Zadanie 1 – MyBlockchain

V zadaní 1 bude naším cieľom vytvoriť vlastnú blockchainovú sieť, ktorá bude zvládať základné dopyty, zaradí transakcie do blokov, jednotlivé bloky do reťaze blokov a dosiahne konsenzus medzi účastníkmi. Odporúčam rozdeliť si prácu na projekte do 3 fáz po jednotlivých týždňoch.

Fáza 1 – Jednoduchý coin

V jednoduchom coine centrálna autorita FIIT prijíma transakcie od používateľov. Vašou úlohou bude implementovať logiku používanú FIIT na spracovávanie transakcií a vytvorenie ledgeru. FIIT zgrupuje transakcie do časových radov alebo blokov. V každom bloku dostane FIIT zoznam transakcií, tieto transakcie skontroluje a zverejní zoznam platných transakcií.

Pozor, že jedna transakcia sa môže odkazovať na inú v rovnakom bloku. Tiež sa môže stať, že medzi transakciami prijatými FIIT v jednom bloku sa nachádza viac ako jedna transakcia míňajúca rovnaký output. Toto by bol samozrejme double-spending a tým pádom transakcia neplatná. Znamená to, že transakcie nemôžu byť kontrolované izolovane; je to problém ako si vybrať podmnožinu transakcií, ktoré sú *spoločne* platné.

Máte k dispozícii súbor `Transaction` obsahujúci rovnomennú triedu, ktorá predstavuje transakciu jednoduchého coinu a má vnútorné triedy `Transaction.Output` a `Transaction.Input`.

Transakčný výstup pozostáva z hodnoty a z verejného kľúča, na ktorý sa platí. Pre verejné kľúče používame štandardné PublicKey knižnice.

Transakčný vstup sa skladá z hash transakcie, ktorá obsahuje zodpovedajúci výstup, indexu tohto výstupu v danej transakcii (indexy sú jednoducho celé čísla začínajúce od 0) a digitálneho podpisu. Aby bol vstup platný, musí obsahovať podpis, ktorý musí byť platným podpisom nad aktuálnou transakciou s verejným kľúčom vo využívanom výstupe.

Podpísané raw údaje sa získavajú metódou `getRawDataToSign(int index)`. Na overenie podpisu použijete metódu `verifySignature()` zahrnutú v súbore `Crypto`:

public static boolean verifySignature(PublicKey pubKey, byte[] message, byte[] signature)

Táto metóda vezme verejný kľúč, správu a podpis a vráti true jedine ak podpis správne overí správu s verejným kľúčom pubKey.

Vám je poskytnutý iba kód na overenie podpisov a to je všetko, čo potrebujete. Výpočet podpisov sa vykonáva mimo Transaction pomocou entity, ktorá pozná príslušné súkromné kľúče.

Transakcia pozostáva zo zoznamu vstupov, zoznamu výstupov a jedinečného ID (pozri `getRawTx()`). Trieda tiež obsahuje metódy na pridanie a odstránenie vstupu, pridanie výstupu, výpočet digestu na podpísanie/hash, pridanie podpisu na vstup a vypočítanie a uloženie hash transakcie po pridaní všetkých vstupov/výstupov/podpisov.

Taktiež máte k dispozícii súbor `UTXO`, ktorý predstavuje neminutý výstup transakcie. UTXO obsahuje hash transakcie, z ktorej pochádza, ako aj index v rámci tejto transakcie. Do UTXO boli zahrnuté funkcie `equals`, `hashCode` a `compareTo`, ktoré umožňujú testovanie rovnosti a porovnanie medzi dvoma UTXO na základe ich indexov a obsahu ich polí txHash.

Ďalej máte súbor `UTXOPool`, ktorý predstavuje aktuálnu množinu UTXO a obsahuje mapu z každého UTXO na jeho zodpovedajúci výstup transakcie. Tento súbor obsahuje konštruktory na vytvorenie nového prázdneho UTXOPool alebo kópie daného UTXOPool a metódy na pridanie a odstránenie UTXO z poolu, získanie výstupu zodpovedajúceho danému UTXO, skontrolovanie, či UTXO je v poole, a získanie zoznamu všetkých UTXO v poole.

