Código Hub: IoT + ETH <3

06.08.2020

Overview

Código is a firmware distribution service with an IoT twist. Developed conceptually as a research project at the University of Edinburgh. We're bringing the idea into the twenty-twenties by rebuilding the network framework as a DApp.

Built to act as a shop-window for users to manage their IoT devices, the platform works to incentivize developers to build, maintain and contribute firmware to be securely deployed across a host of devices in the IoT domain. Designed as a system with no single point of failure, Código is helping to solve one of the biggest challenges facing the uptake of this domain - security.

Key Features

- Contribute firmware to be securely deployed across a host of IoT devices.
- Decentralized infrastructure with no single point of failure.
- Incentivization scheme which rewards developers for producing firmware that users want.

Motivation

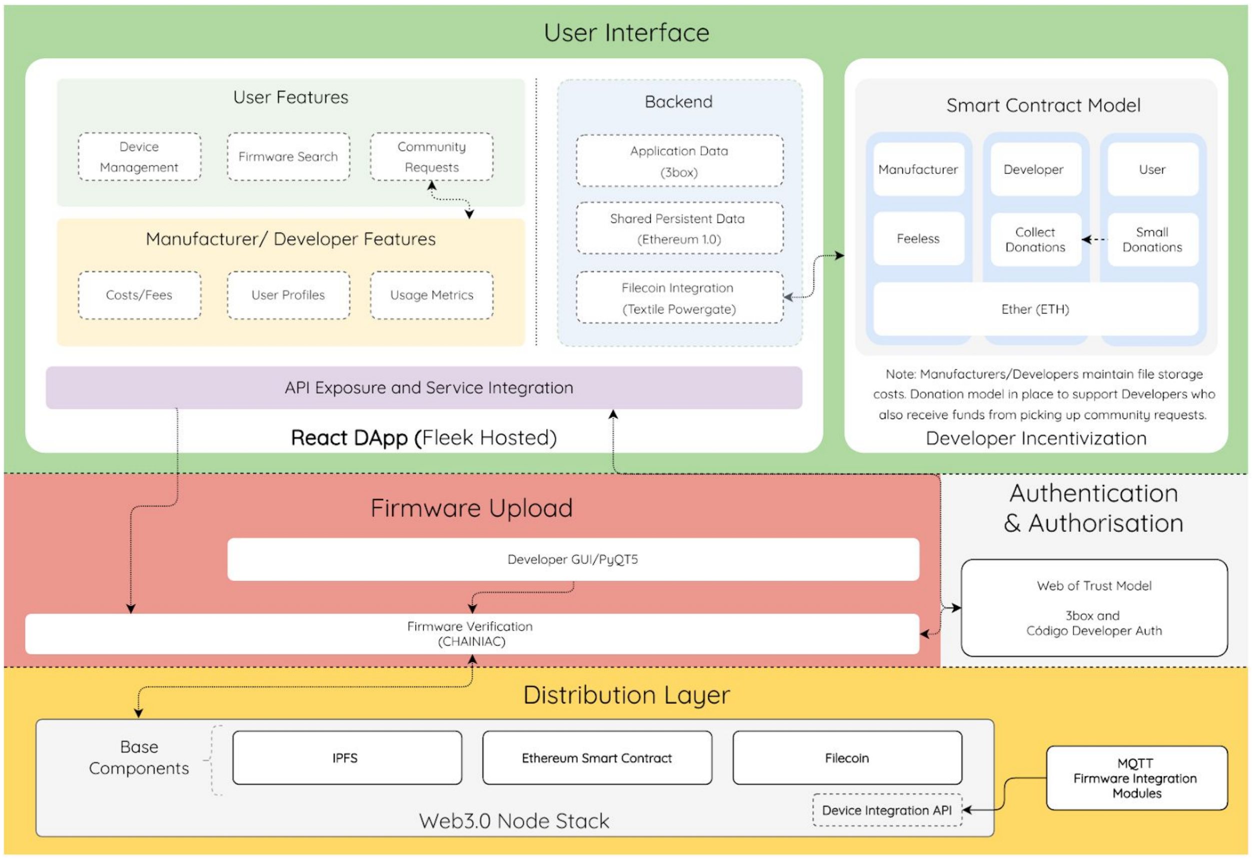
The existing work on the distribution service was built using Ethereum and IPFS, but our intention is only to interface with the underlying system (with small changes as needed) and essentially build an entirely separate, stand-alone Dapp to work in parallel to the network. To help make the judging process easier we're planning to make a clear distinction between the works by utilizing distinct repos.

