ❖ A contract deployer should be able to authorize validators. (S)

Validator

- ❖ A validator should be able to register songs for any user. (M)
- ❖ A validator should be able to set the rights-fee upon registration of a song. (M)
- ❖ A validator should be able to deregister a song they validated. (S)
- ❖ A validator should be able to set zones for which a song is valid. (C)

3.3. Non-functional requirements

Non-functional requirements are different from functional requirements. They are not intended to describe the functionalities of a system, but to describe the operating quality of the system. These are the requirements related to the performance, scalability, or portability of the system for example. Besides these categories, we included other quality categories as well. All of these include requirements that are relevant to achieve for the proper, and user-friendly, functioning of our system.

Performance

- ❖ Selecting a song should start playing audio within 0.5 seconds.
- ❖ The creation of an account should take no longer than 1 minute.
- ❖ The system should be able to register and deregister a distributor within 5 seconds.
- ❖ Skipping around in a song should start playing within 1 second.

Scalability

- The system should be able to scale to 1000 active listeners with 50 distributors.

Portability

- Listening should be possible on mobile devices. (Android/IOS)
- Distributing should be possible on a desktop. (Linux/Windows/Apple)
- Uploading a song should be possible through a website.

Reliability

- Selecting a song should choose a distributor with a good connection 95% of the time.

Availability

- The system should be available whenever and wherever the IOTA ledger is available.

Compatibility

- The distributor application's database should be compatible with new updates of the application.

Maintainability

- The listener client application should be able to receive updates.
- The distributor client application should be able to receive updates.

Security

- The smart contract should only store IP's, usernames, descriptions and wallet addresses of distributors.
- The smart contract should only store usernames, descriptions and wallet addresses of listeners.
- The system should allow for listeners to encrypt their streamed music.
- The system should ensure that the song-data received by the listener is verified to be the same as that on the smart contract.

Localization

- The system should support English.
- The system should display balance in any of the 10 most common currencies.

Usability

- Listening and registering should not require a port to be opened.
- Distributing should be possible after opening a port.
- Withdrawing money should be a couple of clicks and typing in the deposit-address.

4. Development methodology

This chapter gives a thorough explanation of our development methodology. Our overarching methodology has been to use scrum; our process using scrum will be described first. After this description, an explanation of our central organization is given. This explains which technologies we used to aid our development. Finally, our project implementation timeline is shown and explained.

4.1. Scrum

For this project we have chosen to use the Agile approach to software development; in particular, we used the Scrum method. This allowed us to work in an iterative manner and gave us the opportunity to do quality checks, incorporate challenges into our design as they occur and work together closely with our supervisor and client.

The project was split up in sprints, where each sprint has its unique purpose. A sprint lasted one week and was guided by a sprint leader. The sprint leader is a member of our team, who was responsible for keeping track of our progress and had to make sure that everyone was participating and knew their role. Every second sprint a new leader was selected such that everyone was leader for two weeks.

Our group was divided into four teams that primarily worked on different parts of the system. Jelte and Paul focused on the listener client, Jasper focused on the distributor, Daniel focused on the smart contract and validator and Evana focused on the User Interfaces. We met up daily, either physically at the University or online. During these daily meetings we would discuss our challenges and progress, work together on solving problems and integrate our code. Everyone worked together, with good collaboration between team members.

To aid the division of tasks and to help keep track of progress, we used a Trello board which has cards for all tasks. These cards were created from the requirements of our system and implementation details. Hence, most cards were redivided into three categories: could have, should have, and must have. At the beginning of every sprint, we chose which requirements we wanted to fulfill that sprint. The cards belonging to those requirements were dragged to the ‘this sprint’ column. On completion during the sprint, they would be moved to the ‘done this sprint’ column. After the sprint was over, all cards would be moved out of the ‘done this sprint’ column and moved to the ‘done’ column. Furthermore, the process of selecting requirements for the sprint would restart at the beginning of every new sprint.

Moreover, we had weekly meetings with our supervisor/client Mohammed Elhajj. In these meetings, we showed and discussed the progress of the last sprint and would state our goals and possible problems for the coming sprint. It also gave our client the opportunity to provide us with feedback, which we could incorporate into the next sprint.

4.2. Central organization

For the central organization of our project we used a variety of tools. As mentioned above, we used a Trello board to keep track of our progress. Moreover, to organize and store our code, we used GitHub. This allowed easy version control, collaboration with another on the code and a way to open source the code. Our Github organization is TangleTunes and contains four repositories. These repositories are all related to the different parts of our project. The distributing_client repository contains all code for the distributor CLI. The smart_contract repository contains the smart contract definition and documentation. The listener repository contains the code for the mobile application and the validator repository contains code to launch the validator website.

Another tool we used was Google Drive. We wrote all of the documents in Google Docs, which allowed us to work together on the same files. Our drive is organized with folders that have representative names; there are folders with our final documents, presentations for peer review meetings and supervisor meetings, and our design documents for example. We also kept track of our progress in a contribution log that was shared with the supervisor.

(See Appendix 7 : Contribution log)

For communication between team members we chose to use Whatsapp and Discord. Whatsapp is used to determine where we hold our meetings and at what time, or to discuss important deadlines for example. Whereas Discord is used for questions related to the code. During online working days we communicate on Discord and have meetings in voice channels. We made this distinction between Whatsapp and Discord to ensure that we would not miss any important announcements in between the talk about our code, in this way we could work more organized and smoothly.

4.3. Planning

At the beginning of our project we created a project proposal that included a timeframe. This timeframe was intended to set clear deadlines for every stage of the project and to ensure that we knew what would need to be finished per stage. Once the project had started, we followed this planning and aimed to achieve all the tasks that were written down for every stage. Although the planning was very useful at the beginning, we realized during the project that a few changes were necessary for the project to succeed.

As mentioned before, we used the Scrum method for our software development. A part of this was that we determined per sprints what our goals were and what we would devote our time to during that week. Our original time frame was a good guideline for this and based on that we determined what to do in the sprints. Below we will give a more detailed description of the actual course of our project. The original time frame can be found in Appendix 1: Project Proposal.

(See Appendix 1: Project proposal)

Time frame based on sprints

Sprints	Tasks	Dates
1: *Exploration*	- Getting to know the team - Read up on relevant materials for this project - Defining the scope of the project - Defining requirements	Week 1: *6 February - 12 February*
2: *Project proposal*	- Write project proposal - Setting up coding environments	Week 2: *13 February - 19 February*
3: *Design*	- Create UML diagrams - Create design in wireframes - Started on basic implementation of code	Week 3: *20 February - 26 February*
4: *Vacation*	Vacation	Week 4: *27 February - 5 March*
5: *Basic implementation*	- TCP protocol established - Implementation of smart contract functionalities for listener - Distributors can now download a song from other distributors. - Validator can validate uploaded songs. - UI for start up screens, verification screens and discovery screen completed	Week 5: *6 March - 12 March*
6: *Extended implementation*	- Distributor was deployed to Raspberry Pi - Tangle and chain node deployed on Raspberry Pi to allow interaction with smart contract - Deployed validator web application on Raspberry Pi - Listener can fetch songs from distributor	Week 6: *13 March - 19 March*

7: *MVP*	- Listener can play songs that were uploaded and distributed - The songs automatically distribute themselves throughout the network - Rightholder can upload their music and validator can validate it and distribute it	Week 7: *20 March - 26 March*
8: *Finalizing and testing*	- All UI pages are completed. - Finishing touches - Testing of the components	Week 8: *27 March - 2 April*
9: *Reflection*	- Worked on the reflection component group report	Week 9: *3 April - 9 April*
10: *Report*	- Finished the project report - Finish the presentation	Week 10: *10 April - 16 April*
11: *Presentation*	- Have the final presentations	Week 11: *7 April - 23 April*

Table 4.3.1. Planning in sprints

5. Design Considerations

This chapter describes some essential considerations we keep in mind for all designs. These concerns are the proper incentivization of all users, providing an anonymous experience wherever possible and making a platform that can be operated legally.

5.1. Incentives

For the platform to work correctly, all actors must be incentivized properly to participate in the network. This includes the listeners, distributors, validators, right-holders and node-operators. For each of these parties the incentives for participation in the network are given below.

Right-holder

The main motivation for the creation of our platform was to improve the position of the right-holder/music producer in relation to MSS's. MSS's like Spotify and Apple Music do not allow the right-holder to set their own prices for the music. There are negotiations between the right-holder and MSS, and the bigger the right-holder is, the better the contract can become. Our platform incentives especially the smaller, independent music producers to put their music on our platform, since they can ask higher and fairer prices for their music. Therefore, right-holders should have the clearest participation-incentive out of all users.

Listener

The primary incentive of a listener to use our platform is that we provide a platform with music that does not require any kind of subscription. There is no vendor lock-in and thus a very low barrier of entry. If a song is only available on TangleTunes, people can simply use it for streaming only that song and continue using their other MSS's. This means users can enjoy flexibility in using multiple music streaming services without exclusive commitment to one. In addition, we provide a platform where music-producers are properly rewarded for their music. Many users may not consider this ethical aspect, but for the users who do, this is a good incentive.

Distributor

A distributor is incentivized by receiving payments for the distribution of music. We create a competitive market between distributors to reduce costs and improve latency, reliability and offering. In this way, the market will be automatically regulated: If profits are high, more distributors will join, increasing competition and lowering prices. If profits are low or negative, prices of distribution will automatically go up until distributors are profitable again.

Validator

A validator can be incentivized by accepting payments for validating and uploading a song for users. We aim to create a market where multiple, carefully selected validators compete for the best service at the lowest price. Validators set up their own payment methods and may offer additional services besides just validation of music, for example seeding a song into the network for a given duration.

Node-operator

Node-operators should be parties with an incentive to keep the platform running in a stable way. Currently we operate the nodes ourselves, but these nodes should in the future be operated by the biggest right-holders and distributors. These parties have an incentive to keep the platform operating well and can start operating nodes. There is a cost involved in running these nodes, but the value returned is that the node-operator can be more confident in a service that is both secure and fast.

5.2. Anonymity

There is a clear distinction in the system between the centralized and non-anonymous part (uploading) and the decentralized, anonymous part (listening). If a user wishes to upload a song or become a validator they must authenticate themselves before doing so. This is by design; someone must be legally responsible for the music registered on the platform. (*See 5.5. Legality*)

We aim to provide anonymity mainly for the distributor and the listener as far as this is possible. The listener does not register any personal information on the smart contract, except for his IOTA address, an optional username and description. The distributor has to register his IP-address in addition to this information on the smart contract. The registration of your personal IP-address definitely sacrifices anonymity, however can be circumvented by using a VPN.

When the listener contacts a distributor to set up a TCP-connection or the smart contract over HTTP, they expose their IP-address to the distributor. This can again be circumvented by using a VPN. In addition, the entire internet functions on this basis, so is considered out of scope for our project.

5.3. Scalability

Throughout all designs, we consider scalability from a first-principles approach. Streaming itself scales efficiently by having more distributors join the platform, allowing them to distribute music concurrently. The primary bottleneck we identify in the system is the smart contract itself. For this reason, our primary focus regarding scalability is on optimizing the smart contract for better performance and efficiency.

Enhancing the scalability of the smart contract can be achieved through multiple strategies. First, we aim to use minimal and efficient data structures to reduce gas costs and accelerate transaction processing. By streamlining data structures, we can maintain a high level of performance even as the platform grows.

Secondly, we aim to batch transactions together, such as paying for multiple chunks in a single transaction or registering for the distribution of multiple songs simultaneously. Batching transactions can significantly improve the platform's scalability by reducing the number of individual transactions that need to be processed.

Lastly, we strive to design the system in such a way that as many transactions as possible are pure calls. Pure calls do not alter smart contract information and only need to pass through a single node. As a result, they are considerably more efficient and scalable than transactions that modify the smart contract state.

By incorporating these strategies, we hope to improve the scalability of TangleTunes, allowing it to grow and adapt to an increasing number of users and transactions without compromising performance or efficiency.

5.4. Security

Throughout the entire project, we focus on the security of the system wherever necessary. Security is especially important whenever the private key of a wallet is involved. The private key may never be stored in persistent storage unencrypted. Applications in our project that store such a private key are Metamask for the validator website, the mobile application and the distributor-client. In addition to storage, the private key may never be shared with any other users, importing and exporting it in plaintext is also not an ideal solution.

5.5. Legality

Disclaimer*: This section was not written by a lawyer or anyone with expertise on intellectual-property law. It is based on observation of similar products and limited understanding of the laws.*

Many design-decisions have been made keeping the legality of our system in mind. The overall design was guided by the idea to create a distributed, anonymous but legal system. This section describes the decisions we made from a legal point of view.

Intellectual rights

Our primary concern with intellectual rights is that someone *has* to be responsible for the music on the platform. If none of our users can be held responsible because of anonymity, then the responsible entity might be the developers or contract deployers.

Therefore a primary design-pillar was a way to make right-holders responsible without creating a single point-of-failure. In our system, there can be entities competing to become validators, and they become legally responsible for any content that is validated. In this way the legal responsibility moves from us to the validator. The validators can then set up their own policies and legal contracts with the right-holders to move responsibility from them to the right-holder. This makes it so that *we* do not have to be experts on intellectual property law, but can have others compete to do this for us.

Streaming vs Downloading

When a user streams a song, the obvious question is: What rights does he buy when he pays for it? The user is not allowed to store the song in any way after playback, but the user must be able to store the song on their device for a certain amount of time, otherwise buffering becomes impossible. We propose to set a limited time, e.g. 5 minutes, for which a chunk may be stored on the device. When the time is over, the user may not play the song without paying for it again.

There is a difference when the distributor streams a song from another distributor though: Here the user needs to store the song on their device, until the distributor stops the distribution. Therefore, a user may download a song and keep it stored indefinitely, however they are not allowed to play or share it with any entity without a payment through the smart-contract.

6. System design

In this chapter we explain top-down what the overall system architecture looks like, providing information on how all the different components function together. This is exemplified by sequence diagrams of important processes, a use-case diagram and an exact specification of the TCP-protocol between listener and distributor.

6.1. System architecture

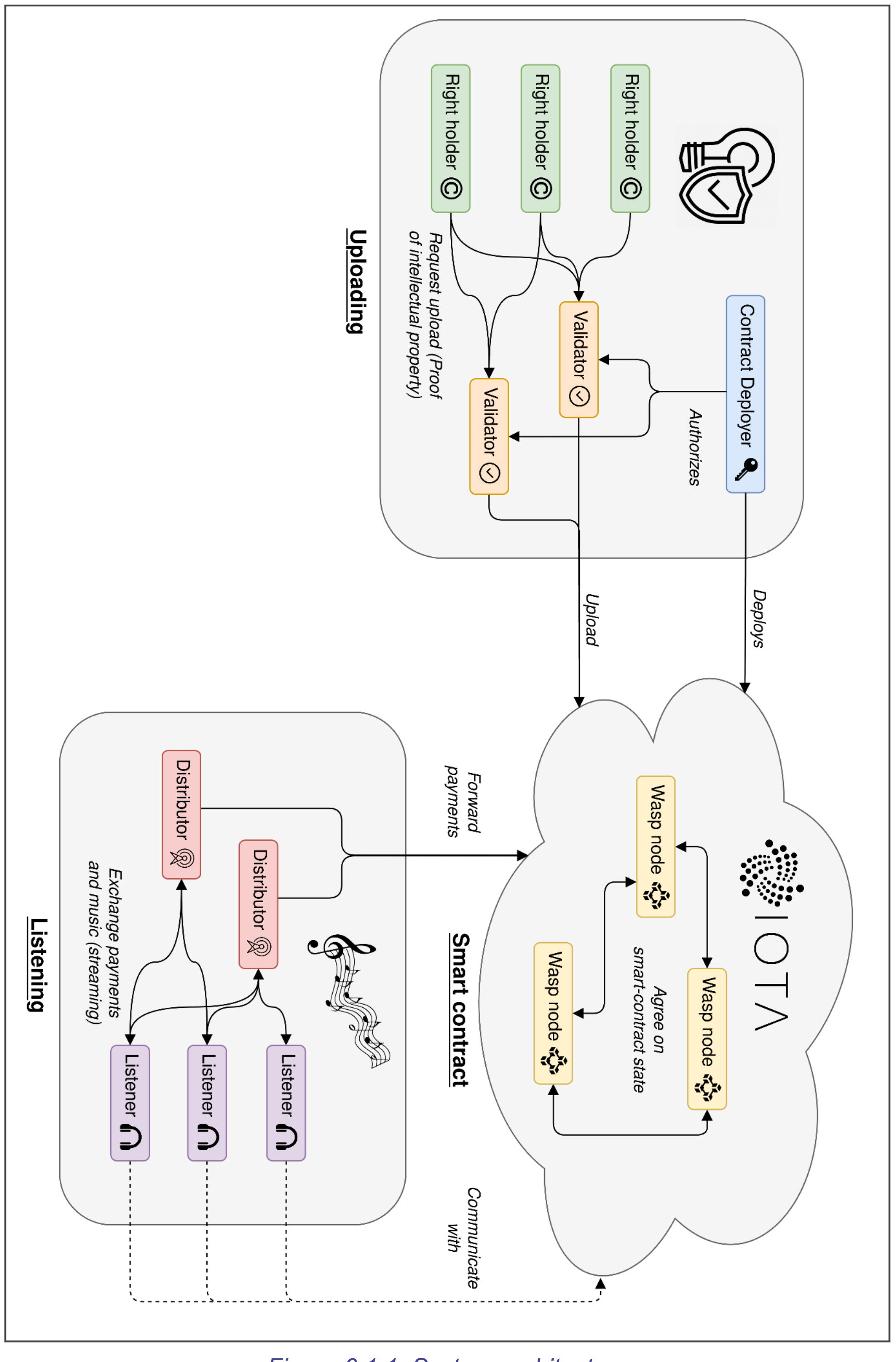


Figure 6.1.1. System architecture

The diagram above describes the most important aspects of our system from a high-level. It displays all direct actors and the most important functions each is responsible for. We can identify three main processes within the system: Validation, listening and the smart contract. Validation concerns uploading and validation of music through a centralized process. Listening is where listeners contact the distributors and music is exchanged for payments. Both the processes of validation and the listening interact with the smart contract, but they never interact directly with another.

Smart contract

The smart contract is the mediator and contains all essential logic and state. It does this in a decentralized manner where the wasp-nodes communicate together to arrive at a consistent state. In our implementation we run these nodes ourselves, but in the long run these nodes should be run by organizations with an incentive in keeping the system functioning well. The smart contract is deployed on a (custom) L2-chain on the Tangle.

The smart-contract stores some essential information of all songs uploaded. It stores their chunk-hashes together with the validator, author, distributor and price. This allows anyone to verify that a song is uploaded and that the data received is correct. When someone registers for distribution, their ip-address and distribution fee is stored on the smart contract. Additionally, the smart contract deals with all payments made by streaming music by storing the balance of all users on the contract and automatically forwarding payments from listener to distributor and right-holder.

Uploading

There is a single contract-deployer with the unique authorization to mark users as validators. When a user is marked as a validator, they are authorized to upload music for other users. The right-holder creates a tamper-proof request for a song to be uploaded and the validator then signs this request and forwards it to the smart contract. By uploading a song for another user, the validator is responsible that the right-holder owns the intellectual rights to the music uploaded. If this is not the case then legal action can be taken against the validator.

Validators are free to implement their own custom platform where users can request an upload by providing proof of intellectual rights. We provide a sample implementation registered at tangletunes.com which implements manual verification through email.

Listening

Once a song is uploaded by a validator, it can be distributed by any user. When the user registers for distribution of a song they automatically become a distributor and can be contacted by a listener to request the song. The distributor must have access to the entire mp3-file before they can start distributing. The song can be acquired either privately, or by downloading it from another distributor if one exists. Once the song has been acquired the data can be verified by comparing chunk-hashes to the hashes stored on the smart contract. Our sample-validator automatically seeds the song into the network by running a single distributor that distributes all uploaded songs; other distributors can join by downloading the song here.

