COMP6452 Software Architecture for Blockchain Applications
Week 4 Tutorial

David Zhang
d.zhang@unsw.edu.au

Yue Liu
yue.liu13@unsw.edu.au

Term 2, 2021

Hands-on sub-session

This part we will write a simple oracle that grabs weather information to the blockchain smart contract. You should learn how to interact with smart contracts using web3.js, and how to design a reusable oracle.

Install prerequisites

- Download and install NodeJS
- Download and install Truffle Ganache (Ganache-CLI or GUI version)

Initialize repository

```
npm init -y # init NodeJS repo
```

```
wget https://raw.githubusercontent.com/github/gitignore/master/Node.gitignore \
-O .gitignore # create gitignore
```

```
npm install typescript --save-dev # add typescript dev dep
```

```
npm install @types/node --save-dev # add typescript node dev dep
```

```
npx tsc --init --rootDir src --outDir build \
--esModuleInterop --resolveJsonModule --lib es6 \
--module commonjs --allowJs true --noImplicitAny true # config typescript
```

```
npm install web3 && npm install solc && npm install axios
```

```
mkdir src && touch src/index.ts # create source code entry
```

```
mkdir smart_contracts # create smart contract entry
```

After running all these commands, the directory should look like this:

```
package.json
package-lock.json
tsconfig.json
src/
  index.ts
smart_contracts/
node_modules/
  ...
```

The program can be built and executed with

```
npx tsc && node build/index.js
```

Write a smart contract which needs an oracle

We want to have a contract indicating whether people need to wear coats or not based on the local temperature.

The contract would be:

```
contract CoatIndicatorMelbourne {
    bool needToWearCoat = false;

    function update() public {
        needToWearCoat = getTemperature("Melbourne") < 20;
    }
}
```

However, we don't have getTemperature(string) function. We need to grab the temperature which is outside the blockchain.

How to achieve this?

- Periodically upload temperatures onto the blockchain, and smart contract query the temperature storage contract
- Periodically upload temperatures onto the blockchain, and smart contract subscribe the uploading
- Smart contract asynchronously pull data from outside world

Which one to choose can be based on the use case. If a lot of smart contracts will query for temperatures of same locations, then the first one or second one would be ideal. If the information is more customized and not reusable, e.g. a RNG (random number generator), then use the third one. This tutorial will demonstrate the third method.

The process is divided into two phases: request phase and a response callback phase. In request phase, the contract emits an event with city name, and the event is received by the blockchain listener. Then listener reply with temperature of the city by calling a function.

```
contract CoatIndicator {
    bool public needToWearCoat = false;

    event temperatureRequest(string city);
    function requestTemperature(string memory city) private {
        emit temperatureRequest(city);
    }

    function requestPhase() public {
        requestTemperature("Melbourne");
    }

    function responsePhase(int256 temperature) public {
        needToWearCoat = temperature < 20;
    }
}
```

Build a blockchain listener

A blockchain listener is needed to listen for the event and reply if by invoking function.

Start Ganache server

Simply open Ganache and choose either QuickStart or New Workspace. 10 accounts will be automatically generated with each has a balance of 100 ETH.

Enable asynchronous calls

```
(async () => {
})();
```

Start Web3

```
import Web3 from "web3";

const web3Provider = new Web3.providers.WebsocketProvider("ws://localhost:7545");
const web3 = new Web3(web3Provider);
```

Add an existing account

```
// account_pri_key is an account generated by Ganache
let account = web3.eth.accounts.wallet.add("0x" + account_pri_key);
```

