Building Smart Contracts

Zorawar Moolenaar

Computer Security Final Project Submission

Summary

The final submission surpasses the expectations of the proposal. In addition to relatively increased comfort with programming in `Solidity`, I have explored specification for interesting dapps[1] like Storj, Cryptokitties/Etheremon, and Dharma among others. Additionally, the final deliverable is a full-stack proof-of-concept dapp, rather than just a smart contract as originally planned. Although the contract employs some novel features of `Solidity`, these only represent a subset of what I have learnt, owing to simplicity of design.

1. Description of Deliverable

The deliverable is based on and extends the final project[1] for the 2018 "Cryptocurrencies and Blockchain Technologies" course at Stanford University.

This project description is inspired by a centralised service called SplitWise, and aims to produce its decentralised counterpart. Splitwise is an application that allows users to collect and resolve owed money ("IOUs") amongst each other. Its decentralised version, called *DecentralisedSplitWise* attempts to a decentralise the service so that users can resolve owed money without any central authority.

The problem description lays the foundation of a few problems that need to be solved. In addition to delegating decisions about client- or contract-side computations, the problem asks to build a few computational features that facilitate peers mutually solving IOUs. It provides some starter-code with limited functionality. Most of this code has been rewritten to extend the constraints of the application.

[1] decentralised applications

1.1. Main Accomplishments

1. Solves the "Loop of Debt" Issue: If A pay B, B pays C, and C pays A, the system automatically adjusts IOUs so that no cycles exist (implemented partly in the contract, in `add_IOU`).
2. Implements 5 other features, required by design:
   (a) `Lookup` (contract-side): computes the value that a user owes another.
   (b) `getTotalOwed` (client-side): computes the total amount that the given user owe.
   (c) `getUsers` (client-side): Returns a list of addresses associated with everyone currently owing or being owed money.
   (d) `getLastActive` (client-side): returns the last active time for a given user.
   (e) `send_IOU` (client-side): allows users to send IOUs via the web interface.
3. Extensions to the base project (listed below)
4. Gained significant knowledge about items listed on the proposed roadmap.

1.2. Technologies Employed

1. Solidity + Remix: for contract writing and deployment
2. Ganache: personal blockchain for ethereum development
3. web3.js: Ethereum JavaScript API
4. HTML/CSS/JS/jQuery

1.3. Extensions

1. API-like extension: providing API-like functionality to neatly communicate with the client-side and possibly with other contracts
2. Relaxing Problem Constraints: the problem asks programmers to assume that only a few transfers/IOUs will be made, so that the entire chain can be iterated over on the client-side. At the cost of some extra gas, this constraint has been relaxed. Further, the need to iterate over the entire chain is removed entirely.
3. Object-oriented Design: For greater expressiveness in design, the contract exploits Solidity's OOP features. It is written in a modular manner, and exploits composition and interface inheritance.
4. Low resource consumption: the contract can complete a sizable number of transaction without any issues, at low gas-price.

5. Contract-oriented UX improvements: Fewer clicks/ refreshes to use the application, thus improving the user's overall workflow.
6. Other Solidity Features: For better communication between the contract-computation and the user frontend, `events` are employed. Events are not used to their full capabilities in the front-end, but the contract-side is fully functional. The application also enforces constraints by using `modifiers`.

1.4. Limitations

Since this project is intended as a proof-of-concept, it does not implement user authentication. Consequently, by design, the user operates in *god-mode*. That is, they can send IOUs on behalf of any user and can test the overall working of the application. Practically, a user would only make transactions associated with their own address.

In implementing user authentication, it may be necessary to modify how the loop-of-debt is resolved. In this iteration of the application, such modifications were omitted in deference to the YANA software design principle. The current approach is probably in-line what the project-assigners intended, for it leverages a BFS pathsearch algorithm provided in JavaScript in the starter code.

1.5. Future Scope

The application curretly operates in *god-mode* and, while core functionality exists, a user experience needs to be designed around it. Since this ability requires heavy front-end engineering, it was omitted for this project. Further, since contracts are permanent, thorough testing must be done before deploying to a blockchain.

2. Addendum

In pursuit of learning more, I completed several published "courses" on Blockchain Technologies and Smart contracts, notably from Lynda.com, Cryptozombies.io, the Stanford Blockchain course, and the "Will It Scale" series.

Consequently, I experimented with plenty of features that were not incorporated into the project for simplicity. Some features include libraries, external-contract communication, time-based events, ERC20 standard, ERC223 standard, among others. As anticipated, I also learned a lot more about the

Ethereum blockchain, the internal workings of Solidity (a bit about low-level EVM through debugging, the different memories, how Solidity enabled the DAO hack) et cetera. Some of these experiments were also conducted on frameworks like Truffle by Consensys.

2.1. Challenges

Solidity is inspired by python, C++, and JavaScript. I don't particularly like these languages for full applications, but Solidity is quite alright in comparison. It is under active development, thus many features that one expects of a modern language are absent, experimental or exorbitantly expensive. Notably: nested arrays, returning/passing non-primitive data type, string operations. To add it its burden (and make its flaws more understandable), programming on a global computing platform presents unique challenges in terms of memory and computational constraints. I imagine, this is what programming a few decades ago was like, when instantiating variables and making function calls was an expensive task. The additional technologies in my toolkit like ganache had bugs of their own and took a while to discover that my program was working as intended.

However, at the same time several different approaches tackle the same problem as solidity+ethereum, and different individuals have varying opinions of its quality. It may be worth trying them out at some point. It is also marvellous to see products that enable subtokens, digital scarcity, decentralised orchestration, and identity management, among other uses.

3. References

[1] D. B. . D. Mazires, Project #3: Ethereum de-centralized app (Dapp), `http://cs251crypto.stanford.edu/18au-cs251/hw/proj3.pdf`, 2018.