If a song is registered and has at least one distributor, a user can stream the song given that they have enough balance to pay for it. The user does this by asking the smart-contract for a distributor's ip-address and then sending payments for the given song to the distributor over TCP. The listener can pay per chunk of a song. These payments are split according to the "price per chunk" and "fee per chunk" between right-holder and distributor.

6.2. Sequence diagrams

The following sequence diagrams give overviews of the most important functionalities of the system where different components interact with another. The diagrams describe how the different components of the system interact with another. For every functionality there is a diagram with a description explaining the diagram.

Streaming a song

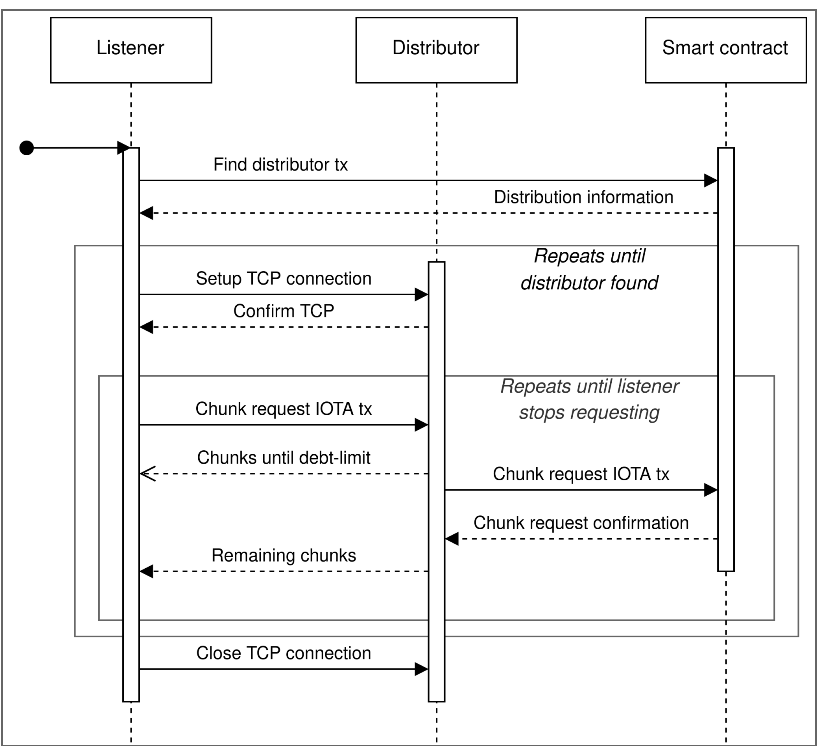


Figure 6.2.1. Sequence diagram - Streaming a song

Describes the process of a listener trying to stream a song. It is assumed that the listener already has an account with funds, the song has already been uploaded and there is at least one distributor for the song. The listener starts by finding a distributor to contact. It does this through the smart contract, and can either get a random one or select one from a list. The listener then sets up a TCP-connection and streaming can now begin.

The listener creates and signs payment-transactions locally, and sends them to the distributor over the TCP-connection (*See* *6.4. TCP-protocol*). The distributor then sends chunks until the debt-limit has been reached and forwards the payment-transactions to the smart contract. Once the transactions have been confirmed, the distributor sends back the remaining chunks of the request. This repeats until the listener closes their TCP-connection with the distributor.

Uploading a song

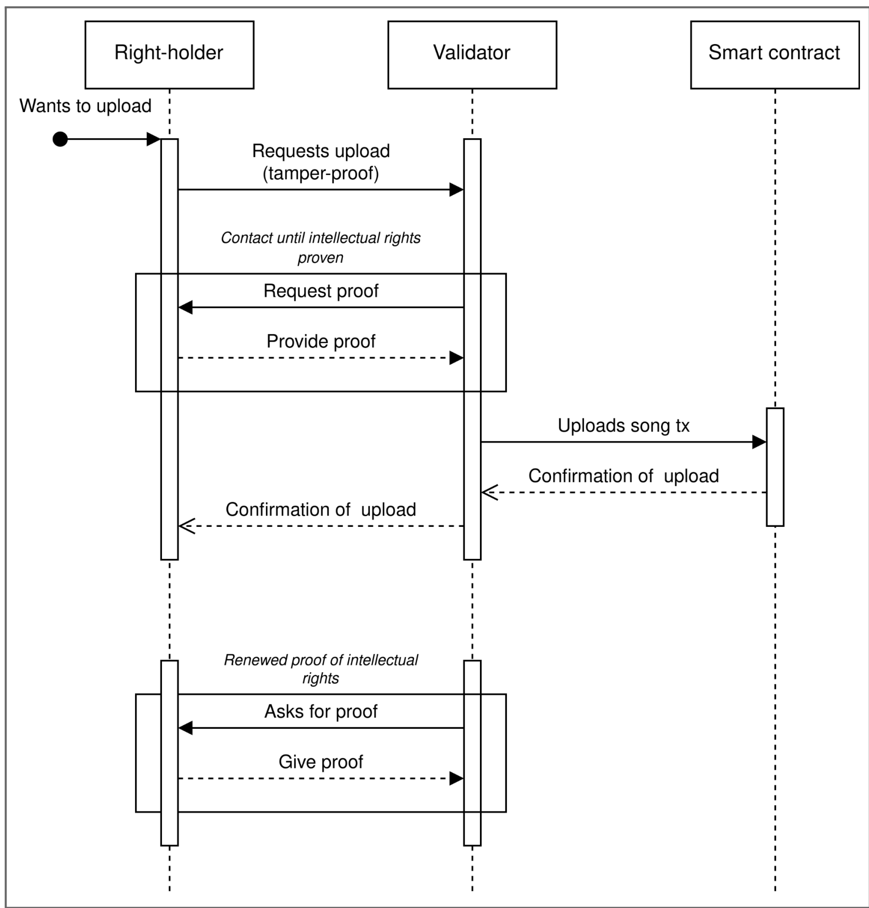


Figure 6.2.2. Sequence diagram - Uploading a song

Uploading a song is done by requesting an upload privately through a validator with a tamper-proof request. The validator may ask money for this service through their own methods. The validator must ask for proof of ownership of the account and proof of intellectual property of the song. If the validator is confident in this, it uploads the song for the user. After the initial validation, the validator can repeatedly contact the owner to ask for proof again. If proof cannot be given the validator may remove the song from the platform.

Registering for distribution

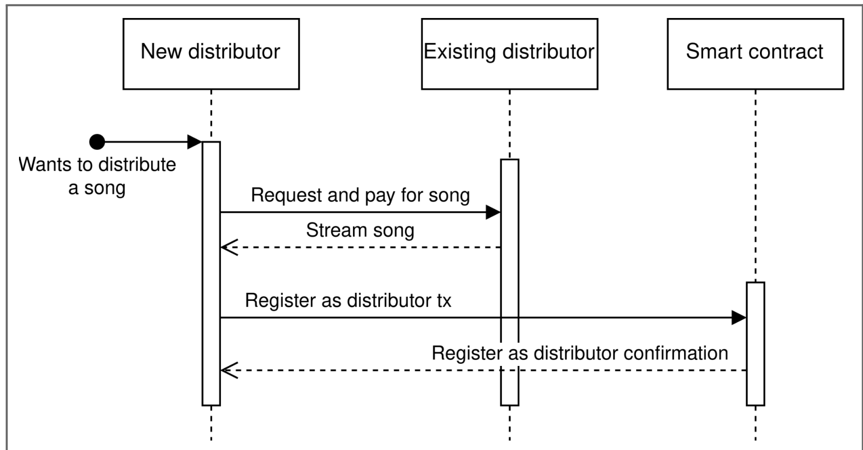


Figure 6.2.3. Sequence diagram - Registering for distribution

This describes the default process of a new distributor joining the network and registering for distribution of a song already in the network. The distributor starts by streaming the song from an existing distributor, paying for it the same way a listener would. Once the song is fully downloaded, the distributor can register itself for distribution on the smart contract.

Creating an account

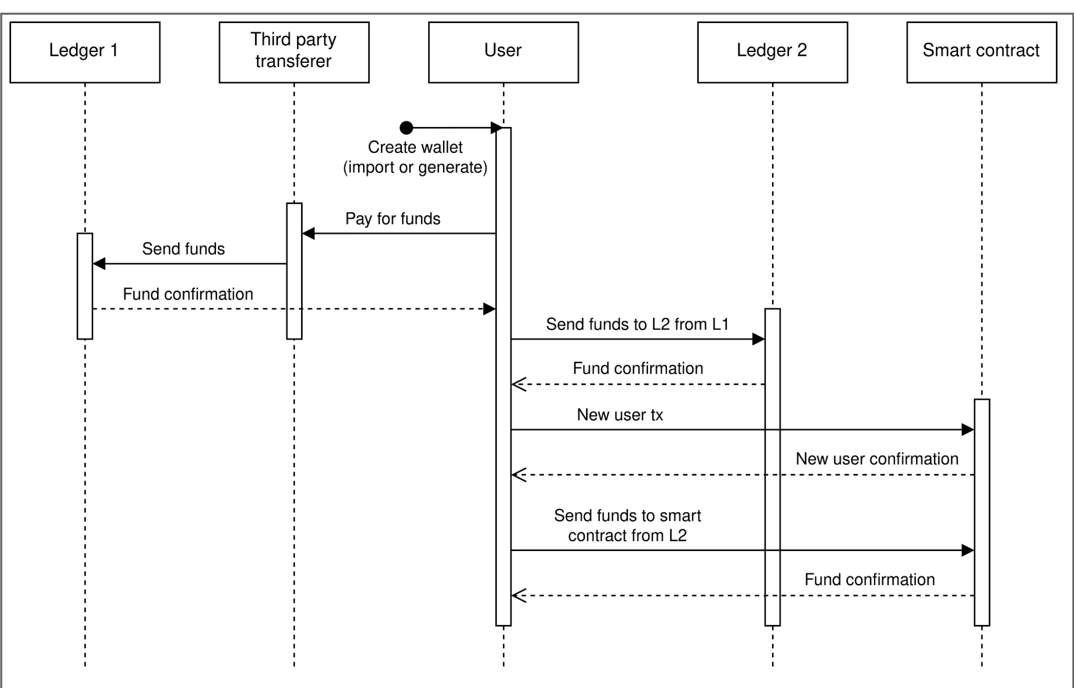


Figure 6.2.4. Sequence diagram - Creating an account

This diagram describes the process a new user goes through when creating an account and sending funds to the account. This process is used both in the listener as well as the distributor. The user will first need to generate or import a private key that is his IOTA wallet. Once they have a wallet, funds can be sent to it by using a third party website. This will send

the funds to the L1-wallet, from there the funds have to be sent to the L2-wallet. When the user has funds in his L2-wallet, it is now possible to create an account through the smart-contract. Only when the user has an L2-account can funds be sent to the smart contract account.

6.3. Use Case Diagram

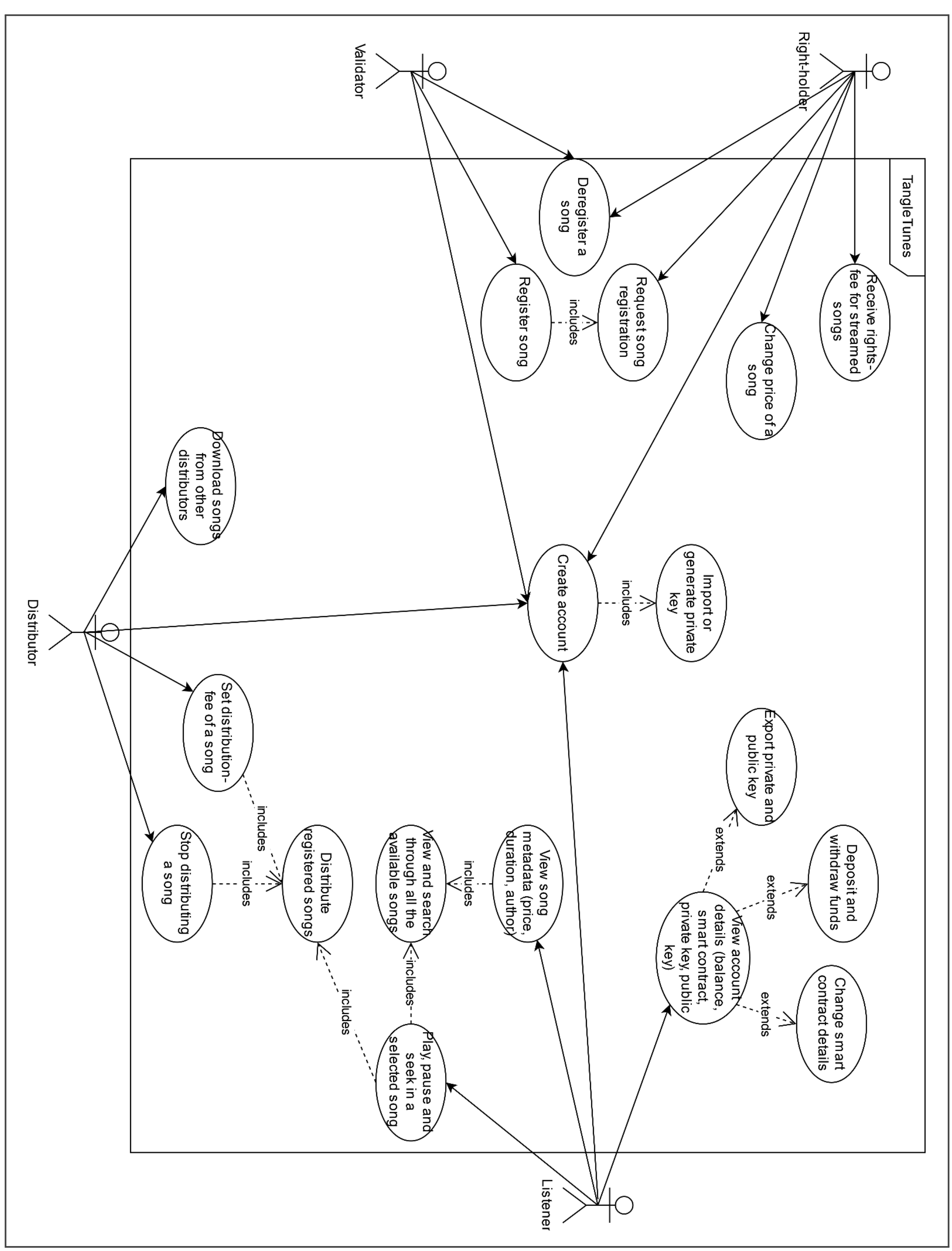


Figure 6.3.1. Use case diagram of entire system

The use case diagram contains the requirements for the right-holder, validator, distributor and listener and their relations. The right-holder can upload a song to the platform by requesting a song registration. The right-holder is able to change the fees for this song, and receive payments for this song once it is registered. The validator can validate requests of right-holders, and is able to change the rights-fee of a song. Both the right-holder and validator have the ability to deregister a song from the platform.

Once the song is on the platform, the distributor can download the song and then distribute it to the listeners. The distributor needs to have an account on the smart contract in order to start distributing. Distributors can set their distribution fee, and they can stop distributing at any time. Moreover, distributors can also download songs from other distributors so that they can distribute those songs themselves.

The listener is able to play, pause and see into registered songs on the platform that are being distributed. Listeners can get an overview of all the available songs on the platform on the discovery page, when they select a song they can view the metadata of the song which includes the price and duration among other things. Additionally, the listener can perform account tasks such as depositing, withdrawing, changing smart contract contact details and exporting the private and public key. All the 4 types of users are able to create an account by generating or importing a keypair.

6.4. TCP Protocol

The exchange of music between distributor and listener uses a custom TCP-protocol defined below. The listener sends chunk-requests to the distributor. These chunk-requests are the raw rlp-encoded bytes of the IOTA smart contract transaction. The parameters of this request can then be decoded at the distributor to retrieve which song and chunks have been requested.

The chunk-reply contains the chunks of the mp3-file that should have been requested. The requests may be sent asynchronously, but must be in the order that was requested. A distributor may split up a request into multiple replies smaller than the original request, but may not merge requests into a single reply.

Once the distributor receives a transaction, he will send the transaction to the smart-contract, checks that the distributor-address is his own and awaits the confirmation. Once this confirmation is received, the distributor must send all requested chunks back to the listener. The distributor may send a number of chunks before receiving a confirmation, until the so-called debt-limit. If a transaction is invalidated for whatever reason, the TCP-connection may be closed by the distributor.

Chunk-request

Sent by the listener to request chunks from the distributor.

[4 bytes] body-len The length of the body
[X bytes] Body The rlp-encoded get-chunks transaction

Table 6.4.1. Chunk request

Chunk-reply

Sent by the distributor as a reply to the chunk-request.

<table>
<tr><td>[4 bytes] chunk-index
The first chunk-id</td><td>[4 bytes] body-len
The length of the body</td></tr>
<tr><td colspan="2">[X bytes] Body
Contains the mp3-data of the given chunks</td></tr>
</table>

Table 6.4.2. Chunk reply

7. Component Design

This section will dive deeper into how we designed our system. It will include a detailed explanation of the design process of each component. We will explain our thought process behind the component, by explaining the component's purpose, its application flow and other relevant design choices. For example, the listener component will include a section about UI design, whereas the Distributor component has a section that explains its command line interface. By explaining our design we want to give the reader a better understanding of what we aim to achieve in this project.

7.1. Listener

The listener client is the component that allows the user to stream music. Because it is the component of the system that most users will interact with, we chose to make it a mobile application with a graphical user interface. In this section, we will give an in-depth overview of the design of the listener component. This includes an activity diagram that was used to model the listener's behavior, the wireframes that show our intended design for the mobile application and a description of the technologies used.