Compile smart contracts

Put your smart contracts under directory `smart_contracts`

```
const fs = require("fs");
const solc = require("solc");

function findImports(importPath: string) {
    try {
        return {
```

```
            contents: fs.readFileSync(`smart_contracts/${importPath}`, "utf8")
        };
    } catch (e) {
        return {
            error: e.message
        };
    }
}

function compileSols(solNames: string[]): any {
    interface SolCollection { [key: string]: any };
    let sources: SolCollection = {};
    solNames.forEach((value: string, index: number, array: string[]) => {
        let sol_file = fs.readFileSync(`smart_contracts/${value}.sol`, "utf8");
        sources[value] = {
            content: sol_file
        };
    });
    let input = {
        language: "Solidity",
        sources: sources,
        settings: {
            outputSelection: {
                "*": {
                    "*": ["*"]
                }
            }
        }
    };
    let compiler_output = solc.compile(JSON.stringify(input), {
        import: findImports
    });
    let output = JSON.parse(compiler_output);
    return output;
}

let compiled = compileSols(["example"]);
```

Deploy a smart contract

```
import { Contract, DeployOptions } from "web3-eth-contract";

let contract_instance: Contract;
let gasPrice: string;
let contract = new web3.eth.Contract(compiled.contracts["example"]["CoatIndicator"].abi,
    undefined, {
        data: "0x" + compiled.contracts["example"]["CoatIndicator"].evm.bytecode.object
    });

await web3.eth.getGasPrice().then((averageGasPrice) => {
    gasPrice = averageGasPrice;
}).catch(console.error);

// assume account balance is sufficient
```

```
await contract.deploy({
    data: contract.options.data,
    arguments: [account.address]
} as DeployOptions).send({
    from: account.address,
    gasPrice: gasPrice!,
    gas: Math.ceil(1.2 * await contract.deploy({
        data: contract.options.data,
        arguments: [account.address]
    } as DeployOptions).estimateGas({
        from: account.address
    })),
}).then((instance) => {
    contract_instance = instance;
}).catch(console.error);

console.log(contract_instance!.options.address);
```

Listen for a smart contract event

```
contract_instance!.events["temperatureRequest(string)"]()
    .on("connected", function (subscriptionId: any) {
        console.log("listening on event temperatureRequest");
    })
    .on("data", async function (event: any) {
        let city = event.returnValues.city;

        /// TODO respond with temperature by calling responsePhase(int256)
    })
    .on("error", function (error: any, receipt: any) {
        console.log(error);
        console.log(receipt);
        console.log("error listening on event temperatureRequest");
    });
```

Invoke a smart contract function

```
contract_instance!.events["temperatureRequest(string)"]()
    .on("connected", function (subscriptionId: any) {
        console.log("listening on event temperatureRequest");
    })
    .on("data", async function (event: any) {
        let city = event.returnValues.city;

        let temperature = "-25";

        // assume account balance is sufficient

        try {
            contract_instance.methods["responsePhase(int256)"](temperature).send({
                from: account.address,
                gasPrice: gasPrice!,
                gas: Math.ceil(1.2 * await contract_instance.methods["responsePhase(int256)"]
                (temperature).estimateGas({ from: account.address })),
            }).then(function (receipt: any) {
```

```
            return receipt;
        }).catch((err: any) => {
            console.error(err);
        });
    } catch (e) {
        console.log(e);
    }
})
.on("error", function (error: any, receipt: any) {
    console.log(error);
    console.log(receipt);
    console.log("error listening on event temperatureRequest");
});
```

Grab weather information from online API

An online weather API https://goweather.herokuapp.com/weather/Melbourne is used. You may use other more trustworthy information provider.

```
contract_instance!.events["temperatureRequest(string)"]()
    .on("connected", function (subscriptionId: any) {
        console.log("listening on event temperatureRequest");
    })
    .on("data", async function (event: any) {
        let city = event.returnValues.city;

        let temperature = await axios.get(`https://goweather.herokuapp.com/weather/${city}`)
            .then(async function (response: any) {
                return response?.data?.temperature?.replace(/[^0-9-\.]/g, "");
            })
            .catch(function (error: any) {
                console.log(error);
            });

        if (!parseInt(temperature)) {
            console.log("invalid temperature");
            return;
        }

        // assume account balance is sufficient

        try {
            contract_instance.methods["responsePhase(int256)"](temperature).send({
                from: account.address,
                gasPrice: gasPrice!,
                gas: Math.ceil(1.2 * await contract_instance.methods["responsePhase(int256)"]
                (temperature).estimateGas({ from: account.address })),
            }).then(function (receipt: any) {
                return receipt;
            }).catch((err: any) => {
                console.error(err);
            });
        } catch (e) {
            console.log(e);
        }
    })
    .on("error", function (error: any, receipt: any) {
```

```
        console.log(error);
        console.log(receipt);
        console.log("error listening on event temperatureRequest");
    });
```