The network was spec'd as part of a research project and an open-source codebase was developed (which provides secure firmware distribution, and network testing) - but there is no work done for the aspects we'd intend to contribute focusing on both developer incentivization and wider user-engagement.

Goals

1. Provide an end-to-end system design for how Código will work in a real-world environment
2. Design a suitable incentivization model for encouraging community engagement
3. Develop a fully functional Código UI

System Specifications


User Interface
User Features
Device Management
Firmware Search
Community Requests
Manufacturer/ Developer Features
Costs/Fees
User Profiles
Usage Metrics
Backend
Application Data (3box)
Shared Persistent Data (Ethereum 1.0)
Filecoin Integration (Textile Powergate)
API Exposure and Service Integration
React DApp (Fleek Hosted)
Smart Contract Model
Manufacturer
Developer
User
Feeless
Collect Donations
Small Donations
Ether (ETH)
Note: Manufacturers/Developers maintain file storage costs. Donation model in place to support Developers who also receive funds from picking up community requests.
Developer Incentivization
Firmware Upload
Developer GUI/PyQT5
Firmware Verification (CHAINIAC)
Authentication & Authorisation
Web of Trust Model
3box and Código Developer Auth
Distribution Layer
Base Components
IPFS
Ethereum Smart Contract
Filecoin
Device Integration API
Web3.0 Node Stack
MQTT Firmware Integration Modules

Managing User Data

One of the core benefits of moving to a decentralized infrastructure is facilitating user control over their own data. Our design utilizes the 3box data storage system to store any user specific information on the system, all of which is configurable from the user's perspective.

Background

3Box allows front-end web developers to keep user data on an open storage network instead of a centralized database server, browser local storage, or the blockchain.[1] To

[1] "Why 3Box - 3Box Docs." https://docs.3box.io/why-3box. Accessed 16 Jul. 2020.

facilitate this, 3Box utilizes OrbitDB[2] - a decentralized peer-to-peer eventually consistent database that works on top of IPFS.

Local OrbitDB/IPFS instances (a Root and Profile Store) are generated in the browser when users arrive at apps that use 3box. The users' data for other available nodes are then synced with that instance. Local changes to the database made using the 3box app which are later synced out to the network using IPFS pubsub.[3] Persistency is enabled through pinning-nodes (currently largely managed by 3box).

Users or applications can also create and store data in Spaces to keep information access controlled separately from the Root Store. Each Space adds another OrbitDB key-value database to the user's 3Box.

Threads are used to enable peer to peer communication and shared messaging, as separate OrbitDB databases. Threads are OrbitDB Feed Stores where one or multiple users can post messages, and these Threads live inside of Spaces. There can be many Threads mapped to a single application Space.

Implementation

There are three core pieces of user data:

- User profile data: name, picture, etc
- Any devices they have registered with the system
- The creation of requests and any comments on these requests that users make

This first component is managed by integrating with 3box's user profile management service, utilizing these React components: Profile Hoover and 3box Profile Edit.

For the second, we utilize a dedicated application space to manage user configuration details around their device management.

The community requests feature is enabled through the use of shared user threads.

These are interfaced with the social web3 profiles users create when registering with 3box. This utilizes the users ethereum address and window.web3.currentProvider which enables access for the application to their user data. This is additionally secured by password

[2] "orbitdb/orbit-db: Peer-to-Peer Databases for the ... - GitHub." https://github.com/orbitdb/orbit-db. Accessed 16 Jul. 2020.
[3] "Data Storage - 3Box Docs." https://docs.3box.io/faq/data-storage. Accessed 16 Jul. 2020.

protecting the application. Prior to this, they will have to have already authenticated using the ethereum network.

Firmware and User Contributions

Central to our application is the shared deployment of the firmware. This information is shared between all users with a transparent, viewable and distributed contribution record.

