Smart Contract Orchestration Patterns

How to make your smart contracts cooperate safely

Alberto Cuesta Cañada

Follow

Coinmonks

--

Listen

Share

All but the simplest of Ethereum applications are composed of several smart contracts. This is because of a hard limit of 24KB in any deployed contract, and because your sanity will slip away as the complexity of a smart contract grows.

Once you break down your code into manageable contracts you will certainly find that one contract has a function that should only be called by another contract.

For example, in Uniswap v2, only the contract factory is supposed to initialize a Uniswap Pair.

The Uniswap team solved their problem with a simple check, but wherever I look I find more examples of orchestration solutions being coded from scratch for each project.

In the process of understanding this problem and developing a pattern, we understood better how to build applications out of several smart contracts, which made Yield more robust and secure.

In this article, I'm going to look in depth at smart contract orchestration with examples from well-known projects. By the time you are done reading it, you will be able to look at the orchestration requirements of your own project, and decide which one of the existing approaches suits you.

I made this example repository to get you going, when you are ready.

Let's dive in.

Background

I advanced already that you will need to break down your project into a collection of smart contracts because two limits, one technical, one mental.

The technical limit was imposed in November 2016. At that time the Spurious Dragon hard fork was implemented in the Ethereum mainnet, including EIP-170. This changelimited the size of deployed smart contracts to a maximum of 24576 bytes.

Bitten by the Dragon

Without this limit, an attacker could deploy smart contracts taking infinite amounts of computation during deployment. It wouldn't impact any of the data stored in the blockchain, but could be used as a kind of Denial-Of-Service attack on Ethereum nodes.

At the time the block gas limits didn't allow for smart contracts of that size anyway, so the change was considered non-breaking:

"The solution is to put a hard cap on the size of an object that can be saved to the blockchain, and do so non-disruptively by setting the cap at a value slightly higher than what is feasible with current gas limits (an pathological worst-case contract can be created with ~23200 bytes using 4.7 million gas, and a normally created contract can go up to ~18 kb)."

That was well before the DeFi explosion. For Yield we coded 2000 lines of smart contract code, which deployed would add up to close to 100 KB. Our auditors didn't even consider our project to be very complex.

Yet we still had to break down our solution into multiple contracts.

Complexity and Object-Oriented Programming

The second reason to break down a blockchain application into several smart contracts has nothing to do with technology limits, but with human limits.

We can only keep that much information in our brains at a given time. We perform better if we can work in smaller problems that interact in limited ways, than in one single large problem where everything can interact with everything.

It is probably fair to say that Object-Oriented Programming allowed software to reach a higher level of complexity. By defining “objects” that represented some concept, and defining variables and functions as properties of an object, developers could make better mental images of the problem they were solving.

Solidity uses Object-Oriented Programming, but at the contract level. You can think of a contract as an object, with its variables and functions. A complex blockchain application will be easier to picture in your mind as a collection of contracts, each representing one single entity.

For example, in MakerDAO you have separate contracts for each one of the cryptocurrencies, another contract for recording debt, separate contracts to represent gateways between the debt vault and the external world, among many others. Trying to code all this in a single contract might be impossible. If possible at all, it would be really hard.

Breaking big problems into smaller ones that interact in limited ways really helps.

Implementation

In the next section, we will look at the orchestration implementations of Uniswap, MakerDAO and Yield. It will be fun.

The simple one — Uniswap and Ownable.sol

I like Uniswap v2 because it is so simple. They managed to build a hugely successful decentralized exchange in 410 lines of smart contract code. They only have two types of deployed contracts: One factory, and an unlimited number of pair exchange contracts.

Because of the way their factory contract is designed, the deployment of a new pair-trading contract is a two step process. First the contract is deployed, and then it is initialized with the two tokens it will trade.

I don’t know the vulnerability they were protecting themselves against, but they needed to ensure that only the factory that created the pair-trading contract would initialize that contract. To solve this problem they reimplemented the Ownable

pattern.

It worked out for them, and it will work out for you if your case is as simple as theirs. If you know that your contract only needs to give privileged access to only one other contract, then you can use Ownable.sol

. You don’t even need to use a factory like Uniswap. You can have one user deploy two contracts (Boss

and Minion

, with Minion

inheriting from Ownable.sol

), and then execute minion.transferOwnership(address(boss))

.