Now the listener can interact with the contract and push information.

Refactor the smart contract and listener

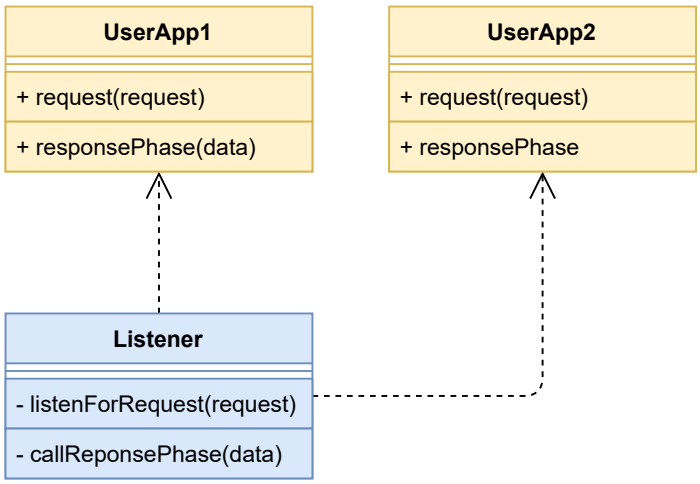

Figure 1: Class diagram before refactoring

While the above code snippets show a workable solution as in figure 1, it has several limitations (and even more):

- Listener must listen on every user smart contract
- Listener must know the ABI of every user smart contract
- Have to copy-paste many duplicate parts when writing a new smart contract
- Smart contract has no control over the allowed sender of the response phase
- Smart contract is recompiled and redeployed every time running the listener

After the refactoring, the oracle and user app are decoupled: figure 2.