Application flow

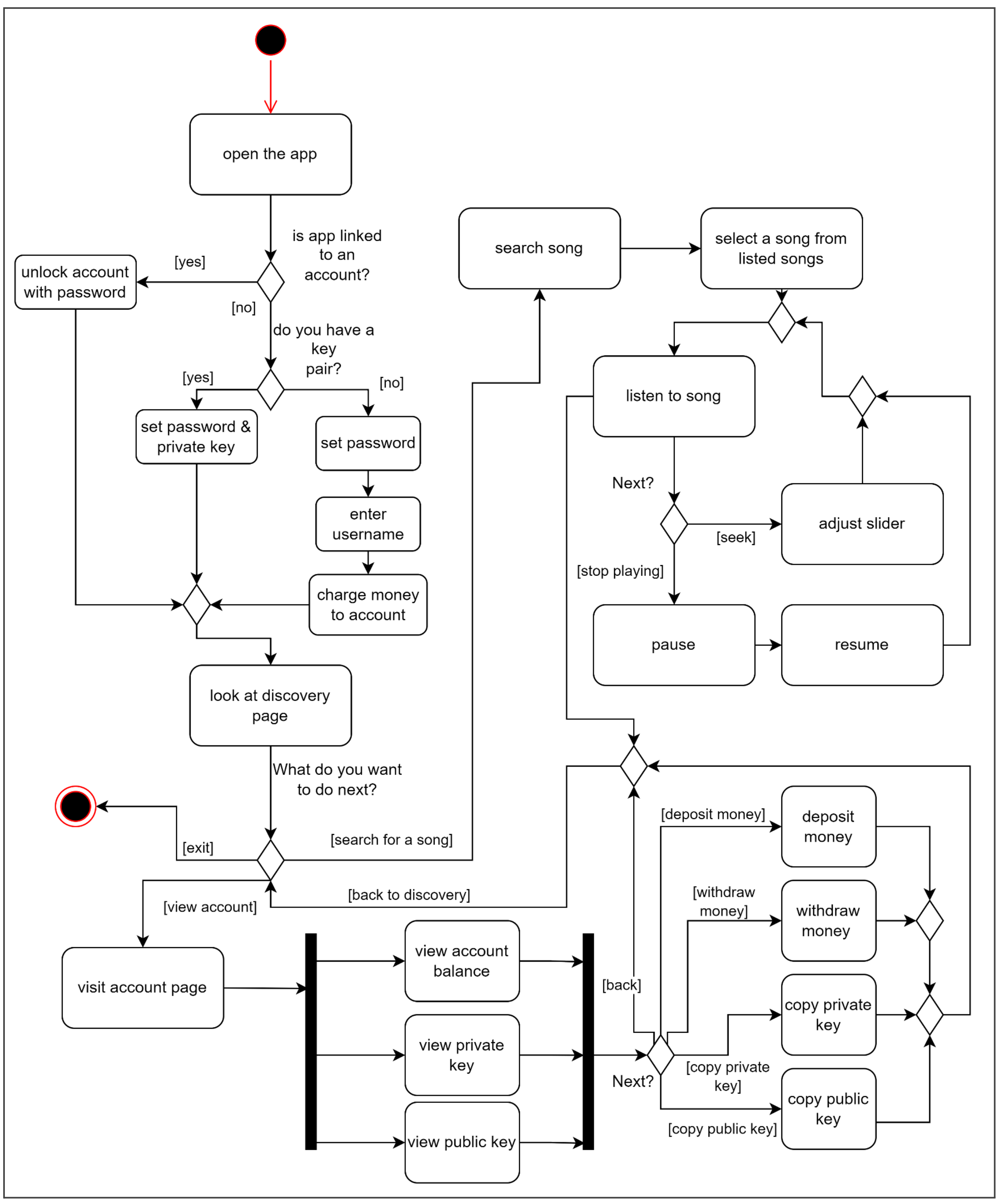


Figure 7.1.1. Listener - activity diagram

To model the application flow, we used an activity diagram. It presents the course of actions in our system and provides an overview of what behavior the application will perform when certain activities arise. In our case developing an activity diagram was useful for the design of the listener, because we could determine the listener's expected behavior. This made it easier to design wireframes and implement the code, because we did not have to think about what behavior the app should showcase for specific situations.

The starting point of the activity diagram is the opening of the app. First-time users will be asked to either register themselves or couple their existing account by entering their private key and a password. Users who have signed in before, are asked to unlock their account with a password when the app is started. If the user wishes to sign in with a different account, they can delete their private key from the phone after providing device authentication like their fingerprint. After users unlock their account, they will be navigated to the discovery page.

From this page users can either search for a song to listen to, or navigate to their account page. To search for a song the user clicks on the search bar and enters the name of the artist or name of the song. A list of songs will be generated and the user can click on the one they want to listen to. The song will start playing and the user will have the option to pause the song, adjust the slider to move forwards or backwards in the song, or click on another song. Finally, the user can navigate to the account page. Here they can view their current account balance, withdraw money from their account, deposit money into their account or copy their private and public key. The application can be exited from any page.

UI design

After the completion of the activity diagram we designed the wireframes. The wireframes are useful to help us think about the actual structure of the user interfaces, while keeping in mind user-friendliness and all functionalities that we want our system to have.

Designs are based on the activity diagram of the listener and the *must-have* and *should-have* requirements of the listener. Every page in the application has its own unique wireframe. When building the wireframes, we took into account which behavior is expected from the system and how the application should act for each individual user page. As an example, we modeled exactly what should happen whenever the user clicks on any button, searches for a song, or creates an account.

For the design of these wireframes we used Figma. Figma allowed us to work together while we developed our designs, as well as providing us with the tools to create our wireframes. The Figma pages include our design iterations before we landed on our final design. It also includes the final design, which has been split up into the different pages of the applications on Figma, so that it provides a clear overview of our actual design. These pages are: the splash screen, the register and log-in page, the discovery page, the library page and the account page. Screenshots of the different wireframes can be found in Appendix 2.

(See Appendix 2: Wireframes)

Technologies

Our aim for the listener was to develop a mobile application and for this, we chose to use Flutter as our primary framework. This decision was based on research we conducted about the available tools. Compared to its alternatives, mainly javascript-based libraries, Flutter provides native performance and hot reload for efficient performance. Additionally, we found Flutter libraries that suited all of our needs. This includes web3dart for sending transactions to the smart contract and just_audio for playing music. Finally, Flutter is also a tool that we were eager to learn more about and heard only good things about its developer experience.

State management

For the listener Flutter app it was also necessary to store states between pages. After all, when a user wants to see their balance, they do not want to fetch the balance from the smart contract each time. Therefore, variables such as balance, username, and the smart contract are stored in so-called ChangeNotifierProviders from the 'provider' package. These providers store the variables so that they can be accessed and changed across different pages.

Continuous integration and development

Upon any push to the main branch of the repository, the following GitHub Action is automatically run:

1. Flutter dependencies are fetched
2. An Android APK is created
3. The APK is released on the repository

The release is labeled with a version number in the format of `vX.X.X`, which increases with every push.

7.2. Distributor

The distributor is a client that allows users to distribute music within the network. It is implemented as a Command Line Interface (CLI) written in Rust and can be built for Linux, Windows or MacOs on both x86 and ARM. Rust was chosen because of very good ethereum support, ease of building scalable and fault-tolerant applications and developer experience.

The most important libraries used are ethers-rs for smart contract interaction, tokio for concurrency and networking, toml as a configuration format and sqlite for permanent storage. This combination allows us to easily build a multithreaded asynchronous server which can serve many listeners concurrently and is extremely fault-tolerant.

Application flow

The activity diagram below provides an overview of how the distributor runs commands. The command can be classified either as a transient command – songs, song-index, account, wallet, – or as the permanent command distribute. For the transient commands the application runs the command, which is completely different for each one, and then exits.

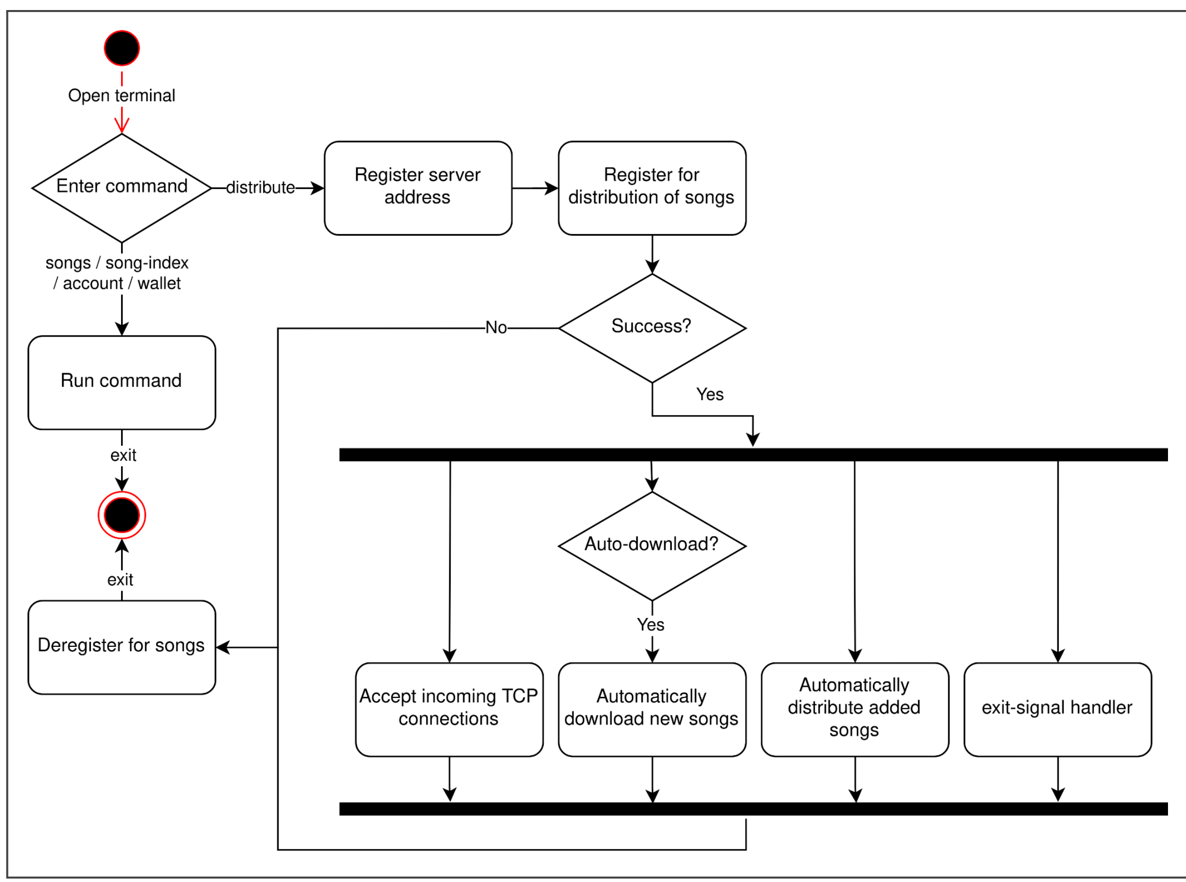


Figure 7.2.1. Distributor CLI - activity diagram

The distribute command will first attempt to register for the distribution of all songs stored in the database. If this is unsuccessful then it will deregister for those songs that were registered and then exit. In the case that registration is successful, the application splits into multiple parallel processes: The main process accepts incoming TCP-connections and sends back music chunks. There is a process that automatically registers for the distribution of added songs, a process that listens for an exit-signal by the OS and optionally a process that automatically downloads new songs from other distributors. If any of these processes exit, then all processes are aborted and the distributor deregisters the distribution for all songs.

The distributor was implemented as a CLI, specifically because this is the easiest way to deploy an application. All commands have been designed in such a way that automatic deployment is made easy. The client is implemented such that it is possible to run commands while having a distribution-process running in the background.

Streaming music

The streaming process of the distributor is the most central aspect of the distributor since this is where the music distribution actually occurs. For every listener that connects to a distributor, a new process is spawned. The diagram below describes what happens when a connected listener requests chunks over an already established TCP connection.

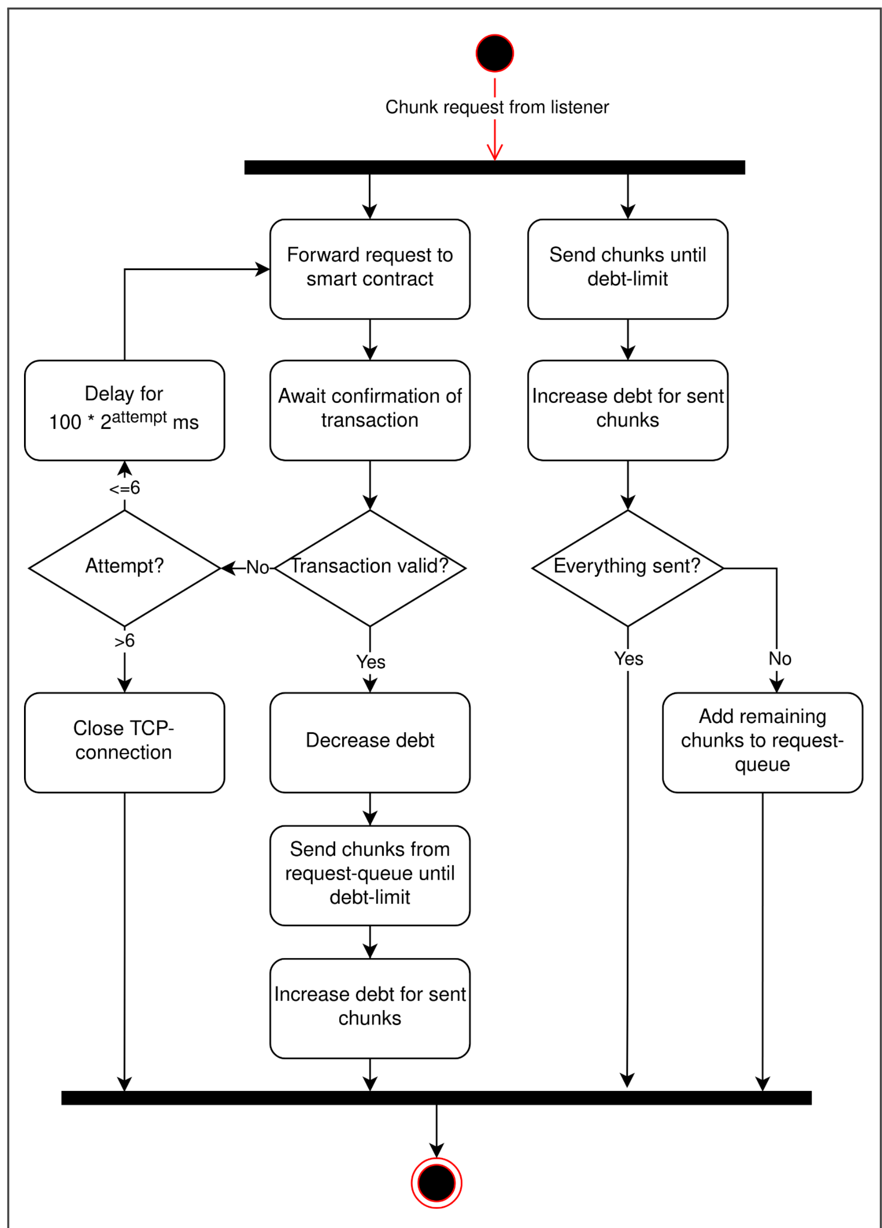


Figure 7.2.2. Distributor streaming - activity diagram

As can be seen, the request immediately splits into two concurrent events: One forwards the transaction to the smart contract to validate it, and the other immediately sends back chunks until the debt-limit has been reached for this particular listener. If the debt-limit is reached before all chunks could be sent then the remaining chunks are added to the request-queue.

The transaction that was forwarded to the smart contract takes some time to be confirmed or rejected. If the transaction is successfully confirmed and validated, then the debt is decreased by the size of the transaction, and chunks can be sent from the request-queue until the debt-limit is reached. If the transaction was not confirmed successfully, then the transaction is sent again using an exponential backoff with a maximum of 6 attempts. This results in a maximum timeout of $100 \ * \ 2^6 \ = \ 6.4$ seconds. (*See Discussion: IOTA Shimmer for more details).* If the transaction fails after the 6th attempt, the TCP-connection is closed.

Persistent storage

For persistent storage, the distributor uses the configuration file along with an SQLite database file. The following figure shows the SQLite database as used in the distributor application.

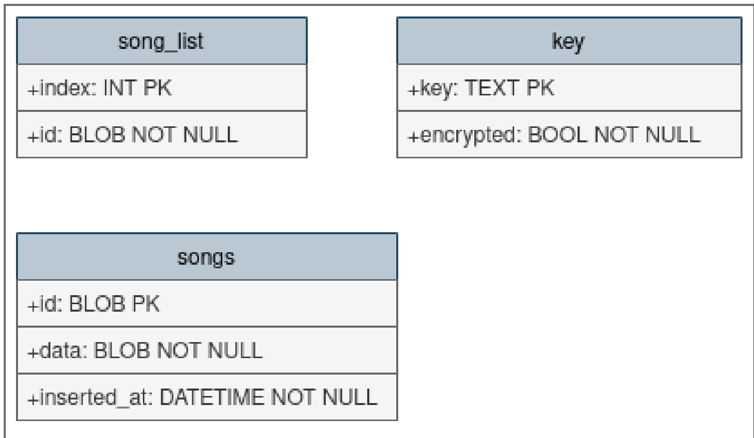


Figure 7.2.3. SQLite database for distributor

It uses a very simple database that does not have any foreign keys. The key table only ever contains a zero or one row, which stores the (encrypted) private key of the user. The song-list contains a local copy of all songs uploaded to the smart contract. It only stores the index in the list and the song-id. The songs table is the most important table and contains the data of all downloaded songs. In addition this table stores an inserted_at timestamp, which is used for the automatic distribution of newly added songs.

Continuous integration and development

In order to aid with continuous integration and development, automatic processes are run with GitHub actions. Upon any pull-request or push to the main branch the following actions are automatically executed:

1. **Check**: Whether the code compiles correctly.
2. **Rustfmt**: Whether all code has been formatted according to the rust-fmt standards.
3. **Clippy**: Whether Clippy finds any potential errors in the code.
4. **Test Suite**: Whether all tests pass correctly.

Whenever a tag with format `vX.X.X` is pushed to main, a release is automatically created and binaries for `aarch64-unknown-linux-gnu` and `x86_64-unknown-linux-gnu` are published alongside the release.

7.3. Validator

The validator component is a website that allows new content to be uploaded to the platform. This software was made to fulfill the right-holder and validator requirements through a simple UI. Rights-holders can create an account and request their music to be uploaded and validators can inspect the request and decide whether to approve or reject a song.

The backend of the website is built using Nodejs which was chosen because of its ease of use as well as for the existing Ethereum libraries. The main libraries used were ethers to connect with the smart contract and express to handle HTTP requests. The frontend requires the MetaMask plugin which allows for direct interaction with the smart contract and stores the user's private keys.

Application flow

The activity diagram below provides an overview of how users would interact with the website. The authentication logic is done at the smart contract level and it is verified by the website's backend after each HTTP request. Therefore, even if a user were to bypass the webapp's authentication logic, any attempt to change the smart contract's state would be blocked.

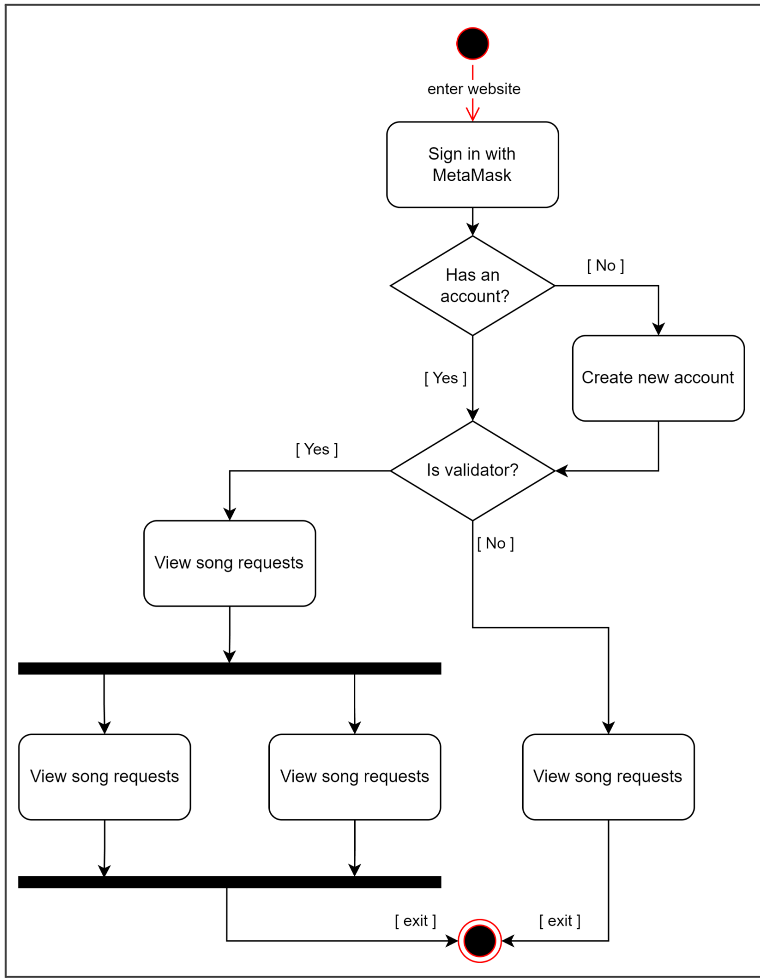


Figure 7.3.1. Validator - Activity diagram

The first step when entering the website is to sign in using MetaMask. To achieve this, the user has to sign a random nonce generated by the website and return the signature. The backend then verifies the signer of the signature and stores the address in a secure cookie that can be used to request access to all the other services in the web application. This authentication logic requires little user interaction and is quite common among decentralized applications, like OpenSea, the largest NFT marketplace.

