TRUESHOE: A BLOCKCHAIN ENABLED MARKETPLACE FOR AUTHENTICATED SNEAKERS

SYSTEM ARCHITECTURE REPORT

Benjamin Kha **Nikhil Sharma** **Dapree Doyle** **Ryohei Kobayashi**

ABSTRACT

The problem that TrueShoe aims to solve is that fake sneakers are currently prevalent on sites such as eBay. There exists services that authenticate sneakers, but these services charge high fees to their users and are inconvenient to use because the shoes have to be manually sent in for inspection. Our solution is to create a blockchain enabled marketplace that allows users to conveniently authenticate sneakers at a much lower cost than current solutions. This is achieved by partnering with shoe manufacturers to burn in a unique identifier on their sneakers, for which they will receive a comission on all transactions involving their sneakers on TrueShoe. In this report, we will provide a detailed analysis of how our service works at a technical level.

1 System Architecture Overview

Our system architecture can be broken up into 4 main services: the shoe verification service, the seller service, the listings service, and the delivery service, which can be seen in Fig. 1. Sellers interact through our frontend application with the seller service, while buyers are shown open listings and agree to buy listings through the listings service. When a seller posts a listing on TrueShoe, he or she scans the burned in identifier, which triggers the shoe verification service to verify that the sneaker is indeed genuine. After a buyer agrees to buy a certain listing, the seller will be notified and be asked to provide the delivery information for his or her shipment. Then, the listings service will create a smart contract on the blockchain, to which the buyer must send funds to. The smart contract will periodically query our delivery oracle, which will then query the USPS (or the appropriate delivery service) website for the delivery status. Once the delivery status is returned as being "delivered", the smart contract will initiate the transfer of funds to the seller's ether address.

2 Frontend and Database Store

A mockup of our frontend interface can be seen in Fig. 2 of what the TrueShoe user interface will look like, with a full interactive demo online[1].

TrueShoe will be implemented as a mobile platform with support for both Android and iOS, which we hope to extend to include a web platform as well eventually. We will construct the frontend utilizing React for development, and use Google's provided integration with Firebase to seamlessly tie our user interface together with the necessary backend support.

The Firebase integration provides a realtime database[2] which allows for efficient querying and synchronization of sneaker listings across our user base. This design choice is important due to the time-critical nature of our product; our auction-based business model mandates low-latency synchronization across consumers for optimal customer satisfaction.