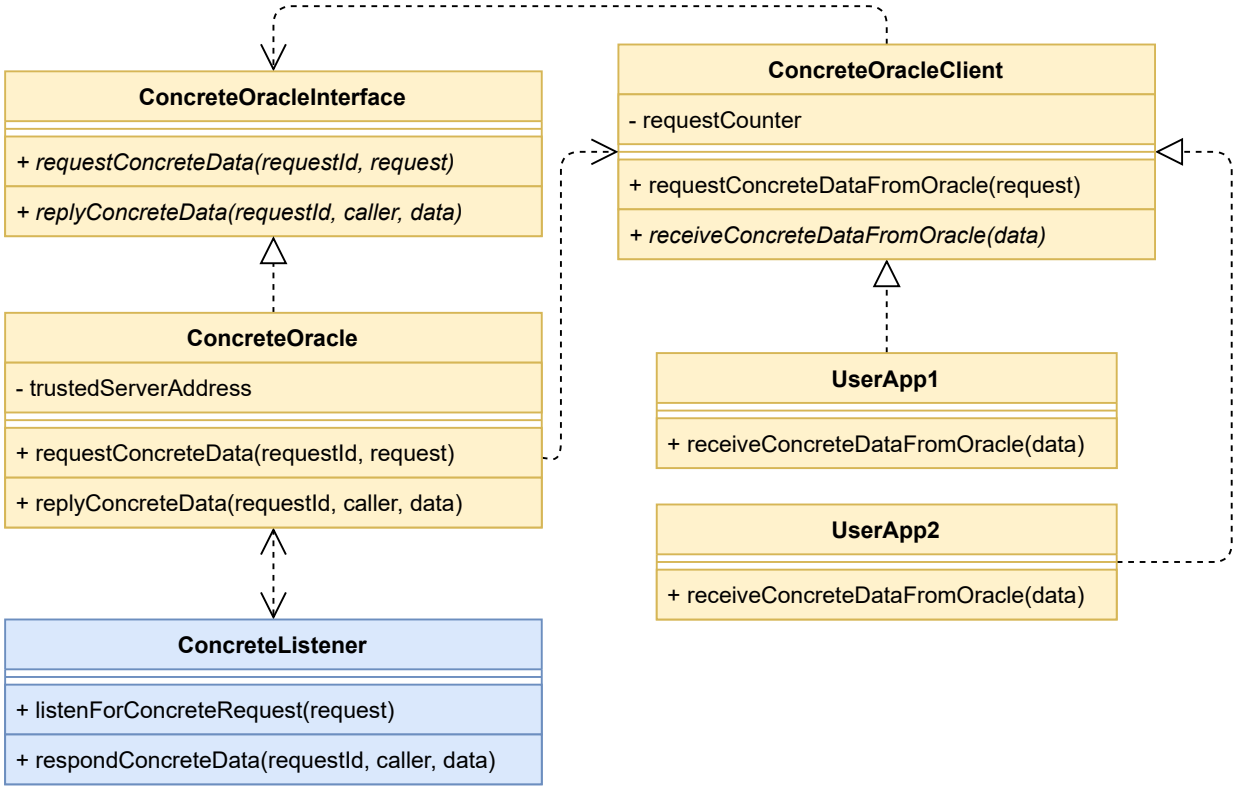

Figure 2: Class diagram refactored

Why a request id is introduced?

A request id is needed to let the client distinguish the callback of its different requests.

The dependencies between oracle and oracle client can be further decoupled: figure 3. However, generic types are not supported by Solidity, so the concrete classes are not strictly constrained by compilation. By the way, the concrete class can also be from oracle part, to move the data related operations under oracle instead of client.

