Web3 Weather Oracle

Tommaso Battistini

February 2023

Contents

1 Introduction

1.1 In a nutshell

Web3 Weather bets allows users to bet cryptocurrencies (Ethereum and Algorand) over the weather in Paris, or to run a centralized oracle in your machine for other gamblers to play (and tip you for your service).

1.2 Responsibilities and references

Since this is an individual project (and, in particular, the "experimental" side of my Master Thesis), I personally took care of all the development. I don't really have sources I used as a reference, as this work has been coded in a brand new language (described later). To the best of my knowledge, this is the first oracle built with this technology!

2 Background

2.1 Blockchain technology

Blockchain technology enables secure and transparent record tracking of digital transactions, that in turn are the building blocks of the blockchain's... blocks. This technology is decentralized - or in other words - managed by a network of computers (nodes) that collaboratively validate new transactions and append them to the existing blockchain.

The origins of this new technology can be traced back to the 1990s. The first successful implementation can be identified in Bitcoin, a (now famous) protocol that was launched in 2009. Bitcoin relies on a proof-of-work consensus algorithm to validate transactions and secure the network.

Since then, a plethora of new protocols has been developed, each aiming to satisfy a different kind of need. Today, it is being used for a wide range of applications, from supply chain management to voting systems. One of the most notable and used features this area's developement is smart contracts.

2.2 Smart Contracts

Smart contracts are, ultimately, programs that are stored on the blockchain. Usually, they are used to automate and execute agreements between parties when a certain trigger condition is met, enabling users to be certain that when that condition verifies the outcome is guaranteed, without the need to involve any third intermediate parties. However, the adoption of this potentially groundbreaking technology must overcome an important limitation regarding this trigger condition, that must somehow be tied to a real-world event for smart contracts to achieve mass adoption of use case. As a matter of fact, blockchains don't really have a built-in function that allows them to pull or push data from or to an external system. This feature is the consequence of

blockchain isolation, or in other words the property that makes these protocols secure and reliable.
This Barrier is known as the Oracle Problem.

2.3 Oracles

2.3.1 The Oracle Problem

Oracles are what connects smart contracts to the outside world. Most of the information we could potentially use as a trigger contidion for them to run lies off-chain. Blockchain oracles are those pieces of middleware that enable communication between the blockchain and any other offchain system, like:

- Web Api's
- Cloud Providers
- IoT devices
- Sensors
- Payment systems

...and many more!
The task of Oracles is crucial for this field's developement. Blockchain technology could potentially revolutionize a number of industries by providing security, transparency and efficiency in it's novel way to distributely store data. Oracles, along side with smart contracts, are among the key features that this technology needs to change the world.
In order to complete their complicated (yet necessary) task they take on a number of functions:

- Listening - Oracles monitor the blockchain to chack if users or smart contract are requesting offchain data
- Extraction - Arguably their core function. Oracles fetch data from one (or more) offchain systems
- Formatting - Manipulate extracted data to obtain a version that is readable by the blockchain
- Computing - perform some kind offchain computation for the smart contract (in the case of this work, the Oracle ultimately decides who won the gamble).

For the oracle to succesfully carry out the tasks described above, it is necessary for its system to act both on and offchain simultaneously. In particular,

- the on-chain component attaches to smart contracts, listens for data requests, broadcasts data
- the offchain component processes requestss, retrieves and formats data, performs computation that might require extra privacy/security.

2.3.2 Oracle Classification

Oracles can be classified in a number of ways, by looking at them from different perspective. For example, we could analuze software oracles, hardware oracles or consensus oracles, besed on what kind(s) of data source they rely on. Another categorization, into 4 types, can be found here: as stated in [1], they can be analyzed from two points of view: according to their behaviour and their nature.