Background

The original Código project already provides a smart contract that tracks firmware uploads to IPFS. It allows clients to search for firmware either by knowing in advance the developer they want the firmware from or by choosing a developer automatically using a web of trust implementation. The front end application builds on this by allowing easier discovery of new firmware for users and providing a convenient interface for developers to manage their firmware uploads and storage on Filecoin.

Implementation

For this purpose we utilize an updated smart contract (Ethereum 1.0). This smart contract is an updated version of the contract from the original. The smart contract is used to address the firmware on the IPFS network.

- The hashes for the firmware binary and source are each stored as 32-byte arrays
- IPFS link stored as a string
- Firmware description stored as a string
- Block number stored as 256-bit unsigned integer
- Target device name stored as a string
- Download numbers
- Plus any incentivization scheme necessities

The user account ID provided by 3box is linked to the ethereum address used to publish firmware to the network via a separate smart contract. This allows a developer who has already published firmware with one address to attach it to an arbitrary user account. If there is no link stored then the user account is assumed to be the same as the developer account.

The process of linking an account to the Código network is as follows:

- The application requests a challenge value providing the address of the account it wishes to link with
- The smart contract responds with a randomly generated 32-bit value
- The application signs the challenge value with the private key of the account it wishes to link with
- The application responds to the smart contract with the signed challenge
- The smart contract verifies that the challenge was signed with the correct private key
- The smart contract stores a two-way mapping between the address performing the transaction and the one provided in the initial challenge request. This mapping can be queried later as necessary

Filecoin Integration

Filecoin was a major addition to our project. Firmware files are now stored on the Filecoin decentralized storage network rather than on plain IPFS nodes.

Background

Since Filecoin is overlaid on top of IPFS, this change is backward compatible and IPFS can still retrieve firmware. On the other side though, Filecoin provides persistence and more control than IPFS.

Previously, a developer uploading firmware to IPFS would need to make sure that there are IPFS nodes willing to pin it, otherwise, it would be lost eventually. For example, a developer had to run their own IPFS node which would pin all their uploads, or they could depend on the firmware users pinning the firmware. The first case is bad in terms of decentralization and availability of the firmware, since all the developer’s uploads are reachable only if a single IPFS node is reachable. The second case is even worse because no one guarantees the persistence of the firmware unless it is popular and noticed early enough by users willing to pin it.

Using Filecoin we give full control to the developer to choose how their files are persisted. The developer can choose the level of duplication, the lifetime of a storage deal and many more attributes. Afterward, the Filecoin network guarantees the persistence of the files without the developer needing to do anything else. A downside to this is that the developer incurs charges for storing files to the network. We believe this is not a deal-breaker because:

- Using IPFS, the developer would still incur charges if they were running their own infrastructure to pin the files

- The developer can still run their own Filecoin miner and thus avoid the Filecoin charges. The upside of this is that the developer wins extra money by providing retrieval deals to users willing to download the firmware.

Implementation

We are interfacing with the Filecoin network through a hosted Powergate instance, provided by Textile.

The process is as follows:

1. When the Dapp loads, we initialize our Powergate client instance by connecting to the Powergate host.
2. To obtain write access to the network, we request a token from the host. This token is tied to all the accounts that the user will create with it and all the file uploads and downloads that will be performed. Thus, we cache it in the browser local storage, to retrieve it at a later time
3. We retrieve all the user accounts associated with the token or we create a new one if there are none
4. When the user performs a firmware upload, the file is staged on IPFS and then a file storage deal is published to the Filecoin network.
5. When a miner accepts the deal, we proceed to register the firmware with the Codigo firmware repository smart contract.
6. Lastly, when a user wishes to download some firmware, we first check if it’s still stored on IPFS. If that’s the case, the file is fetched from IPFS without any charge to the user. Otherwise, a retrieval deal is published to the Filecoin network and the file is retrieved with a small charge in Fil, the cryptocurrency of the Filecoin network.

User Interface Implementation

The User Interface (UI) is made with React and hosted on Fleek.

Background

The app front end is built using React and is hosted on Fleek. This maintains the decentralized nature of the system and offers us continuous deployment as Fleek will build and publish the app to IPFS whenever the master branch is modified. As Fleek deployment is front-end only much of the business logic of the app is implemented client side in Javascript. However, all information that is shared between clients or is exploitable (e.g related to developer earnings, bounties or user reputation) is managed via Ethereum smart contracts.