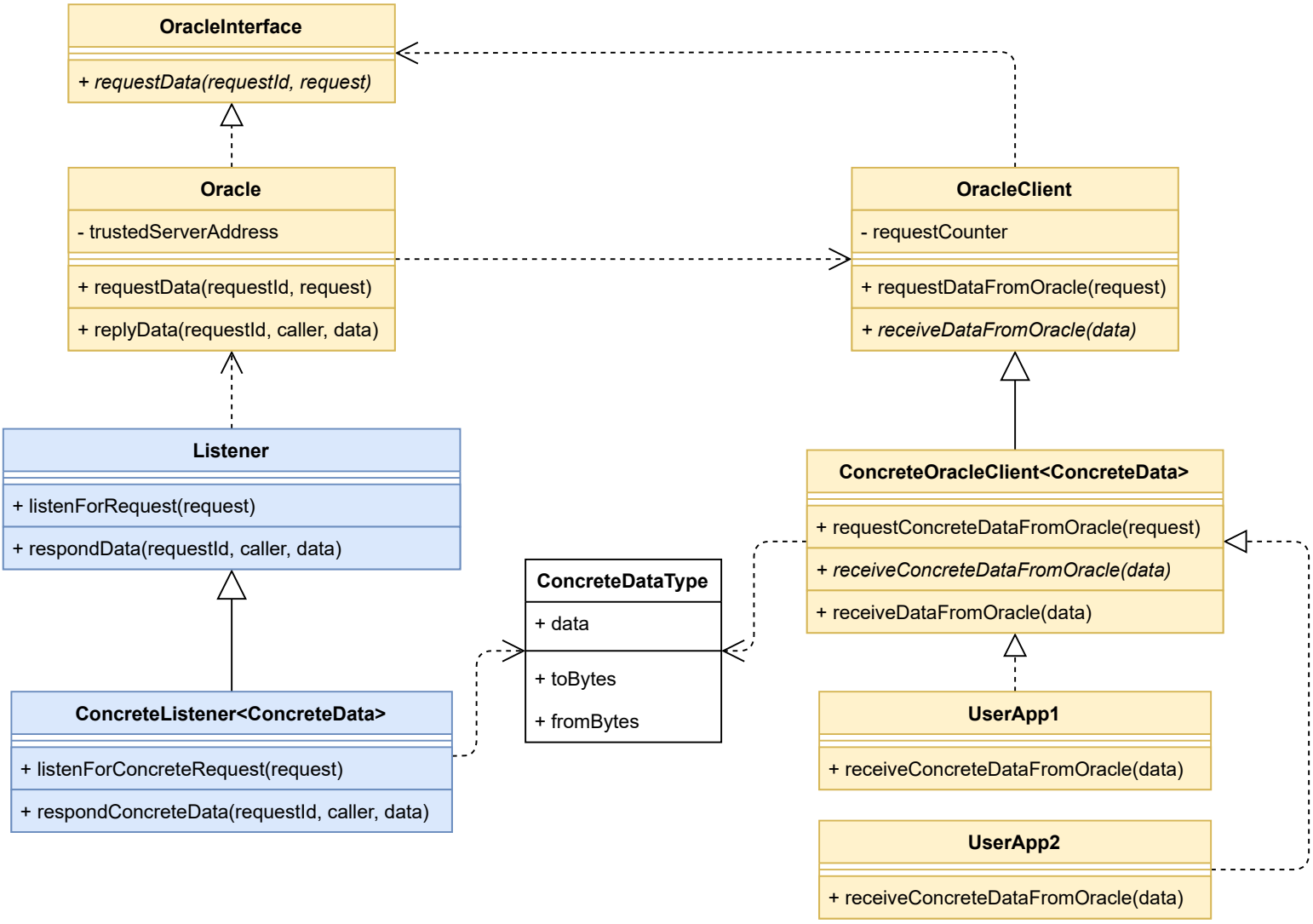

Figure 3: Class diagram generic

Example repositories

original version: UNSW GitLab repository
reactored generic version: UNSW GitLab repository, GitHub repository

Theory sub-session

Qinghua Lu, Xiwei Xu, Yue Liu, Ingo Weber, Liming Zhu, and Weishan Zhang. "uBaaS: A unified blockchain as a service platform". eng. In: *Future generation computer systems* 101 (2019), pp. 564–575. ISSN: 0167-739X. URL: https://primoa.library.unsw.edu.au/permalink/f/1uqach6/TN_cdi_arxiv_primary_1907_13293

Why we write academic papers and why the final exam is about writing a review on the paper?

One of many reasons: academic paper is very useful for presenting ideas. People may think that academic papers are written in a structure that contains a lot of redundant and meaningless content. However, they are not. The format of academic papers has been developed through the history by many and adopting this format can give the reader sufficient information about the research. Styles and formats of papers may also vary from different subjects, institutions and authors, and developing a writing style can be helpful for conveying the research as well as having a greater impact. For example, one popular style is telling stories: you might have heard of a lot of stories in computer science, such as Dining Philosophers Problem, Byzantine Generals' Problem, Baker Algorithm, Postman Problem, etc.

Similar to having so many types of diagrams we mentioned in last tutorial, papers are not to express an idea in a complicated way, but to make it clear for the readers. This competence is not only important for researchers, but also for software engineers. You may find the documentations of some software frameworks are easy to read while others not. Moreover, the ability to understand the document is actually essential for engineers. Some may find them end up having implemented a system that is not identical to the system described by the documentation from software designers.

A few questions you may need to think about when reading a paper in the field of software architecture:

- What is the problem that the research aims to solve?
- Why this problem is significant and what is the causes of the problem?
- What is the architecture design of the solution?
- Compared with previous works, what is the novelty of this research?
- How much of the problem is solved by the research?
- Are there any design trade-off made? Any new problem introduced by the solution?
- Are there any improvements can be done on this research?