Once users have signed in, if their wallet address is not yet linked to an account, they can create a new account and access the main functionalities of the application. Regular users, in this context, are considered right-holders once they request an upload of music from the validator. To achieve this, a simple form must be filled out, where the user provides the wallet address of the song author, the name of the song, its price and the music file as an mp3. Additionally, the user has to provide an email address that is used by the validator to request additional information about the identity of the right-holder. All the information provided is signed using the right-holder's wallet address. Finally, the form with signature is stored in the backend of the web application awaiting a validator to have a look at it.

Validators can also request content for the platform, but they automatically access the validator UI when signing in. This page displays all the information about songs requested through the application. The validator can download or play the music file and use the provided email to verify that the request respects all the relevant copyright laws. If the validator decides that this is not the case, rejecting the request will remove all its information including the file from the backend storage. On the other hand, approving the song requires the validator to sign a transaction which is sent directly to the smart contract where the metadata will be stored. On the backend of the web application, the file is manually entered into the database used by a distributor's client. This client is owned by the website deployer which will be the first user to distribute the song allowing other distributors to download it and spread the file around the network.

UI design

This web application's development was initially considered the lowest-priority component as we wanted to focus on the design of the software used by the listener. That is why, except for the activity diagram, there was no design phase previous to developing the application. The UI that can be seen in the final product is the result of using a simplistic CSS framework called Pico.css meant for demos. We tuned it to include the mobile application's color palette.

This decision came as an advantage at the end of the project as we were free to include extra functionality due to excess time. We implemented features to help us demonstrate and explain the platform's infrastructure visually. This can be seen in the song browser UI, which displays all the songs available in the platform. Selecting any track will then display all the information available about its distributors, such as their IP address or fee. Additionally, we showcase the network by locating distributors on a map using their IP addresses.

7.4. Smart contract

A smart contract is a piece of code that can be stored in a distributed ledger and executed by its nodes. This code may contain a series of variables as well as one or more functions which either returns the values of these variables or modifies them. Usually, a smart contract is compared to a state machine where all the possible combinations of its variables' values are the potential states and executing its functions is the way to move from one state to another. Using these kinds of programs allows different projects to have a decentralized back-end logic as all the nodes in the network execute the code and agree on the output state. Therefore, a smart contract ensures that the service is always available and that its contents cannot be tampered with by any malicious actor.

We use the IOTA Smart Contract Protocol [5] as it is a scalable distributed ledger technology which allows us to deploy a custom chain in which to store and execute our smart contract. The full-node software implemented by IOTA for this protocol is called Wasp and, even though it is still in an early development phase, it is capable of fulfilling most of our requirements. A big advantage of using the Wasp node is that a custom implementation of the Ethereum Virtual Machine is available. This is especially advantageous because we can use Ethereum libraries in all the other components to interact with the smart contract without having to deploy the smart contract in Ethereum, which has problems with scalability and efficiency.

Data storage

This project’s smart contract's main functionality is to keep track of active songs with their distributors and to manage payments when music is streamed. During the design phase, we made the following Extended Entity Relationship diagram to illustrate how data will be stored in our code. We used an EER diagram, even though they are meant for designing databases because there does not exist a standard methodology to design smart contracts. The main way to store structured information in Solidity is by using hash tables called mappings where key values are paired with data structures. This can be compared with standard databases where each mapping in the smart contract is equivalent to a database table. As can be seen in the figure, the smart contract is composed of three different mappings: users, songs and distributions.

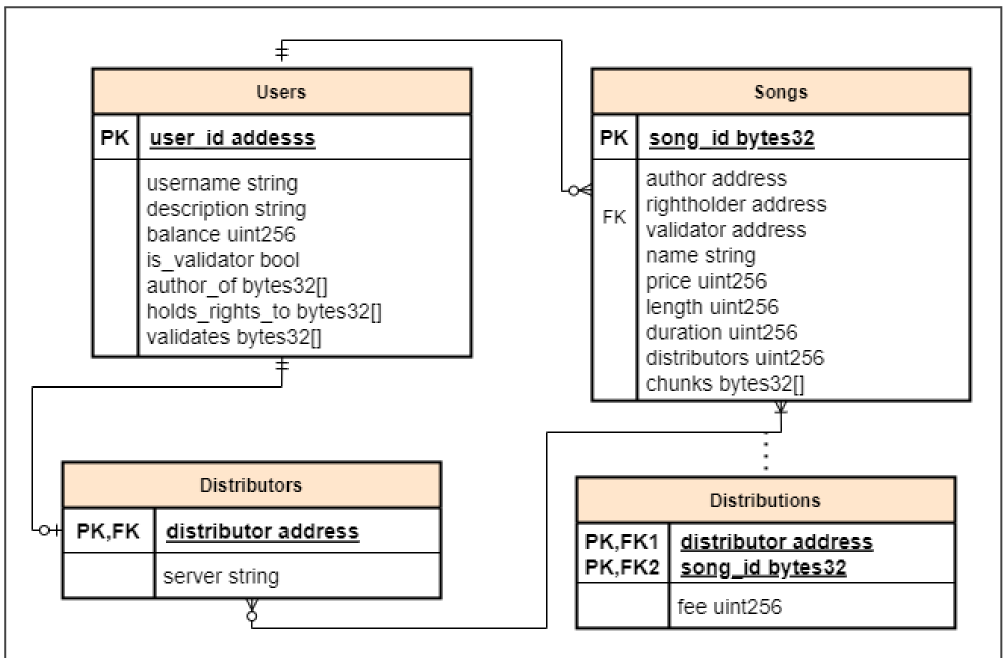


Figure 7.4.1. Smart contract - Class diagram

Users

First, the users' mapping pairs wallet addresses with a user data structure containing information like usernames and descriptions. The balance of each account is tracked using an integer variable and the status as a validator is tracked using a boolean. Also, if the account is used to distribute songs, there is a string variable reserved to store information about how to reach its server which may contain IP address, port, protocol and even a public key certificate to encrypt traffic. Finally, a list of song ids is kept for each user to track the different kinds of involvement in the music that is available. This involvement is divided into three lists, one is for songs that the user is the author of, another one is for songs the user holds the rights for and the last one is for songs that the user has validated.

Songs

All the songs available in the system are kept in the songs' mapping. This pairs song identification values with a song data structure. The identification value is unique for each song and corresponds to the hash value of the song's name and the author's wallet address. This value ensures that it is possible to have multiple songs with the same name but only one per author. The song data structure contains metadata like the song's name and price as well as the wallet addresses of all the involved parties: the author, the right-holder and the validator of the song. It also includes other useful information for the listeners' software like the song's duration in seconds and the file's length in bytes. This is to easily start any audio player, as well as a list of the hashed values of each chunk of the song's original file. This checks if the data that is being received when streaming is authentic.

Distributions

Finally, the mapping for distribution is a bit more complex than the other two because information should be stored, ensuring that it is possible to retrieve an ordered list of distributors based on their fees per song. To solve this problem, we implemented a data structure that mimics an ordered linked list which is not directly available in Solidity. This is achieved by using a mapping that pairs the distribution identification value to the fee as well as the address of the next distributor in the list. The head node's identification value for each list always corresponds to the identification value of the song for which the list is meant. Then, the identification value for each distributor corresponds to the hash value of its wallet address and the identification value of the song it is distributing which ensures that all users can distribute all songs. This data structure, although complex, allows us to add and remove distributors of each song while maintaining the list sorted by fees.

Tamper-proof uploading

The most complex functionality of the smart contract is the mechanism behind the process of uploading new songs. That is because the platform requires strict authorization where only validators are allowed to upload but only with a request received from a right-holder. To achieve this, we use digital signatures to prove the identity of the right-holder and ensure that the request parameters are not tampered with. These signatures are a standard functionality of Ethereum which implements the Elliptic Curve Digital Signature Algorithm (ECDSA).

The right-holder provides the metadata of a file which he wants to upload as well as a nonce, determined by the number of songs linked to the author. These parameters are subsequently signed with the right-holder’s wallet address, ensuring that the song is only uploaded once for a given request. Additionally, any modification to the request is noticeable due to a signature that doesn’t match the data. Once the validator approves the request, it is sent, alongside the signature, to the smart contract. This mechanism allows the right-holder to upload music through a validator without exposing their private key or risking a tampered request.

8. Product

This chapter presents the final results of the final components that have been delivered. These components are the listener GUI, the distributor CLI, the validator website and the smart contract. In addition we describe our deployed instances of the validator, distributor and smart contract. Wherever applicable, screenshots of the products have been provided as appendices.

8.1. Listener GUI

We implemented the GUI of the Listener based on the activity diagram and the wireframes that we developed in the design phase as mentioned in chapter 7. We used these designs as guidelines for our actual implementation of the user interfaces. During the implementation phase, we discovered that some of our design choices did not work as good in reality as they did on paper. We also realized that there were critical elements missing to our Listener application. For example, we added a help page because we realized that user's might need help with understanding some parts of the application. User-friendliness is always our main priority. Because our actual implementation of the Listener UI deviated quite significantly from the design of the Listener component, we chose to display a table that gives an overview of all the different pages in our GUI with a thorough per page explanation. Screenshots of the actual GUI can be found in Appendix 3.

(See Appendix 3: Listener User Interfaces)

Page of GUI	Description
Register page - Enter password page - Enter username page - Charge money to account page	This is the page that the user lands on when they open the app for the first time. They are asked to create an account by filling in their password, a username and charge money to their account. On completion, they can move to the discovery page of the app.
Couple account page	If a user already has an account they can enter their password and private key to couple it. After coupling it they will be directed to the discovery page of the app.
Unlock account page	The user will be asked to unlock their account with their password when the app is still running in the background.

Discovery page	This is the page where the user can select a song to play. A list with all songs is displayed that the user can search by entering a song name or artist name in the search bar. By clicking on a song in the list, a pop-up will appear that asks the user if they are okay with paying the price for the song. If they say yes, the song will start playing.
Account page - personal details	The account page has the user's personal details, such as their balance, their public key and their private key. Money can be deposited and withdrawn from the balance. Moreover, the user's public key can be copied to the clipboard and upon entering the user's password the same can be done to the private key of the user.
Account page - smart contract details	The smart contract details page contains the smart contract address, the hex value and the chain id. The user has the option to change the details too with a button that redirects the user to a page where they can change it.
Change smart contract details page	In this page the user can change the smart contract details by changing the smart contract address, the hex value and the chain id. On save, the details will be stored.
Help page	The help page can be accessed from every page in the application where user questions might arise. This page answers all the possible questions that the user might have.

Table 8.1.1. Description of the Listener GUI

8.2. Distributor CLI

The distributor client is implemented as a simple Command Line Interface (CLI) with commands and subcommands to manage the different functionalities of the client. The CLI must be accompanied by a `TangleTunes.toml` configuration file. This file is used to store the configuration-values necessary to run a distributor. For screenshots of the CLI and configuration format, see *Appendix 5: Distributor CLI and Configuration.*

Command Line Interface

The command line interface can be grouped into five main command groups. Each of these commands is further detailed in the table below. All commands have supplementary flags to customize their behavior.

CLI commands	Description
`wallet` `import, generate, remove, address, private-key, balance, request-funds.`	Commands for managing the IOTA wallet. With these commands it is possible to import, generate or remove the wallet, view its information and request funds from the faucet (on a test-network).
`account` `deposit, withdraw, create, delete, view.`	Commands for managing your TangleTunes account. This can be used to deposit or withdraw to/from the wallet and the account can be created, viewed and deleted.
`songs` `download, add, remove, list.`	Commands for managing the songs stored locally. The songs can be added, listed and viewed manually or downloaded from another distributor.
`song-index` `update, reset, list, download.`	Commands for managing the local copy of the song-index on the smart contract. The index can be updated, reset and viewed. In addition it is possible to download a song from another distributor using its index.
`distribute`	The main command used for running the distributor. It has an optional flag which can automatically download new songs.

Table 8.2.1. Command line interface commands

Configuration file

The configuration format is a toml file whose path is passed as an argument to the CLI. All parameters of this file are explained below.

Parameter	Description
`server_address`	The server address registered on the smart-contract. (IP:port)
`bind_address`	The address that the distributor binds on.
`database_path`	The path where the database file can be found.
`fee`	The distribution fee in IOTA/chunk.

max_price	The (optional) maximum price in IOTA/chunk when automatically downloading new songs.
chain_id	The chain-id of the smart-contract.
contract_address	The contract-address of the smart-contract.
node_url	The Shimmer node to contact.

Table 8.2.2. Parameters of configuration file

8.3. Validator website

We implemented the GUI of the validaotor's website based on the activity diagram that we developed during the design phase. During the implementation phase, we realized that the other components would have difficulties requesting funds to the private chain. To help with their progress, we created a debug page which was not anticipated during the design phase. The following table shows an overview of all the different pages in our GUI with a thorough per page explanation. Additionally the table shows which pages require the MetaMask plugin to be correctly installed. Screenshots of the actual GUI can be found in Appendix 4.

(See Appendix 4: Validator website UI)

Page of GUI	Plugin	Description
/	**No**	There is a link to our github organization as well as a link to download the mobile app installer. Finally, if MetaMask is connected, there is a button to sign up which will redirect to a different page.
/validator/register	**Yes**	There is a form to fill before signing and sending a transaction to create a new account on the smart contract. Additionally, if the user has no funds, there is a button to request from the faucet
/validator/request	**Yes**	There is a form to fill before sending a request. The user has to include the wallet address of the author, the name and price of the song along with its file. The UI will display if the wallet address is correct and the price equivalent that will be displayed on the mobile app.
/validator/validate	**Yes**	There is a list of requests with the author's name and the song's name. Selecting any of these will display the entire request data including a player for the song file. Finally there is a button to approve and another to reject the song.
/browser/songs	**Yes**	There is a list of available songs. Selecting any of them will display its price along with information about all its distributors. Finally, a map is used to place markers for each distributor based on the approximate location of their IP addresses.

/debug/info	No	Displays all the information needed to interact with the smart contract in json format
/debug/faucet	No	There is form to filled before requesting funds to the provided wallet address

Table 8.3.1. Description of validator website pages

8.4. Smart contract

The smart contract is a single Solidity file that functions as the backend logic of the system. It can be compiled using Solidity 8 with optimization enabled. It can then be deployed in any EVM, even though it is meant to run on IOTA's implementation. To correctly compile the code, a small IOTA library is required. This is already included in the source code and it is used to call a special function which triggers the IOTA nodes to transfer funds from the smart contract to a given address in the tangle. Finally, the source code is accompanied with a documentation file. It consists of a smart contract interface which includes all the existing functions as well as an explanation of what they do and how to use them. This document is especially useful for the developers of other components as it can be used to learn how to interact with the code without Solidity knowledge. The document is too long to be included in this report but can be found on github at [TangleTunes/smart_contract/blob/main/contracts/documentation/TangleTunes.sol]().

Solidity standards for documentation are quite similar to Java. The following image shows the information of the function used by users to pay for chunks of a file. As can be observed, the documentation provides a short description of what the function will do as well as a list of required parameters along with their type.

```
/**
 * @notice pay author and distributor for a given amount of consecutive chunks
 * @param _song identification value
 * @param _index of the first chunk
 * @param _amount of consecutive chunks
 * @param _distributor address
 */
function get_chunks(bytes32 _song, uint _index, uint _amount, address _distributor) external;
```

Figure 8.4.1. Documentation of state-changing function 'get_chunks'

Additionally, a list of the data returned is included along with their type if the function does not change the state of the smart contract and only returns the value of some variable. The following image shows one of these view functions in which it is specified that only one song identification value is returned. Finally, extra comments can be found disclaiming possible edge cases which are indicated with the dev tag. For example the following image warns about the possibility of finding values corresponding to removed files in the output of the function.

```
/**
 * @notice provides song identification value of a given index
 * @dev id may correspond to a song that no longer exists
 * @param _index in the list of songs (starting at 0)
 * @return song id
 */
function song_list(uint _index) external view returns (bytes32);
```

Figure 8.4.2. Documentation of view function 'song_list'

8.5. Deployment

The deployment of the system is carried out in three distinct parts: the smart contract/L2 chain, the validator, and the distributors. These deployed components work together to create a fully functioning network that the listener client can use to stream music via TangleTunes.

Smart contract

We first setted up a Raspberry Pi openly connected to the internet. Then, we deployed the IOTA software in this device so that any component in the system has access to the distributed ledger. IOTA's software consists of two nodes, the Hornet nest which corresponds to the Tangle also called L1 and the Wasp node capable of running multiple L2 chains anchored to L1. Using this software we started our private chain in which we deployed the smart contract.

Validator

We then run the validator's website on the same Raspberry Pi, ensuring that anyone can upload songs to the platform. Finally, we got the tangletunes.com domain and set it up to point to the device's IP address. This is specially useful for the other components as they can be compiled to connect to this domain by default. Therefore, if the device's IP ever changes, we can modify the DNS records instead of changing the source code and recompiling our software.

Host name	Type	TTL	Data
tangletunes.com	A	1 hour	217.104.126.34
evm.tangletunes.com	A	1 hour	217.104.126.34

Figure 8.5.1. DNS records of tangletunes

Distributor

We deployed three separate distributors on DigitalOcean instances located in London, Frankfurt, and New York. These instances run the distributor software with the --demo flag. This flag instructs the distributor to automatically download any new songs uploaded to the network. The London instance downloads even-numbered songs, Frankfurt downloads odd-numbered songs, and New York downloads both even and odd-numbered songs. Distribution prices are set within a reasonable range using random values. Collectively, these features emulate the behavior of a network with actual distributors in operation.

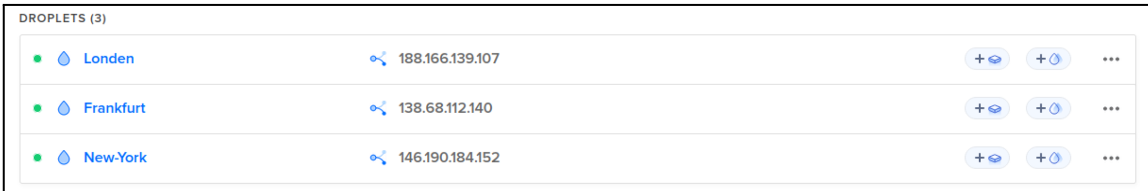


Figure 8.5.2. Deployment on DigitalOcean

9. Testing and evaluation

Testing played a significant role in the development of our system as it helped to ensure that the software behaves as expected. Each component of our system was tested using a set of different types of tests depending on what was most suitable. The most important tests regarding the listener application were manual test cases because they best mimic how the application is interacted with and how it is meant to behave. Unit tests were conducted where possible, but because of the large number of side effects that most methods had, this was limited to only a few parts. The distributor was tested using manual test cases for each of the important commands/functionalities. The smart contract was tested using unit tests, as the smart contract comprises discrete methods that could be tested with automation.

9.1. Listener Testing

Unit Testing

Some, but not all classes of the listener have been tested with unit tests. There are many ‘boundary’ classes which rely on external actors such as the distributor and the smart contract. We decided to not perform unit tests on these classes because there is no way to know whether such a test fails due to external factors or due to actual errors in the code. There were also many classes which relied on ‘providers’ which are not easily testable with unit tests either. We also decided to exclude these classes. The unit tests have been performed with the default flutter_test package.