- The communication can be pull-based or push-based
- The data flow can be inbound or outbound (wrt the oracle)

A more visual classification is provided in the image below (also taken from [1]):

	Pull	**Push**
Inbound	The **on-chain** component **requests** the off-chain state from an **off-chain** component	The **off-chain** component **sends** the off-chain state to the **on-chain** component
Outbound	The **off-chain** component **retrieves** the on-chain state from an **on-chain** component	The **on-chain** component **sends** the off-chain state to an **off-chain** component

Figure 1: Oracle Types

3 DApp Presentation

3.1 Aim of the DApp

After these necessary premises, it's finally time to present the DApp itself. WWO (Web3 Weather Oracle) is a web application that allows users to gamble over the weather in Paris, or to offer an Oracle-like service to Gamblers. These roles and functionalities are described using a Reach-like logic in this section. The aim of this tool is therefore that to enable gamblers to regulate their bets using smart contracts, and to bring real-time external data to the contract with the aid of an Oracle.

3.2 Why use blockchain technology?

There are a number of reasons why blockchain technology is a good fit for this application. To break them down,

- The state of the application is to be stored, as actions have different utilities in different states
- There are multiple users involved in every istance (bet)
- We don't have an always online trusted third party to oversee the trades
- Participants don't necessarily know each other!

3.3 Actors

As previously mentioned, there are three actors in each instance of the program's execution:

- **Bettor** - In other words, the deployer. In WWO, the Bettor creates, deploys and shares Smart Contracts for other users to attach to.
- **Gambler** - This actor attaches to an existing contract, or in other words, accepts the Bettor's terms to make a bet.
- **Oracle** - This actor also attaches to an existing contract, but doesn't bet. Once bets are maid by the other players, the oracle retreives the actual weather (from a web API) for WWO to decide who the winner is. This action is performed automatically: the user only lends his computer for the request to be performed from an external system with respect to any of the players.

4 Software Architecture

This project was developed using the following tools:

- The backend is completely built using Reach

- The frontend is built using react, html, css

Reach's stdlib also takes care the "Middleware". As you can tell, most of this DApp was built using Reach. Before delving into its software architecture, I believe a brief introduction to this technology is necessary. Moreover, since all the relevant code is packed in a single file, and since components are basically the DApp's users, the flow of information is described with words and ordered lists, as I believed these to be more explanatory. Overall diagrams and use-case graphs are displayed later.

4.1 Reach Overview

This brand new, high level Web3 language enables developers to build entire DApps from ground up, unlike tools like solidity, designed to only build smart contracts. When Reach programs are run, they:

- Create the code (solidity or teal) that each participant of the DApp should run in order to interact with the consensus network (and, of course, the front-end interface to allow the user to do so).
- Derives and creates the needed smart contract(s) from the set of instructions descripted above
- Produces individual clients to run the program (using docker)
- Formally verifies that:
    1. funds won't be locked in a contract forever
    2. participant frontends are honest
    3. a number of transaction rules are preserved (for example, checks that output volume == input volume)

4.2 Reach Properties

All these instructions, transactions and security checks are packed in a single reach (javascript-like) file, that is then compiled to the desired language according to the network (Solidity for Ethereum, Teal for Algorand). This allows DApps built with reach to be interoperable, and work on multiple chains.
Reach's perspective on DApps is basically that of a decentralized computation that a number of participants are going to perform collaboratively. Furthermore, in most of these contexts, these participants don't trust each other. During the execution of reach programs, participants make assumptions regarding how other participants compute values; however, since they don't trust each other, it's necessary to perform checks. These checks are passed over from reach to the consensus network. The role of the network is therefore that to be the intermediary between participants that need to perform their actions and take decisions (spend money) according to these actions. For this to be possible, each participant should be built and tested.

Building a DApp using this technology requires developers to take care of two things:

- The Reach program itself - the program that describes your DApp. This program can be broken down into two parts:
  1. The declaration of who the participants are (in terms of roles and functionalities)
  2. An ordered list (in time) of their actions in an execution.
- The user interfaces to allow participants to take decisions (such as transfer money, accept wagers, make bets...).