[1] https://marvelapp.com/3hcg2ae/screen/50209798

[2] https://firebase.google.com/products/realtime-database/

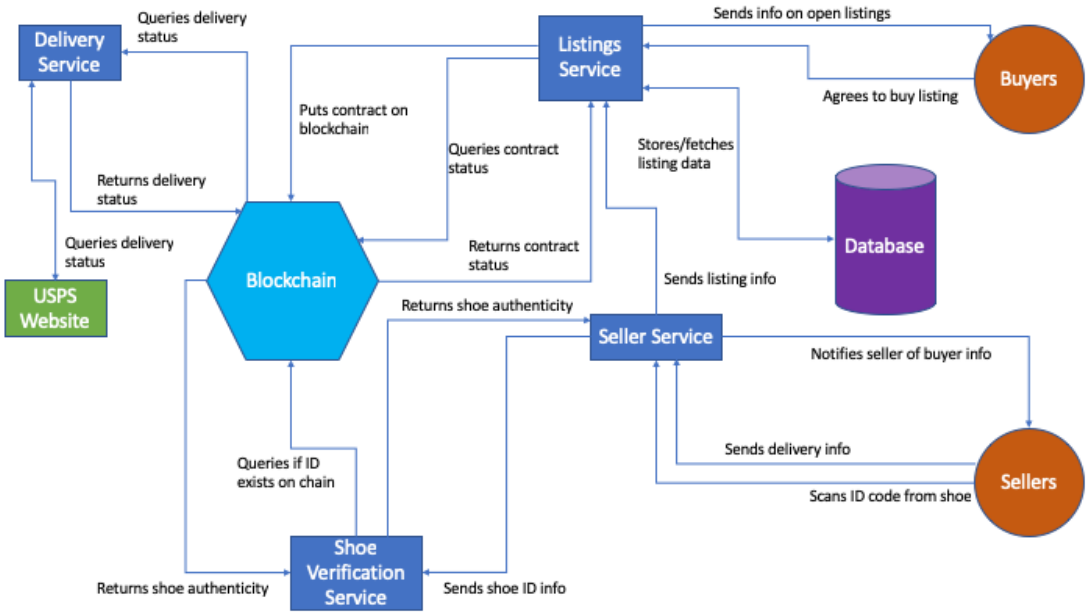

Figure 1: System Architecture Diagram

Firebase also provides support for other features that are relevant to our product. Specifically, its built-in chat framework[3] support will allows us to easily integrate a feature into our application which allows vendors and consumers to get in touch and discuss any concerns relating to the details of the smart contract for a given purchase (detailed more in the Smart Contracts section below). Additionally, Firebase provides both a native integration for OAuth2[4] allowing users to authenticate with ease and an integration with Google Analytics[5] that will allow us to track app usage effectively and update our business model accordingly.

3 Smart Contracts and Cloud Resources

Smart contracts will be utilized to automatically transfer funds from buyers to sellers after the delivery is confirmed. An overview of how the different parties are connected through the smart contract can be seen in Fig. 3.

We will be using smart contracts for two distinct purposes within our system architecture:

1. The primary technical contribution TrueShoe makes in the space of sneaker resale is using smart contracts to transfer payment upon verified delivery of authentic sneakers from vendor to consumer. As mentioned in the previous section, the `SneakerTransferContract` will rely upon a `DeliveryService` which serves as an oracle for triggering asset transfer upon successful transaction by scraping data from the delivery carrier website (e.g. USPS).
   (a) Notably, this system allows vendors to behave maliciously; simply by never delivering any purchased sneakers to their customers, they can lock away their customers assets within the smart contract. To address this problem, we will be implementing a timeout feature, which requires vendors to provide a delivery guarantee window after which the consumer is reimbursed their payment regardless of whether or not the sneakers are delivered after the timeout.
   (b) Timeout policy: the specifics around the timeout policy are business-critical; without a good policy customer dissatisfaction will be high. Hence to select the best policy we've decided some analytics are required to monitor app usage and customer satisfaction. Towards this end, we will be experimenting with the following three possibilities and checking which yields the best results:
      i. Set a default global timeout window (i.e. 1 month).

[3] https://firebase.google.com/docs/cloud-messaging/
[4] https://firebase.google.com/docs/auth/
[5] https://firebase.google.com/docs/analytics/

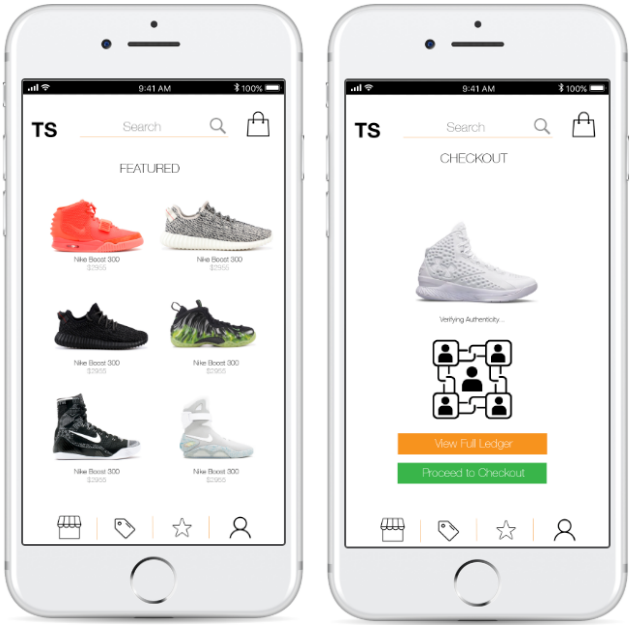

Figure 2: TrueShoe mobile user interface

ii. Require vendors to transparently provide a timeout to the users, displayed along with a base item price when adding an item to sell in the TrueShoe marketplace.

iii. Allow customers and vendors to negotiate a timeout window pre-purchase, either as part of the auction itself or via our Firebase chat integration.