(See Appendix 6: Test results)

Class	Method	Test case	Expected outcome	Pass
price_conversions.dart	weiToMiota	1000000000000000000 wei	1 MIOTA	✅
	weiToMiota	1000000123456111215 wei	1 MIOTA	✅
	miotaToWei	3 MIOTA	3000000000000000000 wei	✅
	priceInMiotaPerMinute	1000000000000000000 wei per chunk for a 60 second long 32500 byte song	1 MIOTA per minute	✅
file_writer.dart	writeToFile	write “test” to test.txt	The file contains “test”	✅

Table 9.1.1. Unit tests of listener client

Manual Testing

The following manual tests have been performed on the Listener application. Each test case can have assumptions that need to be fulfilled. Some assumptions are abbreviated here for conciseness of the table:

- **Contract reachable**: The smart contract is deployed and the app has saved the correct RPC URL, address of the contract and chain ID locally.
- **External private key**: The user has generated a keypair externally.
- **Private key coupled**: There is a private key and it is coupled to the user account
- **L2 funds (sufficient/insufficient)**: The user either has sufficient layer 2 funds for a transaction or they do not
- **Account on SC (exists/does not exist)**: The public key is registered as a user on the smart contract or it is not.
- **Logged in**: The user has unlocked the app.
- **Account funds (sufficient/insufficient)**: The user either has sufficient or insufficient funds on tier account
- **Distributor (available/unavailable)**: There either is a distributor available for the song that is selected or there is not

Test description	Assumptions	Expected outcome	Pass
Creating an account 1) On a fresh install of the app, choose “Create” account” 2) Choose a password and repeat it	Contract reachable.	The keypair is generated. The user will be prompted to provide a username. After that, the user will be prompted to add funds to their account. After that, the user will be taken to the discovery page.	✅
Coupling an account. 1) On a fresh install of the app, choose “Already have a wallet? "Connect it.” 2) Choose a password and repeat it 3) Enter your private key key 4) Click “Couple Account”	Contract reachable. External private key. Insufficient L2 funds. Account on SC exists.	A screen that asks the user to deposit money is presented and entering the discovery page will only succeed if the user has charged money on their wallet externally.	✅
	Contract reachable. External private key. Sufficient L2 funds. Account on SC exists.	The user is redirected to the discovery page.	✅

	Contract reachable. External private key. Insufficient L2 funds. Account on SC does not exist.	First, the user is asked to provide a username. Then the user is asked to deposit money. When the user has deposited money, they can “Continue” on the page that asks for their username and are redirected to the discovery page.	✅
	Contract reachable. External private key. Sufficient funds. Account on SC exists.	The user is redirected to the discovery page.	✅
Unlocking an existing account. 1) Enter your password and press continue	Contract reachable Private key coupled	The user unlocks their account and is taken to the discovery page	✅
Deposit. 1) On the account page, enter a positive number x and press deposit	Contract reachable Private key coupled Insufficient L2 funds Account on SC exists Logged in.	The deposit transaction will fail since the user does not have sufficient L2 funds	✅
	Contract reachable Private key coupled Sufficient L2 funds Account on SC exists Logged in.	The deposit transaction succeeds and the balance of the user is increased by x while the L2 funds are decreased by x.	✅
Withdraw 1) On the account page, enter a positive number x and press withdraw	Contract reachable Insufficient account funds Account on SC exists Logged in	The withdraw transaction will fail since the user does not have sufficient funds on their account	✅
	Contract reachable Sufficient account funds Account on SC exists Logged in	The withdraw transaction succeeds and the balance of the user is decreased by x while the L2 funds are increased by x.	✅
Playing a song 1) Open the discovery page of the app 2) Select a song	Contract reachable. Logged in. Distributor unavailable	The distributor is unavailable so a toast with “Distributor not available” will be displayed	✅

	Contract reachable. Logged in. Insufficient account funds Distributor available	The user does not have enough funds so after the start of the song the player will try to load but fails.	✅
	Contract reachable. Logged in. Sufficient account funds Distributor available	The song plays.	✅
Retrieving the list of songs 1) Open the discovery page of the app	Contract reachable. Logged in.	The app shows a list of all songs.	✅

Table 9.1.2. Tests for interactions with the smart contract

Test description	Assumptions	Expected outcome	Pass
User changes smart contract details 1) Open the account page of the app 2) Go to the smart contract settings tab 3) Enter the new values and press the 'Confirm changes' button	Logged in	The smart contract settings are changed if the entered values are valid, if not, the user will be prompted again to change them.	✅
The smart contract is no longer reachable Given that the smart contract information stored in the app is false, the following actions should fail: ● trying to create an account ● trying to deposit or withdraw ● trying to click on a song which then tries to find a distributor ● trying to open the discovery page (which contains the list of songs)	None	The user is prompted to enter the smart contract details (RPC URL, address, chain ID). Only if the provided information can be reached, the user can leave that page.	✅

Table 9.1.3. Test for changing of smart contract details

Test description	Assumptions	Expected outcome	Pass
User seeks into a buffered part 1) Play a song 2) Seek into a buffered part	Logged in Contract reachable. Distributor available.	The player instantaneously skips to the part that the user selected and the audio plays.	✅
User seeks ahead into unbuffered part 1) Play a song 2) Seek into an unbuffered part	Logged in Contract reachable. Distributor available.	The player skips to the part that the user selected, pauses for a bit to fetch this part of the song and then plays the audio.	✅

Table 9.1.4. Tests for seeking in a song

Test description	Assumptions	Expected outcome	Pass
User wants to access the help page before the smart contract is initialized, through the following pages: ● Create account ● Couple account ● Smart contract settings		An orange button with a question mark is visible in the bottom right. When pressed, the help page is shown.	✅
User wants to access the help page when the smart contract has been initialized, through the following pages: ● Account ● Please deposit ● Provide username ● Unlock account	Contract reachable	An orange button with a question mark is visible in the bottom right. When pressed, the help page is shown and additionally there is an option to reset the smart contract nonce.	✅

Table 9.1.5. Tests for the availability of the help page

9.2. Distributor Testing

For the distributor, many parts of the system are very difficult to test automatically due to boundaries with external components. Essential aspects of the system have been automatically tested with unit tests, like password encryption/decryption, chunking of songs in the database and the streaming logic. The unit tests are automatically run whenever a push is made to the main branch on a GitHub Runner. Interactions with the smart-contract are not included here since the code is automatically generated using the ethers-rs library and is assumed to be correct.

(See Appendix 6: Test results)

Manual testing

In addition to the unit tests, the distributor's commands are also manually tested before new releases are made to ensure their correctness. These tests are simple tests that ensure the client works under normal circumstances. The tests below have been checked for release v0.1.2.

Test description	Assumptions	Expected outcome	Pass
Create plaintext wallet 1) Run import <PRIVATE_KEY> or generate with -plaintext. 2) Run wallet balance. 3) Run wallet request-funds. 4) Run wallet balance.	Valid TangleTunes.toml file.	First time running wallet balance should be 100 Mi less than the second time.	✅
Create encrypted wallet 1) Run import <PRIVATE_KEY> or generate with -password. 2) Run wallet balance with the correct password. 3) Run wallet balance with an incorrect password.	Valid TangleTunes.toml file.	First balance command should complete and the second should fail.	✅
Create account 1) Run deposit 10000000. 2) Run account create -name my_name. 3) Run account view.	Valid TangleTunes.toml file. Wallet with > 100 Mi.	The user now has an account with 10 Mi balance and username my_name.	✅
Update song-index 1) Run song-index reset. 2) Run song-index view. 3) Run song-index update. 4) Run song-index view	Valid TangleTunes.toml file. Account with funds.	The first view should give an empty song-index, while the second should show all songs on the platform.	✅

Add song 1) Download an mp3 registered on the platform, and name it <SONG_ID>.mp3 2) Run songs add ./<SONG_ID>.mp3 3) Run songs list.	Valid TangleTunes.toml file. Song has not yet been added.	The song should show up in the list with the given song-id.	✅
Download song 1) Run songs download –song-id <SONG_ID>. 2) Run songs list.	Valid TangleTunes.toml file. Account with funds. Song has not yet been added. Song has a distributor.	The song should show up in the list with the given song-id.	✅
Distribute 1) Run distribute. 2) Wait for the registration process to finish. 3) Run download –song-id <SONG_ID> on another distributor. (song should be distributed, repeat until this one is chosen) 4) Wait for the song to download. 5) For the running distributor, add a song with songs add –song-id <SONG_ID>. (song should not be distributed yet) 6) Press ctrl-c.	Contract reachable. Logged in. Account with funds for both distributors.	The distributor should register his server address and then register for distribution of all songs in the database. When the song is being downloaded, this should complete successfully on the other distributor and print logs. When the song is added to the database, it should automatically register for distribution. Shutting down the application should deregister for all songs it is distributing.	✅

Table 9.2.1. Manual test cases for distributor

9.3. Smart Contract Testing

The smart contract is tested extensively with automatic unit tests. This phase of smart contract development is crucial for the correctness of the entire platform. Even though this is the smallest component in the project, the nature of programmability in distributed ledgers means that, once deployed, the code cannot be updated. Therefore, any bug or vulnerability that is not caught during development will stay accessible forever.

Each requirement is tested individually at least once, and edge-cases like accounts or songs being removed, are also tested. These cases can be divided into three different categories. First, account management includes all test cases where users set up and manipulate their

accounts. Then, song management contains all test cases with the assumption that users can manipulate their accounts and the song uploading and removal mechanisms are tested. Finally, distribution management includes all test cases for registration and deregistration of distributors with the assumption that the platform already has users and songs.

All unit tests are implemented using the Nodejs package called hardhat. This library runs our smart contract on a simulation of the Ethereum blockchain without requiring a deployed infrastructure. This solution makes the development phase easier but has a few downsides for our project. The main downside is that it does not simulate the exact version of the EVM that we deploy our contract to. This could cause problems as the official EVM may differ from IOTA's implementation but we do not have found this to be an issue. Because of this issue, we are not able to test if the existing function for withdrawing funds to the tangle works as this concept does not exist in Ethereum. Therefore, this function is tested manually after deploying the smart contract. As can be observed in the appendix, there is a test for this function that always fails as a reminder that we must perform the manual test.

(See Appendix 6: Test results)

Methods tested	Test description	Pass
Account Management: ● create_user ● edit_description ● edit_server_info ● delete_user ● deposit ● withdraw_to_chain	User should be able to create account	✅
	User should be able to edit description	✅
	User should be able to edit server info	✅
	User should be able to remove account	✅
	User should be able to deposit	✅
	User should be able to withdraw to chain	✅
Validator Management: ● manage_validators	Deployer should be able to assign a validator	✅
	Deployer should be able to dismiss a validator	✅
Song Management: ● upload_song ● edit_price ● delete_song ● delete_user ● manage_validators	Validator should be able to upload a song	✅
	Right-holder should be able to change the price of a song	✅
	Right-holder should be able to delete their songs	✅
	Validator should be able to delete their songs	✅
	Song deletion when validator is dismissed	✅
	Song deletion when Author deletes their account	✅
	Song deletion when Rightholder deletes their account	✅
	Song deletion when Validator deletes their account	✅

Distribution Management: • distribute • undistribute • find_insert_indexes • find_dist_indexes • get_chunks • delete_song • delete_user • manage_validators	Distributor should be able to distribute a song	✅
	Multiple distributors should be able to distribute a song	✅
	Distributor should be able to decrease fee	✅
	Distributor should be able to increase fee	✅
	Distributor should be able to undistribute song	✅
	Remove all distributions when song is directly deleted	✅
	Remove all distributions when song is indirectly deleted	✅
	User can get chunks from a distributor	✅

Table 9.3.1. Smart contract unit tests

9.4. Validator Testing

The validator website has mostly been tested by manual testing, since the website is the least essential aspect of our platform. The validator is mostly intended as a reference for what such a system can look like, allowing third parties to implement better alternatives. The following table describes the manual test-cases used. These tests cover standard usage of the website.

Test description	Assumptions	Expected outcome	Pass
Creating an account 1) Visit "tangletunes.com". 2) Click "Request funds" and wait for confirmation. 3) Enter username and description. 4) Click "Register".	User has metamask installed with a wallet that does not have an account.	Routed to the "Create account" page.	✅
		After clicking on register, the user is routed to the "Upload a song to TangleTunes" page with their new account.	
Uploading a song 1) Visit "tangletunes.com". 2) Enter a name, price and contact email. 3) Upload an mp3-file by clicking on "Browse…". 4) Click "Request song"	User has metamask installed with a wallet that has an account.	Routed to the "Upload a song" page.	✅
		After requesting the song, the page resets to "Upload a song" without any filled fields.	
		After requesting the song, it is available in the validator-view.	

Validating a song 1) Visit “tangletunes.com” 2) Click on the song to be verified. 3) Click on the play button. 4) Click on “validate”. 5) Click on another song. 6) Click on “deny”.	User has metamask installed with a wallet that has a validator-account. At least two song-uploads have been requested.	Routed to the “Validate” page.	✅
		Clicking play should start playing the song.	
		After clicking validate, the song can be found with Remix.	
		After clicking validate, the song is distributed by the validator-distributor.	
		After clicking deny, the song can not be found with Remix.	
Browsing songs 1) Visit “tangletunes.com/browse/songs”. 2) Click on a song in the list.	User has metamask installed with a wallet.	Clicking a song should display its price.	✅
		Clicking a song should display a list of distributors on the right with their distribution prices.	
		Clicking a song should show their location on the map.	

Table 9.4.1. Manual test cases for validator

9.5. System testing

After testing the individual components of the system with the aforementioned tests, we performed system testing to see how the various components act together. The first step is always to upload a song as a right-holder. Songs of different lengths and with different prices were tested. These songs were then approved by a validator. The distributor is then able to distribute the song, so that the listener could listen to the song. This was also tested with multiple distributors across the globe to see whether the distributors were able to download songs from each other.

To make sure everything is displayed correctly in the listener app, songs with long names were uploaded to see if they would cause any issues. Initially, it resulted in the text overflowing but this issue had been resolved afterwards. It was also tested whether the listener would get different distributors across the globe if there were multiple distributors distributing the same song, this was indeed the case.

9.6. Evaluation

An important part of testing is evaluating how many of the initial requirements of the project have been achieved. Based on a comparison between the test cases and the list of requirements as mentioned in chapter 4 of this report, we can check which of the requirements have been fulfilled. More importantly, it also gives us an overview of which requirements still need to be implemented. This makes it easier for us to see which parts of the project are most in need of improvement. Below, the reader can find comprehensive lists of all functional and non-functional requirements that still need to be worked on.

Functional requirements

First, we will take a look at the achievement of functional requirements. This list was divided based on the different stakeholders and sorted based on the MoSCoW prioritization of requirements. The following table contains all functional requirements that were not implemented.

Stakeholder	Functional requirement
Listener	❖ A listener should be able to add songs to their library. (S) ❖ A listener should be able to remove songs from their library. (S) ❖ A listener should be able to search in their library. (S) ❖ A listener should be able to set a maximum price/chunk. (C) ❖ A listener should be able to add songs to the queue. (C) ❖ A listener should be able to remove songs from the queue.(C) ❖ A listener should be able to view their queue. (C) ❖ A listener should be able to select a listening-strategy. (W) ❖ A listener should be able to tip a right-holder. (W)
Right-holder	❖ A right-holder should be able to lock their distribution-fee of a song for a set time. (C) ❖ A right-holder should be able to lock their rights-fee of a song for a set time. (C)
Distributor, Node operator, Contract deployer and Validator	*Complete*

Table 9.6.1. Unimplemented functional requirements

As can be seen from this list, it is evident that most of the functional requirements have been implemented in the project. The Listener and Right-Holder components are the only ones that still have requirements that need to be fulfilled. These requirements are either *should-have, could-have,* or *won't-have* requirements. All *must-haves* have been implemented.

The reason why some *should-have* and *could-have* requirements of the Listener were not implemented is mostly because we ran out of time. Implementation of audio-streaming took up a lot more time than initially anticipated. This meant that we could not have delivered a

library page that was up to the standard quality of our application and we decided against implementing it.

The *could-have* Right-holder components were not implemented, mostly because we were not sure whether these would actually be an addition to the platform. Implementation would have added a lot of complexity, possibly entirely unnecessary complexity.

We had already decided beforehand that the *won't-have* requirements of the Listener fell outside the scope of the project. This is why they have not been implemented during the project. In the future, they could become a part of our project.

It can be concluded that only a small number of requirements have not been implemented, and that we managed to implement all of the *must-have*, most of the *should-have,* and a number of *could-have* requirements. All of these requirements are fully functional and have been tested thoroughly. This means that we have managed to implement a system that works and can actually be used by real users.

Non-functional requirements

In addition to the functional requirements, also defined non-functional requirements. The following list shows only those requirements that still need to be fulfilled.

Type	Non-functional requirement
Performance	❖ The creation of an account should take no longer than 1 minute. ❖ Skipping around in a song should start playing within 1 second.
Scalability	❖ The system should be able to scale to 1000 active listeners with 50 distributors.
Portability	❖ Listening should be possible on mobile devices. (Android/IOS)
Reliability	❖ Listening should be possible on mobile devices. (Android/IOS)
Security	❖ The system should allow for listeners to encrypt their streamed music.
Localization	❖ The system should display balance in any of the 10 most common currencies.
Usability	❖ Withdrawing money should be a couple of clicks and typing in the deposit-address.
Availability, compatibility and maintainability	*Complete*

Table 9.6.2. Unimplemented non-functional requirements

First, there are a number of requirements related to our systems performance that were not achieved. The performance is related to our systems' response time under different conditions. The creation of an account takes longer than one minute at the moment; due to the state of distributed ledger technologies, this is incredibly hard to reduce. Skipping a song also does not start playing within one second due to the delay of confirming a transaction on the smart contract.

Furthermore, we also were not able to test if our system is scalable, and managed to test our system with around 5 listeners and 4 distributors with 20 uploaded songs, whereas our requirement was to have a system that scales to 1000 listeners and 50 distributors. Another requirement that still needs to be implemented is that our mobile application for listening can be used on both Android and IOS. At the moment, users can only download the application on Android, but the creation of an IOS binary should not be a lot of work.

There was also a requirement regarding the reliability of the system. This requirement stated that the selection of a song should choose a distributor with a good connection 95% of the time. As of now, we have not started to implement this requirement. This will be a task for the future.

The last requirements that were not implemented regard the security, localization, and usability of the system. Firstly, encryption of music has not been implemented but should be a relatively easy modification. Secondly, the localization requirement states that the system should display the balance in any of the 10 most common currencies, while at the moment this is only displayed in MIOTA. Finally, the usability requirement states that the withdrawal of money should be doable in a couple of clicks and by typing in the deposit-address. This was impossible to implement into the listener application due to library incompatibilities with Flutter.

To conclude, even though we have already managed to implement a number of the non-functional requirements there is still work to be done. All requirements regarding the availability, compatibility, and maintainability of the system have all been implemented.

10. Discussion

This chapter highlights some of the most interesting aspects of our system and the struggles we experienced throughout our design and implementation. Each subsection presents a new aspect of the system and provides our experiences within this area.

10.1. IOTA Shimmer

As beta software, Shimmer presents several challenges, including non-descriptive error messages, rapidly changing tools, and compatibility issues with existing Ethereum software. Shimmer offers an Ethereum Virtual Machine implementation that closely resembles Ethereum, enabling the use of more mature Ethereum tools. However, due to minor differences in implementation, it does not always function as expected.

For example, Shimmer currently does not support out-of-order transactions. If a transaction with nonce 10 arrives before the one with nonce 9, the transaction is rejected. Developers have intentionally omitted this feature to simplify the implementation. Consequently, sending transactions in rapid succession can result in denied transactions, while waiting for confirmation can cause unacceptably long delays. To address this, we implemented a send-queue in the client that resends transactions using exponential backoff. Although this resolves the issue, it generates considerable overhead for both the client and the network, and complicates proper error handling.

While Shimmer offers promising potential for decentralized applications, its current beta status presents challenges that require workarounds or alternative solutions. As the platform matures, we hope that it further matures and refines the implementation, focusing on compatibility, error handling, and efficiency to ensure a more seamless integration with existing tools and systems.