4.3 Architecture overview

In this section, the software architecture of the system is broadly describe. To start with, the following diagram gives a visual, general, description of the system:

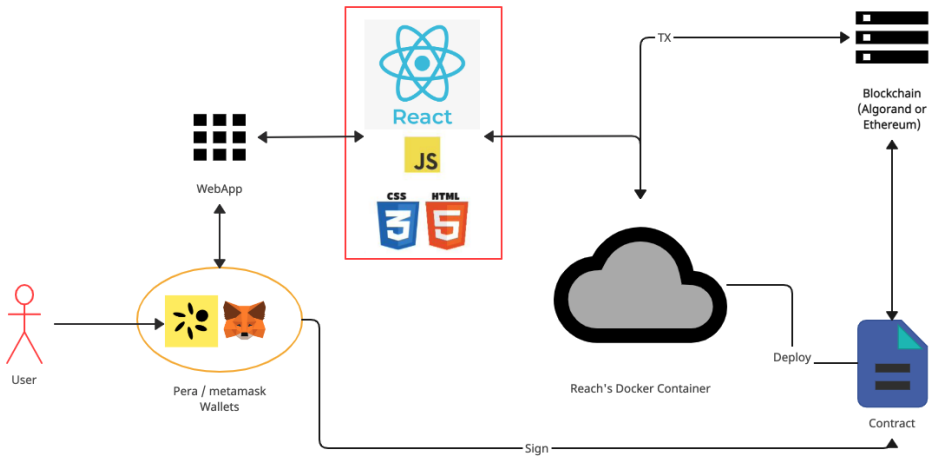


Figure 2: Possible Weather icons

While the reach component has been extensively spoken of in the previous sections, some others aren't:

- The Front end component was built using React, HTML and CSS. The first library is a widely used Javascript library for interfaces, that enables web developers to write efficient code thanks to it's component-oriented logic. Infact, each component can be reused and rendered multiple times in the browser. While these qualities alone would make a good enough reason to consider React the go-for technology for web developement, there was a more practical one in this case: Reach is naturally built to connect it's backend to react.
- The Wallet connection is also, actually, a built-in reach function.

4.4 Actions and Use Cases

4.4.1 Active and passive functions

Each of these actors has a specific set of actions they must perform for a bet to be resolved.

- All users must connect their wallet as soon as the WebApp is launched. If the wallet has no currency in it, a "fund-your-account" function is immediately triggered (unfortunately, this function is no longer available for Goerli and Sepolia ETH devnets).
- The **Bettor user** has to:
  1. Choose the wager for the current bet
  2. Make his bet (choose a possible weather) once the Gambler and an Oracle attach to the contract
- Under the hood, without the user actively doing anything, the **Bettor participant** also:
  1. Creates and deploys the contract after choosing the wager
  2. Pays the wager to the contract
  3. Computes and publishes a commitment of the weather he picks (more about this later), and a salt for this commitment's verification to be possible
  4. Tips the oracle for a small, fixed amount
- The **Gambler user** has to:
  1. Actively read and accept the wager (and therefore attaches to the contract)
  2. Pick his own predicted weather (or in other words, make his bet)
- Without any required action from the user, the **Gambler participant** also:
  1. Pays the wager to the contract
- Finally, the **Oracle** actively:
  1. Attaches to the contract
- and passively:
  1. Calls the AccuWeather Web API once both players have placed their bets
  2. Converts the received information to an appropriate format
  3. Publishes his discovery to the system

4.4.2 Use Cases

The following use-case diagram describes the possible actions that users can actively take from a use-case point of view:

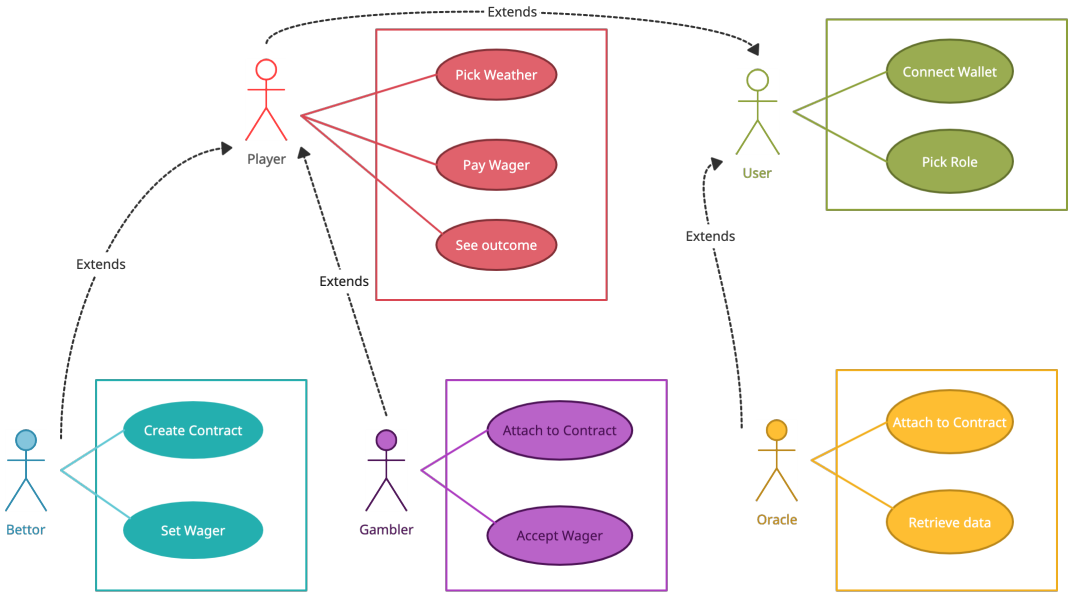


Figure 3: Possible Weather icons

5 Implementation

5.1 Frontend

This section describes the frontend implementation details of ths DApp. As previously mentioned, the user interface has been realized using React.js, HTML and CSS. The structure is quite simple, and the 3 different users share a number of components. The first, starting, view is immiediately shown as soon as the user connects his wallet:

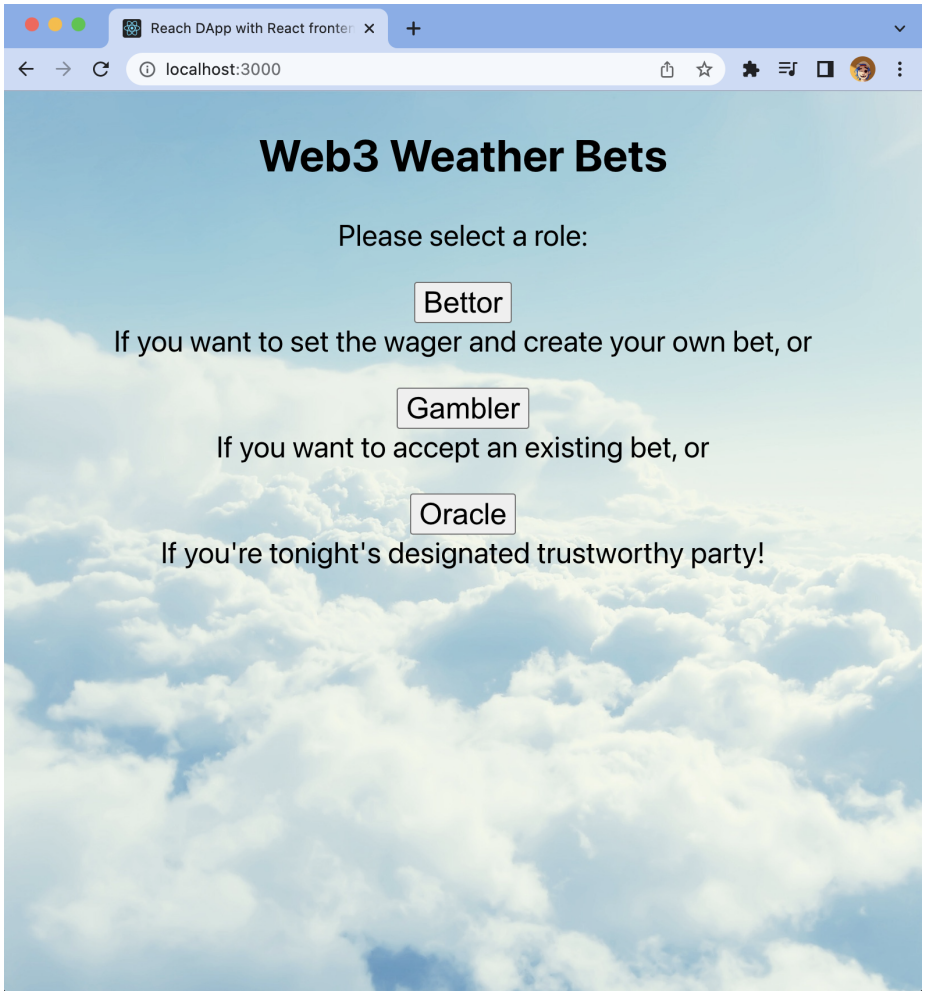
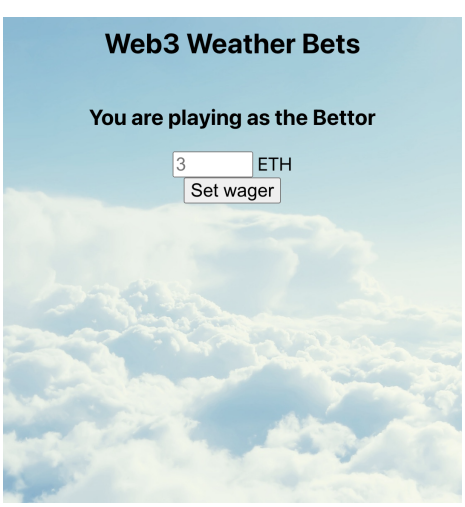
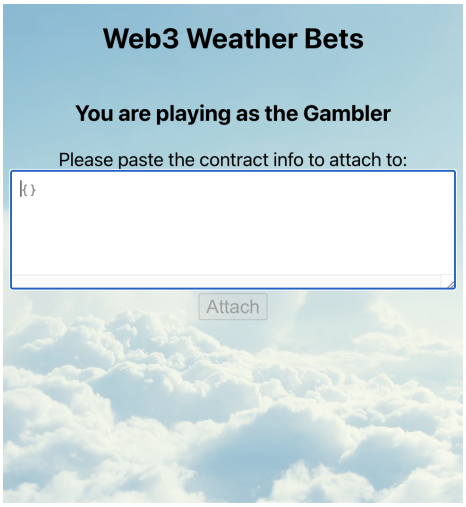