The complete one — Yield

For Yield we didn’t manage to code a solution as simple as Uniswap v2. Our core is five contracts, and privileged access relationships are not one-to-one. Some contracts have restricted functionality that we need to make available to several contracts in the core group.

So we just extended Ownable.sol

to have two access tiers, one of them with multiple members:

The contract owner can add any addresses to a privileged list (orchestration

). Inheriting contracts can include the onlyOrchestrated

modifier which will restrict access to registered addresses.

As an additional security check, each address is registered along with a function signature, narrowing down the access to a single function in the Orchestrated

contract. Check the example repository for details on this.

There isn’t a function to revoke access because we would orchestrate

the contracts during deployment, and then owner

would renounce to its own privileged access by calling transferOwnership(address(0))

on all contracts.

Our own platform token, yDai

, would inherit from Orchestrated

and limit mint

to specific contracts set up during deployment:

This pattern is relatively easy to implement and debug, and allows us to implement functions that should be used only by contracts we control.

The obfuscated — MakerDAO

MakerDAO is infamous for using nonsensical terminology, which makes it extremely hard to understand. It was only well after I solved orchestration for Yield that I realized they use almost exactly the same implementation.

1. The contract deployer is the original member of the wards

.

1. wards

can rely

on other people (usr

) so that they also become wards

.

1. Functions can be restricted (auth

) so that only wards

can execute them.

As an example, the fold

function in MakerDAO’s Vat.sol

contract is used to update an interest rate accumulator, and should only be called by another of the contracts in their collection (Jug.sol

contract, drip

function). If you look at the function you will see the auth

modifier, which is what they use for orchestration:

In a way, auth

and the other orchestration implementations are an extension of the private

and internal

functions concept, only for access control between contracts.

The MakerDAO implementation is very similar to the one we came up with ourselves.

1. The contract deployer is the original member of the wards

. In Yield it would be owner

.

1. wards

can rely

on other people (usr

) so that they also become wards

. In Yield, only the owner

can designate other addresses as privileged.

1. Functions can be restricted (auth

) so that only wards

can execute them. In Yield we say that onlyOrchestrated

addresses can call the marked functions. We further restrict access to a single function.

Except for the fact that in Yield we used two access tiers (owner

and authorized

) and individual function restrictions, the implementation is the same. Contract orchestration is a common pattern that could be implemented once and reused often.

To make auditors and users even happier, we also developed a script thattrawls the blockchain events and paints a picture of ownership and orchestration for our contracts. This script will be available from our website at go-live and proves that no one has or ever will have privileged access to them, except the contracts set up at deployment.

After all, that’s the whole point of contract orchestration.

Conclusion

Orchestration of smart contracts is a problem that repeats itself in most projects, and which most projects implement from scratch. Often the solutions implemented are nearly identical to each other.

While a canon emerges that we all can use to implement orchestration safely and efficiently, please use the examples in this article to understand and implement a solution for your requirements. Use the code in the example repository if it suits you.

Finally, thanks for reading this, and if you have any questions or feedback, please don’t hesitate to contact me.

Thanks to Allan Niemerg, Dan Robinson and Georgios Konstantopoulos for their excellent feedback while coding theYield smart contracts.

Also, Read

- The Best Crypto Trading Bot
- Crypto Copy Trading Platforms
- The Best Crypto Tax Software
- Best Crypto Trading Platforms
- Best Crypto Lending Platforms
- Best Blockchain Analysis Tools
- Crypto arbitrage guide: How to make money as a beginner
- Best Crypto Charting Tool
- Ledger vs Trezor
- What are the best books to learn about Bitcoin?
- 3Commas Review
- AAX Exchange Review | Referral Code, Trading Fee, Pros and Cons
- Deribit Review | Options, Fees, APIs and Testnet
- FTX Crypto Exchange Review

- NGRAVE ZERO review
- Bybit Exchange Review
- 3Commas vs Cryptohopper
- The Best Bitcoin Hardware wallet
- Best monero wallet
- ledger nano s vs x
- Bitsgap vs 3Commas vs Quadency
- Ledger Nano S vs Trezor one vs Trezor T vs Ledger Nano X
- BlockFi vs Celsius vs Hodlnaut
- Bitsgap review — A Crypto Trading Bot That Makes Easy Money
- Quadency Review- A Crypto Trading Bot Made For Professionals
- PrimeXBT Review | Leverage Trading, Fee and Covesting
- Ellipal Titan Review
- SecuX Stone Review
- BlockFi Review | Earn up to 8.6% interests on your Crypto