2. We will also use smart contracts to drive the sneaker resale economy via an auction-based system. Similar to eBay, consumers will be able to place bids for items directly through our interface, and these bids will be binding by depositing consumers bid amounts within the items corresponding SneakerResaleAuctionContract. Once the auction ends, its smart contract will reallocate funds to the vendor, the consumer, shoe manufacturers, and our company in accordance with our business model.

The blockchain platform we will utilize is the Oasis Labs Devnet[6], a new decentralized cloud platform that provides privacy-preserving cloud computing built on Ethereum. The Oasis Devnet is a very new technology which combines existing smart contract technology with secure hardware enclaves and differential privacy to provide a simple, fast, and secure interface for building dApps. Such a platform was selected with the foresight that TrueShoe will ultimately expand to require more compute-intensive functionality (i.e. private decentralized machine learning for sneaker recommendation to our users).

4 Oracle

TrueShoe requires an oracle for validating shoe delivery, and for this we include a `DeliveryService` in our design to query delivery tracking information on the delivery carrier website. The design choice to include `DeliveryService` as a component existing outside the products blockchain ecosystem was intentional, enabling a greater degree of adaptability to configuration/API changes made by USPS (or other carriers) without affecting the implementation of our `SneakerResaleAuctionContracts` for purchases.

We wanted to design a `DeliveryService` policy with the following two important properties:

1. The `DeliveryService` should stop querying after the vendor-specified timeout delivery period (plus an added grace period which we will select robustly using our data analytics).
2. The `DeliveryService` should attempt to pay vendors in the most timely manner possible.

[6]http://docs.oasiscloud.io/en/latest/quickstart-guide/

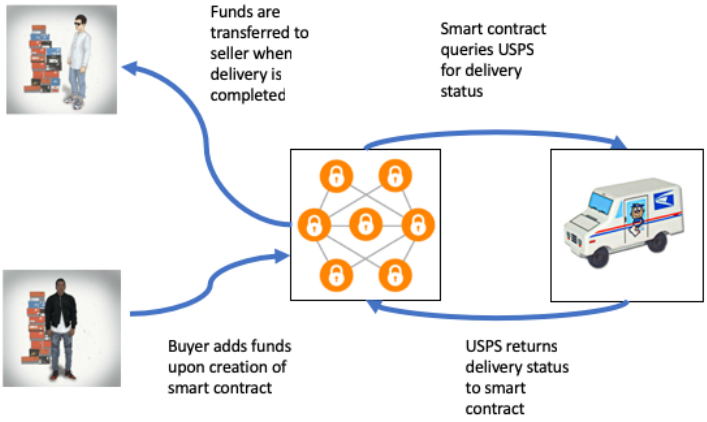

Figure 3: Smart contract overview

```
{
  "name": "Nike Boost 300",
  "price": 2955,
  "description": "Used bright red Nike Boost 300",
  "img": "http://bucket.s3.amazonaws.com/nike_boost.jpg",
  "delivery_carrier": "USPS",
  "tracking_number": 1857474555
}
```

Figure 4: Data schema for listings information

As such, the local query policy for querying the delivery carrier website we have selected behaves according to the following equation:

$$\text{sample\_time}(i) = \min(\text{mean\_delivery} + 2^i - 1, \text{timeout} + \text{grace\_period})$$

As you can observe, the smart contract starts polling for delivery at the vendors mean delivery time, then continues to poll with exponential backoff until the timeout window (plus grace period) expires. This policy is enacted trying to achieve a balance between querying too frequently to be effective, and query too infrequently and delaying vendor payment. This policy is far from perfect, but we hope it is a reasonable starting point while our user base remains small. As it grows, we hope to eventually extend this local policy to a global querying policy that rate-limits queries across all active auction smart contracts; this will likely be based on query time series analysis and machine learning.

5 Backend/Data Format

For our services, we plan to use Python, specifically leveraging frameworks such as Python Flask for the backend. To communicate between services, we plan to use to the JSON data format, specifically because it is well supported among all the different components of our product: frontend, backend, and blockchain. As an example of what this JSON data would look like, consider what happens as a seller is trying to create a listing.

After a shoe has been verified by the `ShoeVerificationService`, the seller will be asked to provide information that will be sent to `ListingsService`. The information requested will be include things like the name of the listing, the price, and also the delivery information, an example of which can be seen in Fig. 4.