Figure 4: Home View

By selecting a role, users are then redirected to a role specific view:
The Oracle is brought to a similar view. However, since he actually doesn't



(a) Bettor

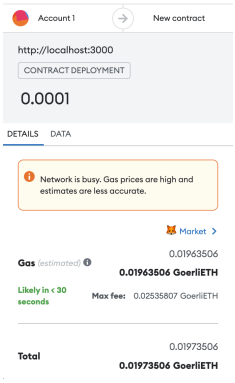


(b) Gambler

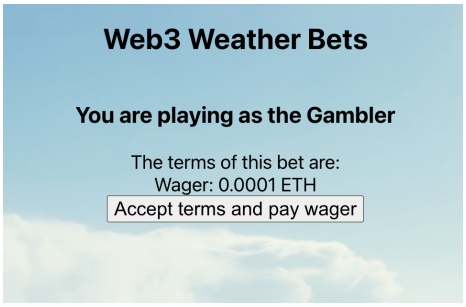
have to do anything (the oracle code runs passively), he'll just wait for the DApp to show him the result of his data request, and later the winner of the game. The next steps are:

- The Bettor deploys his contract and signs it (through Metamask)
- after that, the Gambler accepts the wager (and also signs the contract). This can only happen after the Oracle, too, joined the contract. This way the gambler can check if the oracle's address looks suspicious.

These actions happen through these views:



(a) Bettor



(b) Gambler

Finally, both users are brought to their weather selection dashboard: The load-

Figure 7: Selection screen

ing screens, as well as the final winner revelation views, are very similar to the home one, and only include a closure message.

6 Security measures

Reach encourages (actually, necessitates!) developers to carefully check for security breaches during contract development. These security measures can be classified in two categories:

- **Passive verification** - Every reach compilation (and therefore WWO) includes a formal verification using a satisfiability modulo theories (SMT)-based theorem prover. This feature verifies general properties that all programs should fulfill, such as never overflowing memory bounds, accessing uninitialized memories etc. It also verifies properties that are more specific to DApps, such as token linearity (avoidance of double spending) or any kind of mismatch between input and output volumes.