10.2. Flutter and Just_Audio

Our design decision to use the Flutter framework had significant consequences for the implementation phase. Not only had none of us ever built a Flutter app or used its programming language Dart, but also the limited amount of libraries came into play. In order to play back audio to the user, we used a library called just_audio. Since one of the requirements was that the user pays per chunk, we had to overwrite the behavior of just_audio's buffering and seeking behavior which posed large challenges especially when it came to seeking within the song. A more low-level library would have been very useful but was not available.

10.3. Denial of service at smart contract

In the current implementation of the smart-contract, any party can register for distribution on the smart contract. There is a slight gas cost attached to the registration, but this is very minimal, and the distribution is stored on the smart contract forever. If there are many false distributors for a song, it becomes increasingly difficult for a listener to find a good distributor. This can be combated by disincentivizing anyone from registering for distribution of a given song.

The simplest solution to this problem is to make distributions temporary, such that after a certain amount of time the distribution becomes invalid and the distributor has to re-register for the distribution. This makes a denial of service attack costly to keep up, since a gas-fee is calculated for every registration.

An alternative solution is to increase the reputation, stored on the smart contract, whenever a payment is made to a distributor. That way, the longer people listen to a distributor, the better his reputation becomes. Users can now select a distributor based on their reputation. However, a big downside of this approach becomes scalability: Every transaction now needs to update more information on the smart contract.

It is even possible to implement both features combined to increase the protection against denial of service attacks, whilst simultaneously incentivizing distributors to deliver better performance. We have not been able to implement or test either solution due to time constraints.

10.4. Security concerns

Our system is based on a private/public key pair for authentication. This poses a risk that if the private key is lost or stolen, someone may access your account and the money it contains. To address this security concern, the listener application saves the private key with a user-chosen password through AES encryption. The distributor also saves the private key encrypted with AES through a password by default and only saves it as plaintext if the user specifically specifies to do so. Despite these measures, unknowing users could accidentally expose their private key if they fall for phishing or a similar attack.

Another concern is that the communication between the listener and distributor happens through an unencrypted tcp connection. This allows for eavesdroppers to see what song someone is listening to, and may also allow for spoofing attacks. The content of the TCP message that the listener sends is a smart contract transaction which is cryptographically signed and therefore not feasible to alter. This is important as the transaction does involve payment. Nonetheless, parameters such as what song and what chunk are requested could be altered and it is to be evaluated what spoofing attacks this could lead to.

10.5. Streaming architecture

Melero’s [2] original prototype used a streaming architecture in which the listener would send his payment-transaction to the smart-contract and wait for confirmation. Once the confirmation was received, the listener sends the transaction-id to the distributor, the distributor checks the validity of the transaction with the smart contract and finally sends back the music. This setup was good with regards to stability and security: We used practices that were standard with distributed ledger technology.

The main problem with this setup was the latency before receiving the first chunk of music. The distributor has to wait for the transaction to be confirmed, ping the distributor, the distributor then has to contact the smart-contract to check the transaction before finally

sending back the chunks. Under optimal conditions, this took around 3 seconds, but in a real-life scenario with longer round-trip-times this delay can become much larger.

Therefore, we chose another approach for this project: The listener creates and signs their payment-transactions locally and sends them to the distributor. The distributor then forwards these transactions to the smart contract and awaits their confirmation. This removes a single round-trip-time between listener and smart-contract. Combined with the debt-limit, we were able to reduce latency enormously.

However, this setup introduced a big problem: how do we manage nonces when signed transactions are sent to untrusted third parties? When a signed transaction is provided from listener to distributor, the distributor can do one of two things with it: Send it to the smart contract or throw it away. As a listener the only way to find out what happened with the transaction is to contact the smart contract. This provides information if the transaction was sent, but if it was not sent we have no idea whether the distributor will send it in the future.

This makes proper nonce-management on the client almost impossible when contacting multiple distributors at the same time. Fundamentally, it is impossible to contact multiple distributors simultaneously while guaranteeing that transactions arrive in-order at the smart contract. Combined with our challenges of using IOTA Shimmer (*See Discussion: IOTA Shimmer*) error handling in the listener became an impossible task. The best we could do was resetting the nonce whenever an error occurred with any kind of transaction.

After working on the project, we have become less confident in this approach. We think that it becomes too hard to properly handle errors and build a fault-tolerant application if signed transactions are sent to an untrusted party. For a future implementation we would be interested to see other approaches for reducing round-trip-time that do not involve the sending of signed transactions to untrusted parties.

It may be possible to implement such a system by creating multiple wallets for a single listener, this allows for simultaneous contacting of distributors. This adds a lot of complexity and overhead to every client, with a system that would still have a hard time properly synchronizing nonces.

11. Conclusion

In this section we will conclude what we have written in this report. It will include a short summary of the project, our achievements during the project, a comparison with similar works and future work regarding the project.

11.1. Summary

The primary goal of this project was to create a decentralized music streaming service that gives right-holders of music the chance to be fairly compensated for their work. Moreover, we had five design concerns that had to be respected throughout the design and implementation: Security, scalability, anonymity, legality and user incentivization. We aimed to build a system that aligns with these goals using the distributed ledger technology IOTA, and to provide valuable insights into this ledger's capabilities and possible limitations.

Our proposed solution consists of four distinct components, each essential for correct functioning of the platform. The mobile listener application is intended to allow users to listen to music using an intuitive and aesthetically-pleasing interface. The CLI for the distributor has been designed to allow anyone to distribute music, primarily focusing on ease of deployment. The validator component is a website used to validate uploaded songs and provide essential system-information. Finally, the smart contract binds the components together and acts as a decentralized backend for payments and storage.

We made use of the Scrum methodology to implement our software in an Agile way. Every week had a sprint with a specific goal. This allowed us to focus on the essential elements for that week, while keeping track of the requirements that still need to be implemented. All progress was tracked using Trello, code was hosted on GitHub and communication was done through Discord and WhatsApp.

11.2. Achievements

We believe that this project was executed with great success. We were able to organize the project into distinct phases, complete each phase within the expected timeframe and distribute the workload among all group members. As shown in the evaluation section of this report, we were able to meet our most critical requirements and deliver a functional product.

Our achievements include the creation of a user-friendly interface that simplifies the process of streaming and distributing music in a distributed manner along with the development of a validator system to secure intellectual rights on the platform. Additionally, we have explored and employed innovative approaches to enhance scalability and optimize distributed music streaming, ultimately contributing to a more robust and reliable platform.

Throughout the project, we demonstrated the value and potential of the IOTA distributed ledger in developing decentralized solutions for the music industry. We managed to design and implement a platform that effectively addresses the challenges outlined in our problem statement and fulfills our initial goals.

11.3. Future work

There is a substantial amount of future work remaining, both in terms of concrete improvements to our implementation and broader research questions that remain unanswered. The main areas requiring further research include our streaming protocol, regulations regarding intellectual property rights, and performance testing of the system.

Concrete improvements

Some of the most crucial and straightforward improvements include:

- Expanding the custom TCP protocol to use error codes instead of merely closing the connection. This would provide better error handling and feedback to the user, helping to identify and resolve issues more efficiently.
- Adding a feature that enables users to like songs in the listening client. This would improve user experience immediately by allowing them faster access to their favorite music.
- Optimizing distributor selection on the client-side of the listener based on the location of its IP address, stored lists of good and bad distributors, and distribution fee. This would lead to faster and more reliable connections for streaming, especially once bad actors join the network.
- Disincentivization of song-distribution when not actually distributing. This would help to combat the denial of service attacks outlined in the discussion.

Research

- As noted in the discussion, the streaming protocol we chose has several downsides. Further research is needed to identify suitable alternatives that allow for low-latency, concurrent, and secure streaming between listeners and distributors.
- Concerning the protection of intellectual rights, further research by legal experts in intellectual property rights could help refine the platform's approach to rights management. None of the authors of this report have any legal expertise, therefore to ensure that the system is compliant with relevant laws and regulations, more research is required.
- We have provided a system that works on a smaller scale, using a handful of listeners and distributors. However, it remains unclear how well the system scales as more listeners or distributors join the network, and as more songs are uploaded. Performance testing is necessary to determine how the scaling of users affects performance metrics such as latency and throughput.

12. References

[1] J. Dimont, “Royalty Inequity: Why Music Streaming Services Should Switch to a Per-Subscriber Model,” *Hastings Law Journal*, vol. 69, no. 2, p. 675, Feb. 2018, Available: https://repository.uchastings.edu/hastings_law_journal/vol69/iss2/5/

[2] D. Melero Martinez, “IOTA-MSS : a pay-per-play music streaming system based on IOTA,” *essay.utwente.nl*, Feb. 03, 2023. http://essay.utwente.nl/94343/

[3] Musicoin, “Musicoin Project White Paper V2.0,” *Medium*, Jul. 07, 2019. https://musicoin.medium.com/musicoin-project-white-paper-v2-0-6be5fd53191b

[4] “eMusic Redefining Music Distribution Through Blockchain,” *eMusic*, Feb. 04, 2019. https://token.emusic.com/assets/pdf/eMusic_White_Paper_EN.pdf

[5] E. Drąsutis, “IOTA Smart Contracts,” 2021. Available: https://files.iota.org/papers/ISC_WP_Nov_10_2021.pdf

13. Appendices

13.1. Project Proposal

Background

Since 2017, Music Streaming Services (MSS) have become the foremost contributor to the global recorded music industry's revenue. Many argue that the service-centric model used by these services to decide the musician's revenue do not adequately serve the industry as the average pay per stream rounds the 0,004$. In this market, MSS's have a big power-advantage due to their oligopoly, and it is almost impossible for an independent producer to get paid well for their music.

The main objective is to make the music streaming market more competitive and give back power to the (independent) music producers; anyone should be able to record a song, set a price for it and make it available for streaming anywhere in the world. There should not be a single company with rights to change the terms-of-service. Instead the distribution-rights and payment should be organized in a decentralized way.

Objectives

1. To create and deploy a decentralized music streaming service on the IOTA distributed ledger where anyone can upload, distribute and stream songs. *(Smart contract)*
2. To create a music streaming application for mobile with a simple-to-use GUI. *(Listening client)*
3. To create a music distribution client for the desktop. *(Distributing client)*
4. To create and deploy a validator with a basic web-interface that allows anyone to upload songs. *(Validator server)*

Scope

A music streaming service will be run on the IOTA distributed ledger as a smart-contract deployed on multiple nodes around the world. This is the trusted third-party that connects listeners with distributors and allows them the distributor to be paid for the music it provides.

For listening to music, a music streaming application is provided for mobile. It allows any (non-technological) user to stream songs from the distributors. The client has a nice GUI that allows the user to select a song to listen to, and the user is able to pay automatically for this music after sending funds to his account's IOTA-address.

Distribution of music will be possible through a command line interface for the desktop, which allows anyone with a little bit of technological background to start distributing music. It allows the user to automatically download and distribute songs other users have uploaded. We will initially deploy multiple distributors that will provide copyright-free music for free. The mobile application can then be used to test the system.

In addition we will run a very simple validator that any user can use to upload their music. All music requested will have to be validated manually by an administrator, after which the music is automatically registered on the smart contract.

Timeframe

Phase	Task	Dates
Design Phase	Make design documents concerning all important parts of the project: - Detailed planning - Functional and Quality requirements - Test plan - Risk analysis - Diagrams concerning core aspects of the system	Week 1, 2, 3 6 February - 24 February

Prototype Phase	Start prototyping all aspects of the system, mainly the following three parts. - Listening client - Distributing client - Smart contract While the system should function like outlined above in objectives, there may be bugs or usability issues.	Week (4), 5, 6 27 February - 17 March
Build Phase	Continue work on the prototypes focusing on the following aspects: - Improving usability for all clients - Unit and integration testing of the system - Initial deployment of the smart-contract to a few servers around the world In addition, a simple validator will be built and deployed.	Week 7, 8, 9 20 March - 7 April
Release Phase	Deploy a system which allows anyone with the distributing or listening client to connect to the network. Every part of the system should work according to the objectives outlined above. In the last week the final poster-presentation will be presented and handed in alongside the final design-report.	Week 10, 11 10 April - 21 April

Table 13.1.1. Timeframe

Project Budget

To run a real-world scenario for a month, we would need to deploy the following services:

- 3 distributors located around Europe distributing copyright-free music. We estimate that each distributor requires 1 CPU core with 2GB RAM and 40GB SSD storage.
- 3 IOTA nodes located around Europe running the smart contract. We estimate that each node requires 2 CPU cores with 4 GB RAM and 40 GB SSD drive storage.
- A single validator.

Deployment to a service like DigitalOcean or Amazon AWS is preferred, with an estimated monthly cost of around 100 euros to run the distributors, IOTA nodes and validator. Alternatively we can run the nodes locally on Raspberry Pi's spread around Enschede, which would not cost any money.

Key Stakeholders

Internal Stakeholders

Client / Supervisor	Mohammed Elhajj
Developers	Evana Reuvers, Daniel Melero, Jasper van der Werf, Jelte Koorstra, Paul Blum

Table 13.1.2. Internal stakeholders

External Stakeholders

Listener	The user who wants to listen to music
Distributor	The user who distributes the music to listeners
Right-holder	The user who registers music on the platform

Validator	The entity that assesses whether a song that is to be added to the system satisfies the requirements of the platform.

Table 13.1.3. External stakeholders

Monitoring and Evaluation

The client is Mohammed Elhajj. Throughout all four phases we will meet weekly with the client and supervisor. During the design phase, we will work on the design and planning components of the project and monitor if these satisfy the requirements. In the prototype phase we will work on prototyping the three components of the system, we will continuously work towards satisfying the functional and non-functional requirements. During the build phase we will start testing the software to see if it satisfies the requirements and the needs of the client. In the release phase the software will be deployed and tested in the real world.

Planned deliverables

Over the course of this project we will deliver the following:

- Project Proposal: **17 february**
- Informal Presentation: **19 April**
- Design Report + product delivery: **21 april**
- Poster presentation + hand-in: **To be planned in week 9/10**

Risk analysis

The risks that could affect the success of this project include the following factors.

- The system requires a sufficient number of distributors because of the nature of a P2P network. This will make both testing the system on a large scale harder as well as the success that this solution can achieve in the real world.
- Copyright infringements are to be avoided at any cost. This risk can legally affect the solution and will most likely be addressed through a centralized validation system.

Description of the project organization (responsibilities, procedures)

All members of the team are expected to contribute about equally to the project work. A Google Drive folder and a git organization will be the main place of collaboration, they will be shared with the supervisor. Every two weeks a project manager will be selected that tracks the progress, schedules the meetings with the supervisor and organizes the daily meetings. After every meeting with the supervisor, the team meets and plans the next sprint, reflected in a Trello board. Every team member is expected to be project manager once. Work is to be done mainly in-person and alternatively online. The green and red card procedure will be followed if these cases arise.

13.2. Wireframes

Reference: Figma design of TangleTunes

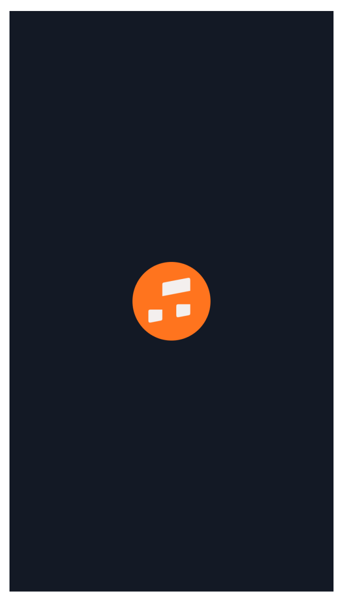


Figure 13.2.1. Wireframe of the splash screen

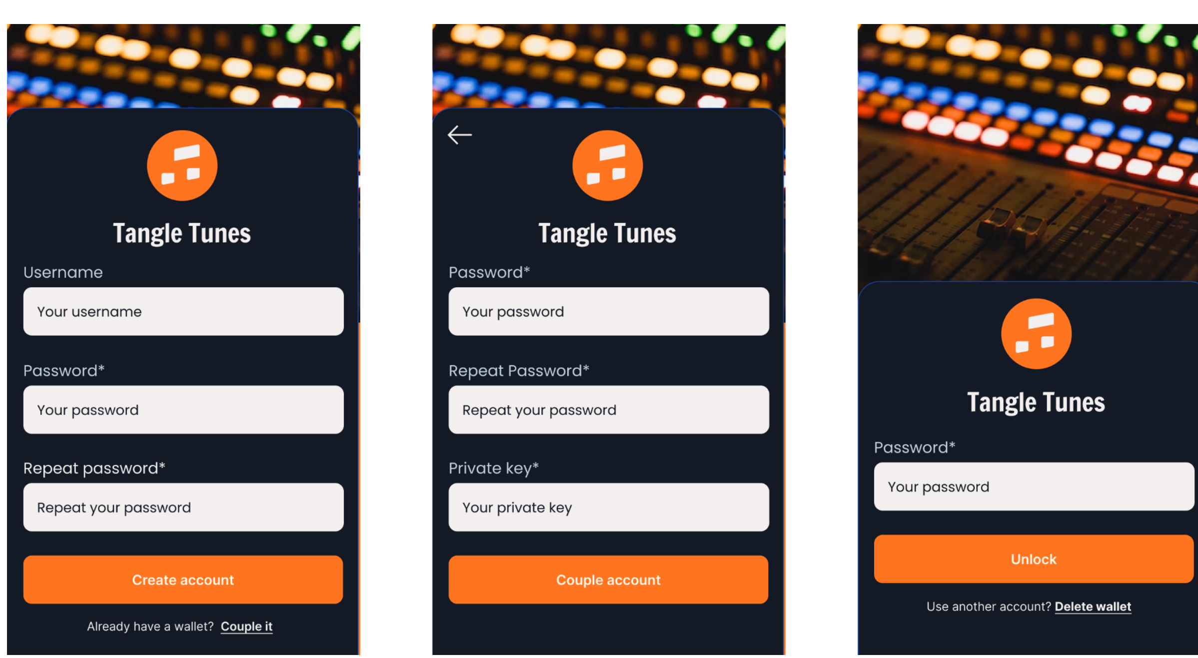
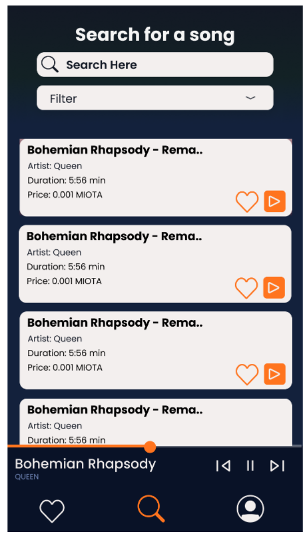
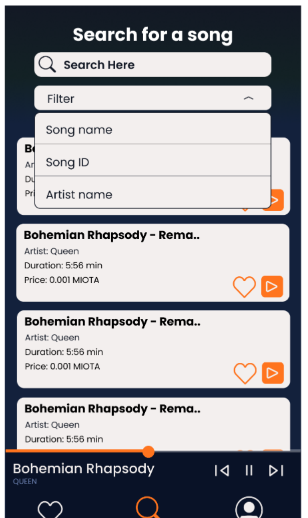


Figure 13.2.2. Wireframes of the login page, register page and unlock account page