Implementation

A template for the React app is originally sourced from here and heavily modified and improved for our purposes. It integrates Redux which we use to coordinate all the dataflows in our application.

The UI is designed to be clean, simple and minimalistic. Our intention is to make as easy to use as possible. The UI has been designed with both developers and firmware downloaders in mind. The signup process has been made sleek and succinct to allow the 3box account to be linked to Codigo quickly and easily.

Incentivisation Scheme

The developer incentivization scheme currently revolves around bounties. Users can post requests for specific firmware support/features and pledge an arbitrary amount of ETH which will be released to a developer who fulfills the request. Users can also contribute further ETH to an existing request in order to make it more lucrative for a developer to pick up. Once a developer produces appropriate firmware they can claim the bounty, receiving the pool of ETH currently attached to it.

In the prototype system implemented the bounty system has one weakness, it is impossible to automatically verify that firmware produced to fulfill a bounty actually matches an arbitrary request. The solution we propose is to integrate user reputation into the bounty system. For example when a developer claims to have fulfilled a bounty the firmware would be first released to a small number of high reputation users (see below) who would act as moderators. The bounty reward would not be released to the developer until the provided firmware passed moderation. Moderation could be conducted either by flashing the firmware image onto a test system or by manual source code review at the discretion of the moderator. A system such as CHANIAC[4] can be used to automatically verify that the code a moderator reviews is the same as the code that the firmware image was built from.

User Reputation

The user reputation system mentioned above currently exists and is displayed on a user's profile page, so users can query one another's reputation. Currently it isn't used for bounty verification as described above.

[4] https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin

User reputation works as follows. Note that the reputation thresholds given were chosen for development convenience and would likely need to be changed in advance of real world deployment:

- New users start with a reputation of 0
- Reputation cannot fall below zero, so new users have less to lose by contributing to the system vs already trusted users. This lowers the barrier of entry for new users.
- Reputation can effectively grow arbitrarily high (reputation is stored as a uint256 in Ethereum) but increasing reputation only has an effect up to a value of 100
- Users gain a fixed reputation amount (currently 10) for submitting firmware to the system
- Users gain between 1 and 10 reputation if somebody upvotes a comment they made or firmware they submitted
- Users lose between 1 and 10 reputation if somebody downvotes a comment they made or firmware they submitted
- The reputation a user gains or loses is dependent on the reputation of the user making the upvote/downvote. This is calculated by dividing the voting user’s reputation by 10, flooring and capping at 10.
- As a result of the previous point users with <10 reputation can upvote/downvote but doing so doesn’t affect the target user’s reputation. This prevents a dishonest user from making many new accounts to increase their own reputation or decrease somebody else’s.

Milestones

I. Checkpoint 1: Technology Research and Specification

Initially we focussed on specifying the system design. We chose the technologies we needed to research in order to reach our goal for the application: A fully decentralized driver/UI for the Código Network, distinct from the network itself. UI designs were drafted and in some cases prototyped and backend engineering decisions were made.

II. Checkpoint 2: UI and Client Side backend work

After deciding to deploy the app on Fleek we produced initial prototypes of all of the main views required to support our desired functionality. In order to solidify the requirements for backend work. Work was also started to integrate 3box user profile management, an issue with 3box delayed progress on this feature initially.

Finally, we deployed the original Código Network source contract on the Ropstein test network so that our app could communicate with it.

III. Checkpoint 3: Further UI work and Ethereum integration

With the main parts of the UI stabilizing we began work on integrating with Ethereum. Both to fulfill the stated purpose of the project to provide a Código UI and to utilize Ethereum to store data unique to our app which should be safe from user manipulation. The new smart contracts were also deployed on Ropstein once created.

IV. Checkpoint 4: Final UI polishing, demo prep and additional features

In the final sprint we added Filecoin management support, communication with devices via MQTT, a comprehensive search feature, the bounty system and user reputation tracking to the app. No new functionality was added to the UI but existing features were reviewed and tweaked as necessary. Finally, time this sprint has been reserved for preparing submission and demo materials ahead of the deadline.