- **Security features** - Reach has a variety of built-in functions and required standard procedures that enforce smart contracts to act in safety. WWO adopts a number of these measures:

    1. In Reach, variables can be initialized to be **user specific**, by allowing users to declare them inside "steps" that nobody else aware of. When Bettors and Gamblers place their bets, this happens inside local steps to ensure that the other player can't see the bet. For an external entity to have access to a variable's content, it is necessary that its creator Declassifies it, and then **Publishes** it. These are distinct steps and must be performed in this order.
    2. When the Gambler accepts his bet, the contract already has information regarding the Bettor's chosen weather. Since this piece of knowledge could potentially be used from the Gambler to make smarter bets, a cryptographic commitment scheme is implemented to keep it fair. The Bettor creates a **commitment** based on his bet, and a **salt** for the system to check what his bet was at the right time. This way, when the Gambler attaches to the smart contract, he can only see an encrypted commitment.
    3. To double check the previous point, and future similar ones, after users interact with the systems to place their bets, reach's **unknowable** function is used to ensure that their choice is private.
    4. Once all players accept the terms and attach to the contract, after the Oracle retreives the real world data (but before the outcome of the bet is revealed) the Bettor publishes his salt and bet, and his (previously published) commitment is compared with his actual bet. This is the second (and last) part of the cryptographic commitment scheme mentioned earlier in this section.

6.1 Weathers

In order to retreive weather data, WWO leans on [2]. These API's return a very detailed report of the live weather in a number of cities. The accuracy of their sensors enable them to classify weathers into almost 40 categories:

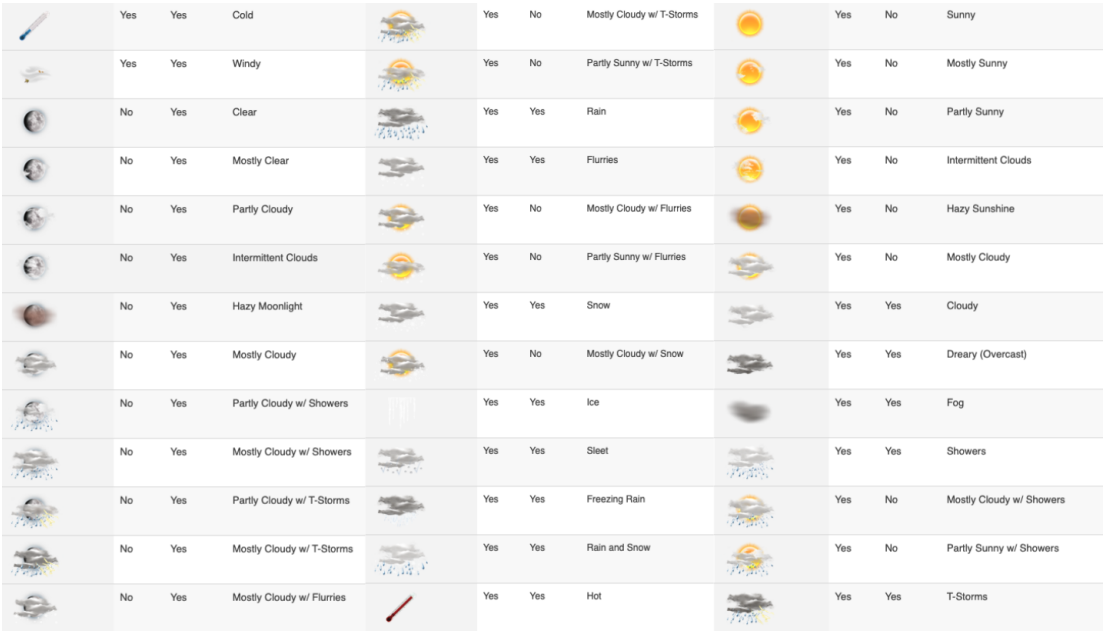


Figure 8: Possible Weather icons

6.2 Game Rules

Since there are many options here, it clearly looks like anyone would have a hard time guessing what the exact weather is like. Thankfully, [2] Ranks weathers (1, Sunny) to (39, T-Storms). WWO compares the bets made by Bettors and Gamblers to the one retrieved by the Oracle and declares which one is closer. This leads to 4 possible scenarios:

- The Bettor's guess is closer to the retrieved weather. The Bettor wins.
- The Gambler's guess is closer to the retrieved weather. The Gambler wins.
- The two guesses are equally distant from the answer. In this case, the Bettor wins. Beware, gamblers!
- One of the two parties fails to place his/her bet within a certain time limit (default: 5 minutes). In this case, the contract times out and the wagers are returned.