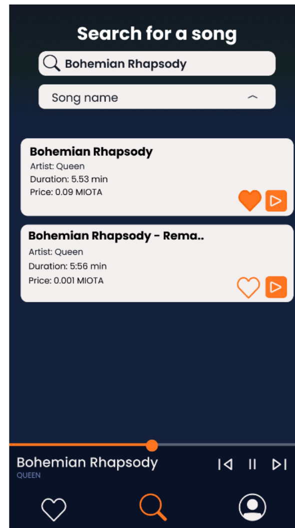
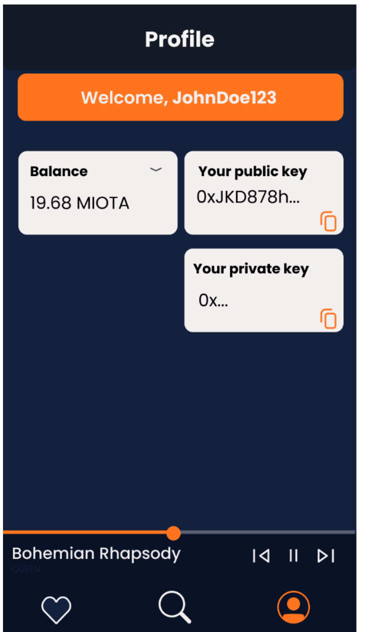
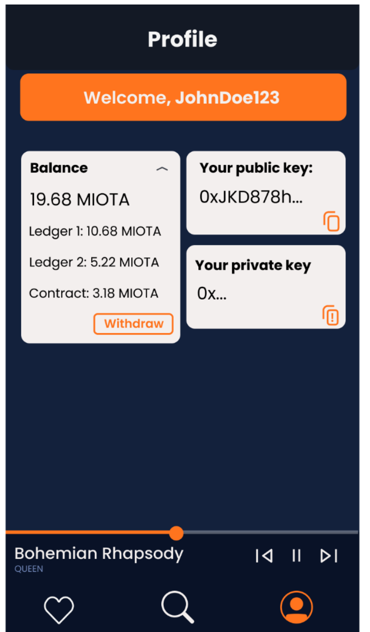