7 Conclusions

The goal of this project was to show an implementation of a simple, centralized oracle built using a novel, powerful technology. While the shown interfaces and execution are Ethereum-specific, the code perfectly works for Algorand as well: it is only necessary to switch the connector before the execution. The DApp should actually work for Conflux too, but it hasn't been tested yet. However, there are a number of limitation to this work's applicability as it is right now:

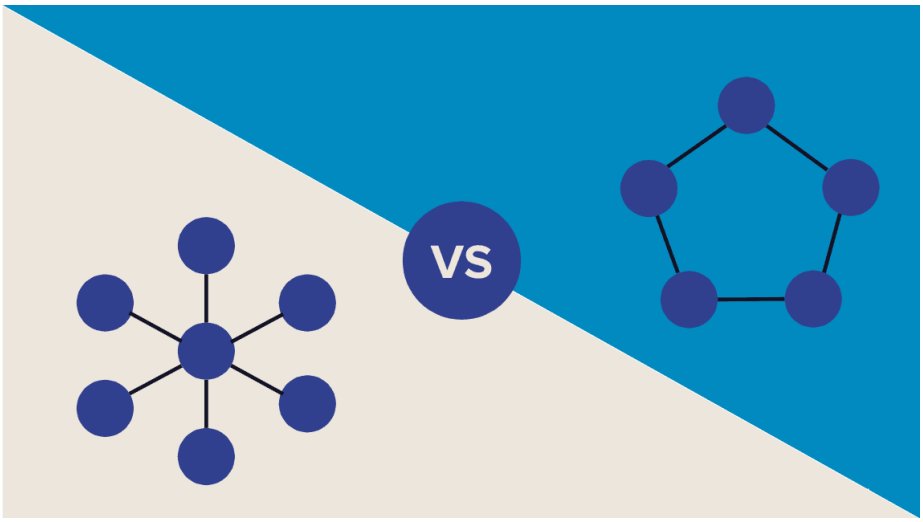
7.1 Known Limitations

- Of course, the first limitation is within the Oracle itself: It is centralized. While I'm working on a decentralized version for my thesis, I couldn't quite wrap it up yet. This is a major limitation, as the decentralization property we strive to achieve with blockchain is falling apart.
- The contract information still needs some kind of external communication for the Bettor to succesfully share it with the Gambler and the Oracle.
- At the time of writing, it is only possible for users to bet on the weather in Paris, and right now. This is easily editable by tweaking the code.
- The "fundAccount" feature when initializing the program isn't available in ETH devnets at the time of writing (It was up to december 2022).
- The weather could be checked before betting

7.2 For Future Work

I am still working on this project. To enhance it, I plan to:

- Give users the possibility of choosing a variety of cities to bet over
- Have the bettor choose when the weather must be detected. Obviously, this fixed point in time would also be shared with the gambler. This would allow the system to make more realistic bets: while it could be argued that the Gambler could simply check the weather before making his bet, both players could, at most, rely on predictions if they were forced to bet over the future weather.
- My not-so-secret dream is to achieve decentralization! At the time of writing, i'm working on a version with a fixed number (5) of oracle participants who consensually fetch and deliver data to the smart contracts.



8 References

- **[1]** Mühlbehrger, Bachhofner, Ferrer, Di Ciccio, Weber, Wöhrer and Zdun: Foundational Oracle Patterns: Connecting Blockchain to the Off-chain World. https://doi.org/10.48550/arXiv.2007.14946
- **[2]** Accuweather API's: https://developer.accuweather.com/apis