Figure 13.2.3. Wireframes of the discovery page





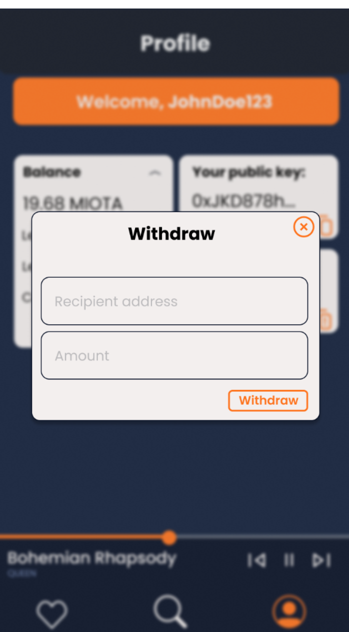


Figure 13.2.4. Wireframes of the account page

13.3. Listener User Interface

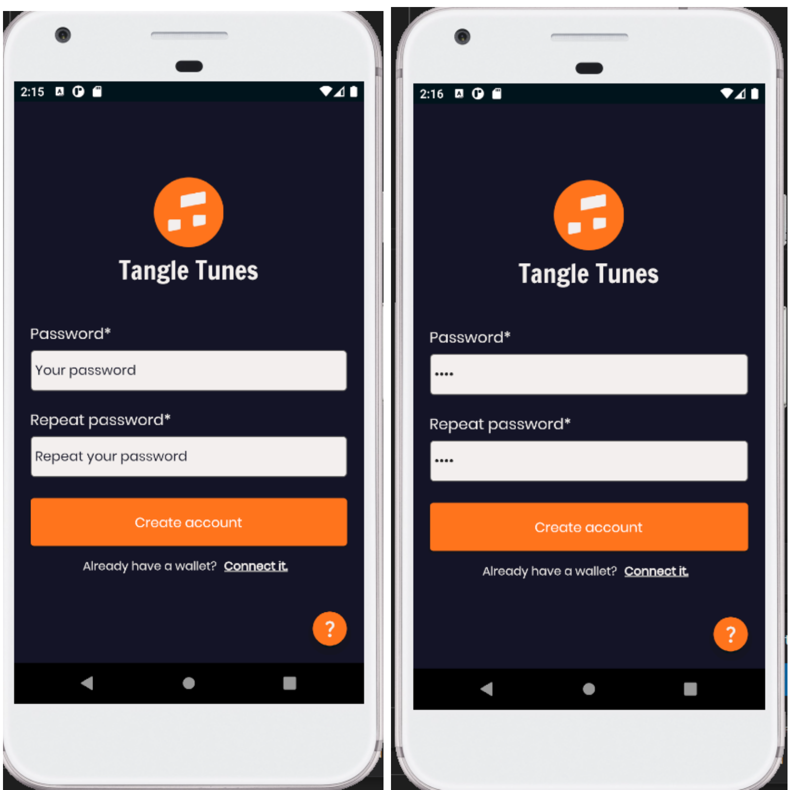


Figure 13.3.2. Screenshots of the create account page - enter password

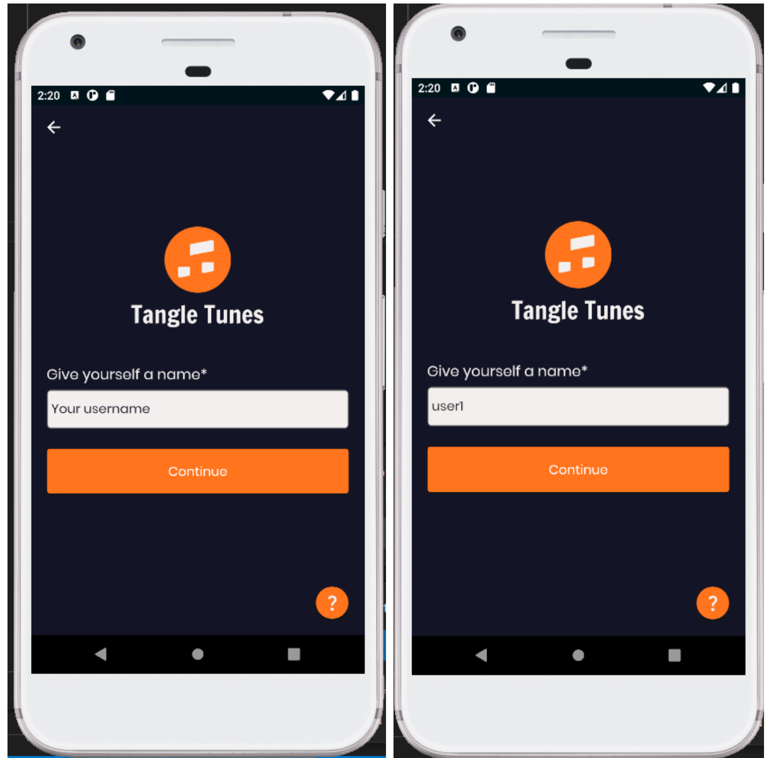


Figure 13.3.4. Screenshots of the create account page - enter username

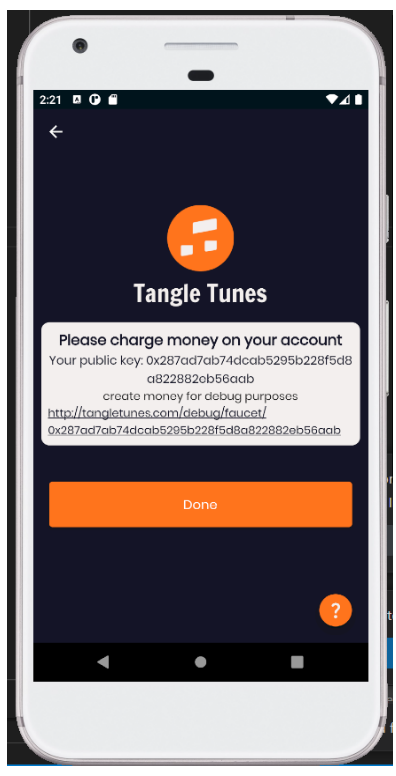


Figure 13.3.5. Screenshots of the create account page - charge money to account

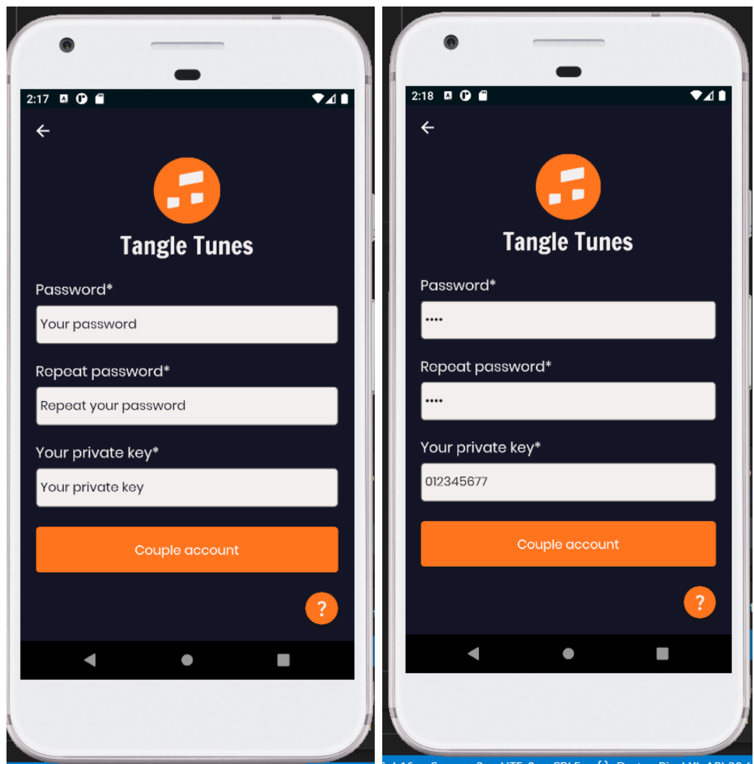


Figure 13.3.7. Screenshots of the couple account page

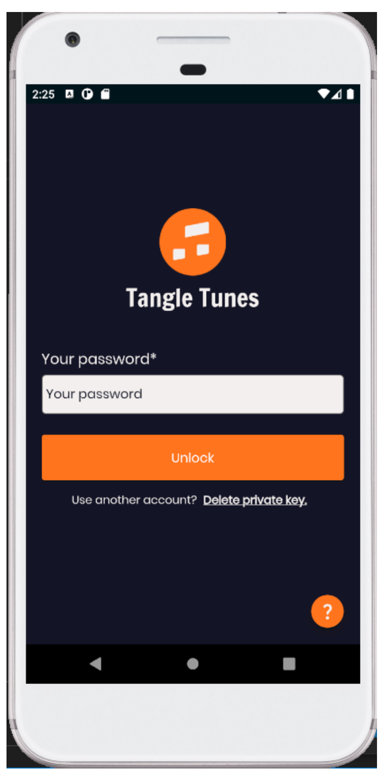


Figure 13.3.8. Screenshots of the unlock account page

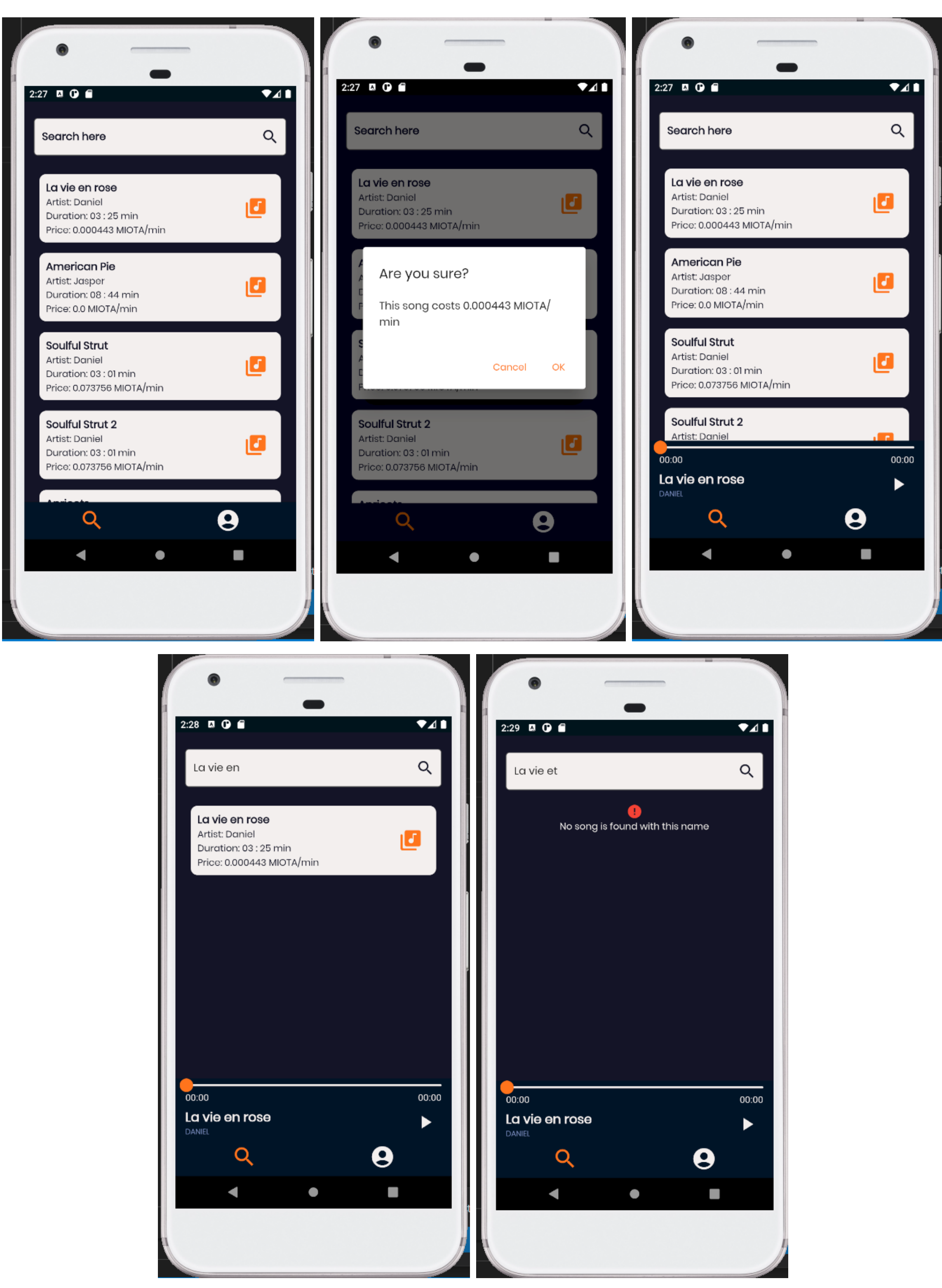


Figure 13.3.13. Screenshots of the discovery page

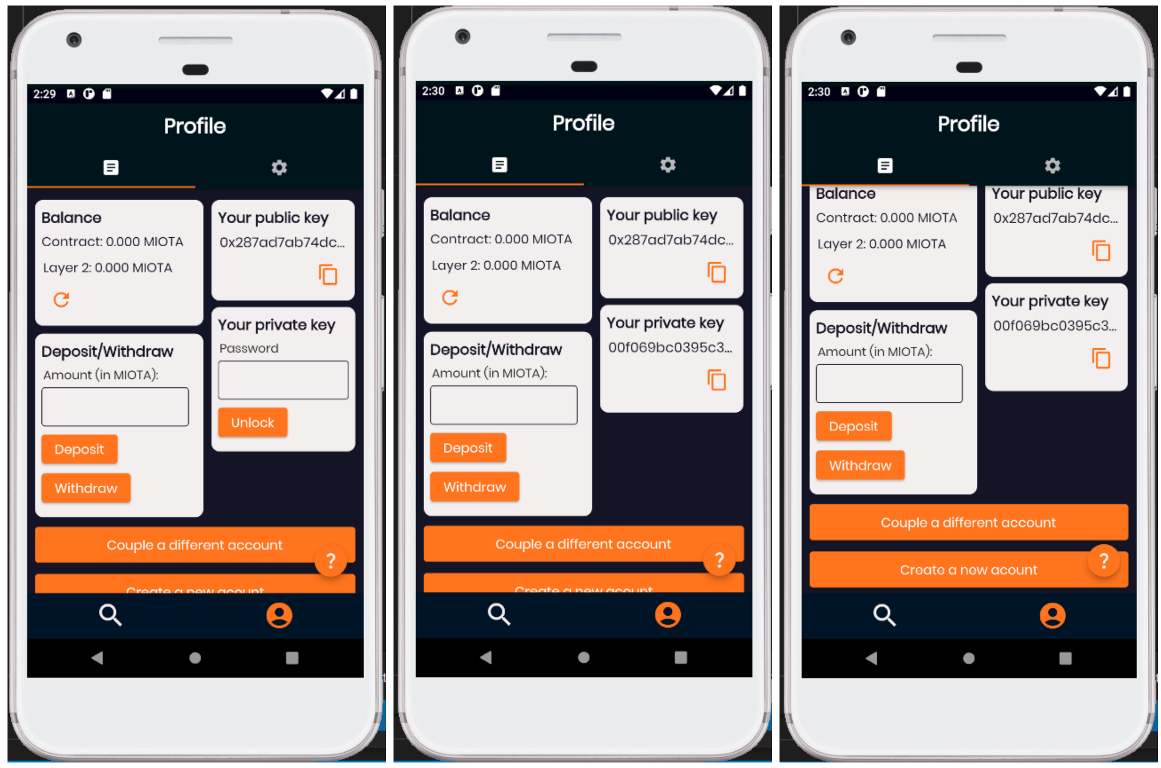


Figure 13.3.16. Screenshots of the account page - personal details

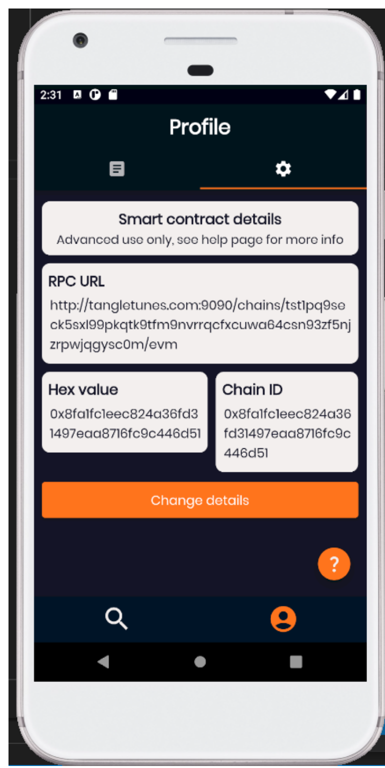


Figure 13.3.17. Screenshots of the account page - smart contract details

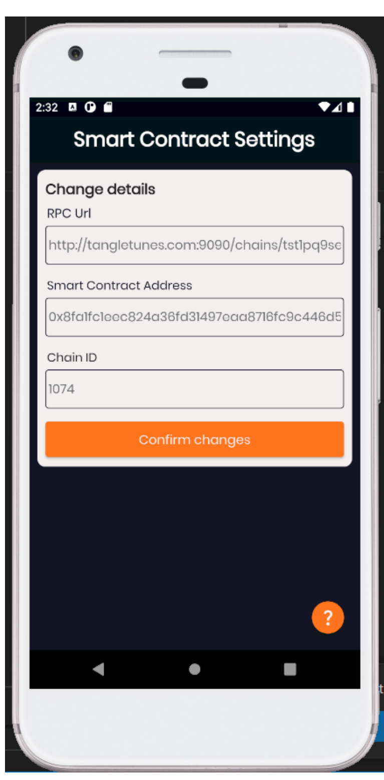


Figure 13.3.18. Screenshots of the change smart contract details page

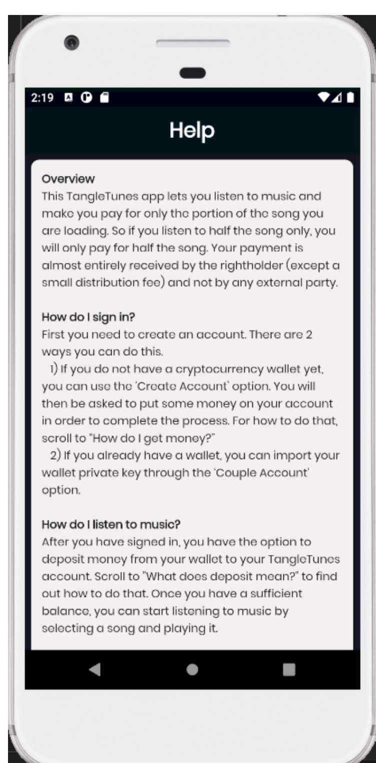


Figure 13.3.19. Screenshots of the help page

13.4. Validator website UI

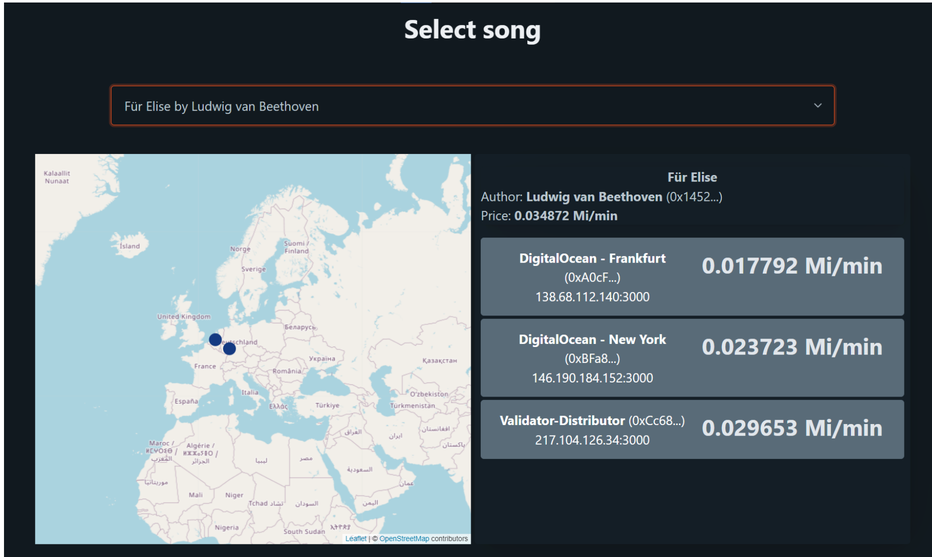


Figure 13.4.1. Screenshot of the distributor map

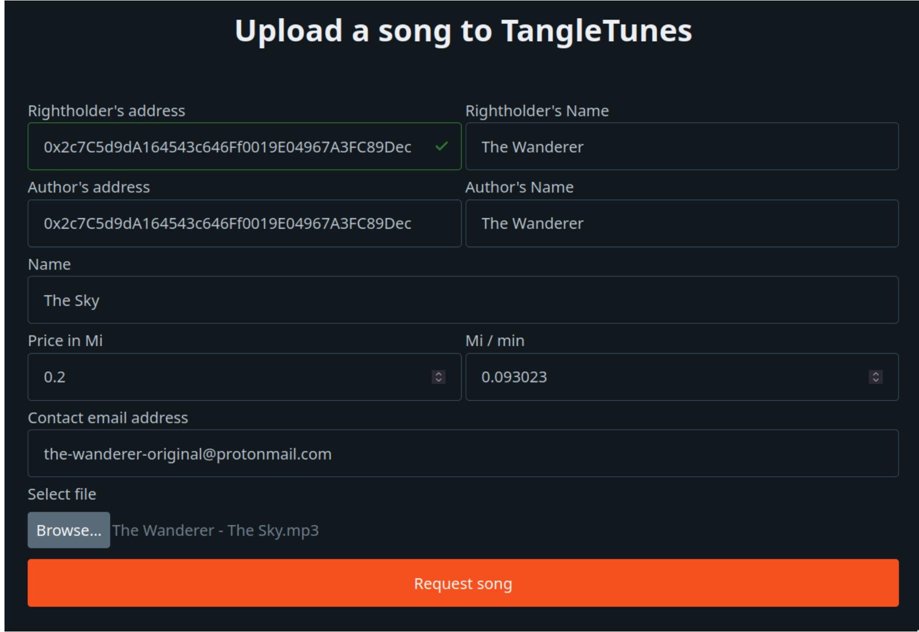


Figure 13.4.2. Screenshot of song upload page

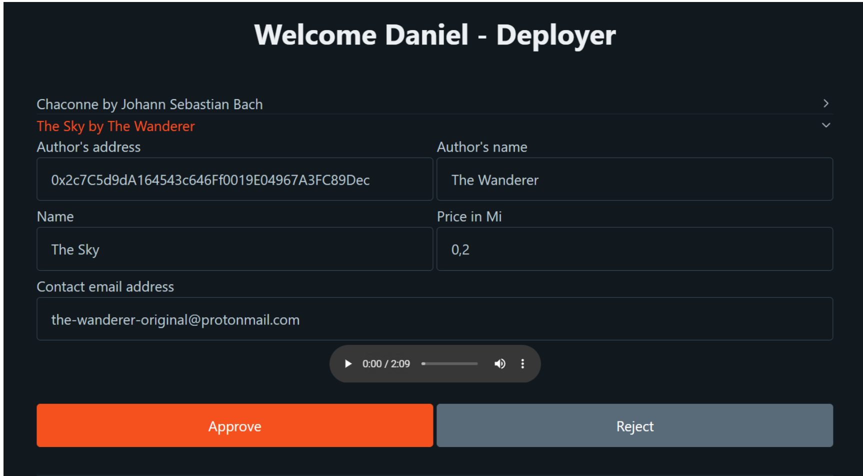


Figure 13.4.3. Screenshot of validate song page

13.5. Distributor Commands and Configuration

```
● ➜ tt-distributor --help
A distribution client for TangleTunes

Usage: tt-distributor [OPTIONS] <COMMAND>

Commands:
  wallet      Manage your IOTA wallet
  account     Manage your TangleTunes account
  songs       Manage your downloaded songs
  song-index  Manage the local copy of the song-index
  distribute  Start distributing
  help        Print this message or the help of the given subcommand(s)

Options:
  -c, --config <CONFIG>      The path to the configuration file [default: ./TangleTunes.toml]
  -p, --password <PASSWORD>  The password for encryption of private key
  -h, --help                 Print help
  -V, --version              Print version
```

Figure 13.5.1. Main command

```
● ➜ tt-distributor wallet --help
Manage your IOTA wallet

Usage: tt-distributor wallet [OPTIONS] <COMMAND>

Commands:
  import         Import a wallet with the given private key
  generate       Generate a new wallet with randomized private key
  remove         Remove the current wallet
  address        Export the address
  private-key    Export the private key
  balance        Display the balance of your wallet (L2)
  request-funds  Request funds from the test-network
  help           Print this message or the help of the given subcommand(s)

Options:
  -c, --config <CONFIG>      The path to the configuration file [default: ./TangleTunes.toml]
  -p, --password <PASSWORD>  The password for encryption of private key
  -h, --help                 Print help
```

Figure 13.5.2. Wallet subcommand

```
● ➜ tt-distributor account --help
Manage your TangleTunes account

Usage: tt-distributor account [OPTIONS] <COMMAND>

Commands:
  deposit   Deposit into your account from your wallet
  withdraw  Withdraw from your account into your wallet
  create    Create a new account coupled to your wallet
  delete    Delete the account coupled to your wallet
  view      View your account details
  help      Print this message or the help of the given subcommand(s)

Options:
  -c, --config <CONFIG>      The path to the configuration file [default: ./TangleTunes.toml]
  -p, --password <PASSWORD>  The password for encryption of private key
  -h, --help                 Print help
```

Figure 13.5.3. Account subcommand

```
● ➜ tt-distributor song-index --help
Manage the local copy of the song-index

Usage: tt-distributor song-index [OPTIONS] <COMMAND>

Commands:
  update    Update the list of songs from the smart-contract
  reset     Reset the list of songs from the smart-contract
  list      List all songs
  download  Download indexed songs from another distributor
  help      Print this message or the help of the given subcommand(s)

Options:
  -c, --config <CONFIG>      The path to the configuration file [default: ./TangleTunes.toml]
  -p, --password <PASSWORD>  The password for encryption of private key
  -h, --help                 Print help
```

Figure 13.5.4. Song Index subcommand

```
● ➜ tt-distributor distribute --help
  Start distributing

  Usage: tt-distributor distribute [OPTIONS]

  Options:
          --auto-download        Automatically download and distribute songs from other distributors
    -c, --config <CONFIG>      The path to the configuration file [default: ./TangleTunes.toml]
    -p, --password <PASSWORD>  The password for encryption of private key
    -h, --help                 Print help
```

Figure 13.5.5. Distribute subcommand

```
# The address registered on the smart contract
server_address = "183.23.5.32:7483"
# The address that the tcp-listener should bind on
bind_address = "127.0.0.1:3000"
# The path to the database relative to this file
database_path = "./target/database"

# The fee per chunk in IOTA
fee = 100
# The max price per chunk in IOTA
max_price = 100_000

# Smart-contract details
chain_id = 1074
contract_address = "0xC009692aeF725817bba53B919a7e57f3062076eA"
node_url = "http://tangletunes.com:9090/chains/tst1pq9seck5sxl99pkqtk9tfm9nvrrqcfxcuwfs64csn93zf5njzrpwjqgysc0m/evm"
```

Figure 13.5.6. Example configuration file

13.6. Test results

```
running 12 tests
test command::distribute::background_tasks::test::song_queue ... ok
test library::client::download::test::request_queue_test ... ok
test library::client::download::test::song_is_complete_test ... ok
test library::client::test::get_songs ... ok
test library::crypto::test::encrypt_decrypt_correct ... ok
test library::crypto::test::encrypt_decrypt_incorrect ... ok
test library::crypto::test::generating_and_importing ... ok
test library::database::test::add_remove_song ... ok
test library::database::test::chunking_is_correct ... ok
test library::database::test::set_get_key ... ok
test library::database::test::song_index ... ok
test library::database::test::song_metadata ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.18s
```

Figure 13.6.1. Distributor test results

```
> smart-contract@1.0.0 test
> npx hardhat test



  Account Management
    ✓ User should be able to create account (3841ms)
    ✓ User should be able to edit description (172ms)
    ✓ User should be able to edit server info (150ms)
    ✓ User should be able to remove account (156ms)
    ✓ User should be able to deposit (123ms)
    ✓ User should be able to withdraw to chain (141ms)
    1) User should be able to withdraw to tangle

  Distribution Management
    ✓ Distributor should be able to distribute a song (459ms)
    ✓ Multiple distributors should be able to distribute a song (290ms)
    ✓ Distributor should be able to decrease fee (392ms)
    ✓ Distributor should be able to increase fee (334ms)
    ✓ Distributor should be able to undistribute song (565ms)
    ✓ Remove all distributions when song is directly deleted (228ms)
    ✓ Remove all distributions when song is indirectly deleted (189ms)
    ✓ User can get chunks from a distributor (485ms)

  Song Management
    ✓ Validator should be able to upload a song (292ms)
    ✓ Rightholder should be able to change the price of a song (98ms)
    ✓ Rightholder should be able to delete their songs (163ms)
    ✓ Validator should be able to delete their songs (64ms)
    ✓ Song deletion when validator is dismissed (77ms)
    ✓ Song deletion when Author deletes their account (107ms)
    ✓ Song deletion when Rightholder deletes their account (127ms)
    ✓ Song deletion when Validator deletes their account (110ms)

  Validator Management
    ✓ Deployer should be able to assign a validator (223ms)
    ✓ Deployer should be able to dismiss a validator (58ms)


  24 passing (9s)
  1 failing
```

Figure 13.6.2. Smart contract test results

```
PS C:\Users\Jelte\Documents\GitHub\listener> flutter test test/file_writer_test.dart -r  expanded
00:00 +0: (setUpAll)
00:00 +0: writeToFile
C:\Users\Jelte\Documents\GitHub\listener/test
test.txt now contains test
00:00 +1: (tearDownAll)
00:00 +1: All tests passed!
```

Figure 13.6.3. Listener unit test results

```
PS C:\Users\Jelte\Documents\GitHub\listener> flutter test .\test\price_conversions_test.dart -r  expanded
00:00 +0: weiToMiota
00:00 +1: miotaToWei
00:00 +2: priceInMiotaPerMinute
00:00 +3: All tests passed!
```

Figure 13.6.4. Listener unit test results

13.7. Contribution log

	Jasper van der Werf	Paul Blum	Jelte Koornstra	Daniel Melero	Evana Reuvers
Week 1	- Got to know team members and talked about the project. - Read up on IOTA and distributed ledgers.	- getting to know the team - read & understand about the distributed ledger technology IOTA - discussions with team members - define scope of this project - define functional & non-functional requirements	- Meeting the team - Read up on IOTA and Flutter - Creating requirements according to MoSCoW	- Introduction to the team - Presenting the project - Produced list of resources to learn more about IOTA - Defining requirements	- Getting to know the team - Read up on IOTA and Flutter - Defining scope of the project - Defining functional and non-functional requirements
Week 2	- Started initial work on the distributor client in Rust. It can connect to an IOTA network and do some basic TCP stuff. - Write a proposal plan and functional/non-functio nal requirements.	- Project Proposal contributions - setting up flutter environment & emulators - setting up git for listener client - learning about programming language Dart - start development of a music player that works with a stream of bytes - user interface for a basic music player with slider	- Setting up VSCode for the flutter environment - Setting up github - Fixing issue with emulators - Learning dart - Working on the listener client that works with bytestreams - Working on the project proposal stakeholder and monitoring / evaluation sections	- Set up testing environment for the smart contract - Defined MoSCoW prioritization - Organized brainstorming session - Set up Private Tangle with the latest versions of the IOTA nodes - Wrote project proposal	- Setting up flutter environment in VScode and emulators - Working with Flutter and Dart - Start development of a music player that works with a stream of bytes (With Jelte & Paul) - Starting tutorials on how to build a good UI in Flutter - Writing on the project proposal
Week 3	- Continued work on the distributor client. It can now manage an encrypted private key in an SQLite database. - Created the 4 sequence diagrams. - Helped with the UI design in Flutter and with the Mediaplayer in the Listening application. - Finished up requirements from last week. - Create distributor activity diagrams.	- activity diagram for listener client - implementation of buffering audio only up to certain duration in flutter - implementation of caching audio locally in flutter - slides for supervisor meeting	- Working on use case diagram for listener and distributor - Setting up the hornet node on local device	- Wrote Smart Contract documentation - Made Extended Entity Relationship (EER) diagram - Implemented smart contract's basic functionality (MVP ready) - Implemented automated tests for existing functionality in the Smart Contract	- Made an activity diagram for the listener client - Finished all the UI designs for the application in Figma - Wrote about the activity diagram for the listener client in the report - Made slides for the presentation
(Week 4)	Vacation	Vacation & Illness	- Connecting the listener to the smart contract (it is now possible to send transactions to the smart contract)	- Created nodejs web server capable of interacting with the smart contract	Created the UI for the : - Loading screen - Register account screen - Couple account screen - Unlock account screen

Week 5	- The distributor can now download a song from another distributor. It encodes and decodes the tx, checks its validity and confirms it with the smart-contract. - Designed the tcp-protocol for streaming a song.	- implementation of requesting chunk by TCP connection (not working yet) - understanding TCP response from distributor and trying to play it as a song	-Implement all the smart contract functions that the listener needs - Setting up the distributor locally and working on the connection between listener and distributor	- Implemented automatic chain generation as well as persistent storage for validator web app using docker container - Implemented: A validator should be able to register songs for any user	Created the UI for the discovery page, including: - a bottom navigation bar - search bar that can search a list of songs - slider + music player (without functionality) Started on the UI of the library page
Week 6	- Continued work on the distributor, fixing many bugs and making the user experience better. Pretty much everything works now as intended. - Deployed the distributor to a Raspberry Pi	- listener can now fetch songs from distributor - logic to handle with private key and save it securely on device - large progress in flutter and making app state retain - merging codebases for user interface and simple music player	- Implement state transferring across different pages - Merge UI and listener codebases - Worked on being able to request chunks in larger batches from the distributor	- Set up Raspberry pi in my room and configured router to forward required ports - Deployed tangle and chain node on the Raspberry pi to allow everyone to interact with a single smart contract - Implemented authentication logic based on in-browser wallet for the validator's web server - Deployed validator web application on Raspberry pi	Did not contribute because Grandma suddenly fell ill and passed away.
Week 7	- Split the distributor commands into separate subcommands, account management easier now. - Password encryption for distributor	- accounts page now shows balance, public key and allows to change private - you can now view and edit the smart contract RPC URL, chain ID and address in the app - pages in the app now redirect to loading screens if appropriate data is not loaded - merged UI like play button and seek bar - now fingerprint/pin authentication required to delete private key on unlock page	- balance is shown on the account page and added a deposit function - Implement loading screens on the transition pages - Fix the bug where skipping into an unbuffered part of audio also buffered the audio in between - Improved error catching in the app	- A right-holder should be able to request registration at a validator for their music. - Validator should distribute uploaded songs - Upgraded smart to manage distributors based on their fee	- worked on the account page UI, the first tab in it is almost done and functional - improved consistency in UI throughout the app - added loading screens that continue automatically when data is loaded instead of buttons - fixed some bugs in the UI - removed the library page from the app

Week 8	- Distributors can automatically download new songs from other distributors. - Work on final report: Component Design for the Distributor. - Distributor can now request funds through the faucet automatically	- also show L2 balance on on account page - user is asked to please deposit money when creating an account on the smart contract fails - help page in listener app - warning on that song costs money when clicking on it - withdraw function in account page - testing of listener app and reporting those tests in the Design report	-the user is asked to please deposit money when creating an account on the smart contract fails - help page in listener app - confirmation dialog before listening to a song - withdraw function in account page - describing the manual testing process of the listener application in the design report	- Finished implementing smart contract's automated testing - Fixed major bugs on the validator's website - Started writing design section in the report about Validator and Smart Contract - First drafts of the poster	- finished the UI of the account page - made the basic UI of the help page - Created the final layout of the report - wrote Domain Analysis in the final report - wrote Development Methodology in the final report
Week 9	- Write a design report: Introduction, and System Architecture. - CI/CD: Distributor releases created automatically and tests are run automatically. - Class diagram for SQlite database.	- write parts of Design report: listener technologies, achievements, discussion of just_audio and flutter	- Write unit tests for the listener - Finish section about unit testing in the design report - Work on use case diagram	- Implemented song browsing page in the website to show distributors in a map - Finished Poster design - Started writing report about validator and smart contract components	Report: - wrote Requirements specification and Analysis chapter - wrote application flow and UI design sections of the Listener in component design chapter - improved the Activity Diagram of the Listener - wrote Listener GUI section in Product chapter - wrote Terminology in the Domain Analysis - add screenshots of the wireframes in the Appendix
Week 10	- Work on final presentation. - Deployment of distributor on DigitalOcean Instances. - Continued work on report: Made parts better and created activity diagrams for the distributor.	- presentation slides & rehearsal - write part about implementation of listener: CI/CD	- Finish use case diagram - Finish the section about the use case diagram in the design report - Make slides for stakeholders and goals in the presentation	- Finished Poster contents (paragraphs and diagrams) - Started writing report about testing on the validator and smart contract - Worked on the final presentation	Report: - wrote Evaluation in Testing and evaluation chapter - wrote the Summary section of the Conclusion chapter - add screenshots of the application in the Appendix Wrote final presentation slides
Week 11	- Work on the design report, mostly working on discussion/future work. - Fix distributor's final problems to create the last release - Unit testing for distributor	- write parts of the Design Report: security concerns, Introduction to testing and evaluation	- In the design report, write about some technologies that were used in the development of the listener - In the design report, write about system testing	- Designed and print flyers - Final touches to the website's map of distributors - Finished sections about validator and smart contract	Report: - write the evaluation of non-functional requirements in chapter Testing and Evaluation - add an explaining paragraph for component design

Table 13.7.1. Contribution log