Vašou úlohou je vytvorenie súboru `HandleTxs.java`, ktorého základná štruktúra je k dispozícii a ktorý implementuje nasledujúce API:

```
public class HandleTxs {

    /**
     * Vytvorí verejný ledger, ktorého aktuálny UTXOPool (zbierka nevyčerpaných
     * transakčných výstupov) je {@code utxoPool}. Malo by to vytvoriť ochrannú kópiu
     * utxoPool pomocou konštruktora UTXOPool (UTXOPool uPool).
     */
    public HandleTxs(UTXOPool utxoPool);

    /**
     * @return true, ak
     * (1) sú všetky výstupy nárokované {@code tx} v aktuálnom UTXO pool,
     * (2) podpisy na každom vstupe {@code tx} sú platné,
     * (3) žiadne UTXO nie je nárokované viackrát,
     * (4) všetky výstupné hodnoty {@code tx}s sú nezáporné a
     * (5) súčet vstupných hodnôt {@code tx}s je väčší alebo rovný súčtu jej
     *     výstupných hodnôt; a false inak.
     */
    public boolean txIsValid(Transaction tx);

    /**
     * Spracováva každú epochu prijímaním neusporiadaného radu navrhovaných
     * transakcií, kontroluje správnosť každej transakcie, vracia pole vzájomne
     * platných prijatých transakcií a aktualizuje aktuálny UTXO pool podľa potreby.
     */
    public Transaction[] txHandler(Transaction[] possibleTxs);
}
```

Vaša implementácia `txHandler()` by mala vrátiť vzájomne platnú množinu transakcií maximálnej veľkosti (takú, ktorú nie je možné zväčšiť jednoduchým pridaním ďalších transakcií). Nemusíte počítať množinu maximálnej veľkosti všeobecne (takú, pre ktorú neexistuje väčšia vzájomne platná množina transakcií).

Na základe transakcií, ktoré sa rozhodla prijať, `txHandler()` by tiež mal aktualizovať svoj interný UTXOPool tak, aby odrážal aktuálnu množinu nevyčerpaných transakčných výstupov, aby budúce volania na `txHandler()` a `txIsValid()` boli schopné správne spracovať/overiť transakcie, ktoré si nárokujú výstupy z transakcií prijatých pri predchádzajúcom volaní na `txHandler()`.

Extra bod: 1 bod naviac získajú tí, ktorí vytvoria druhý súbor s názvom `MaxFeeHandleTxs.java`, ktorého metóda `txHandler()` vyhľadá množinu transakcií s maximálnymi celkovými poplatkami za transakcie – tj. maximalizuje súčet všetkých transakcií v množine (súčet vstupných hodnôt - súčet výstupných hodnôt).

Fáza 2 – Dôvera a konsenzus

V tejto časti navrhnete a implementujete algoritmus distribuovaného konsenzu s grafom vzťahov „dôvery“ medzi uzlami. Toto je alternatívna metóda odolávania sybil útokom a dosiahnutia konsenzu; má výhodu v tom, že „neplytváte“ elektrickou energiou, ako to robí proof of work.

Uzly v sieti sú buď dôveryhodné, alebo podvodné. Napíšete triedu `TrustedNode` (implementuje poskytované rozhranie `Node`), ktorá definuje správanie každého z dôveryhodných uzlov. Sieť je orientovaný náhodný graf, kde každá hrana predstavuje vzťah dôvery. Napríklad, ak existuje hrana A → B, znamená to, že uzol B počúva transakcie vysielané uzlom A. Hovoríme, že B je nasledovník A a A je nasledovaný B.

Každý uzol by mal uspieť pri dosahovaní konsenzu so sieťou, v ktorej sú jeho peermi ďalšie uzly používajúce rovnaký kód. Váš algoritmus by mal byť navrhnutý tak, aby sa sieť uzlov prijímajúca rôzne sety transakcií mohla dohodnúť na sete, ktorý akceptujú. Máte k dispozícii triedu `Simulation`, ktorá generuje náhodný graf dôvery. Nastavte si počet kôl, počas ktorých budú vaše uzly v každom kole vysielať svoj návrh svojim nasledovníkom a na konci kola by mali dosiahnuť konsenzus o tom, na ktorých transakciách by sa mali dohodnúť.

Každý uzol dostane svoj zoznam sledovaných osôb prostredníctvom boolovského poľa, ktorého indexy zodpovedajú uzlom v grafe. Hodnota „true“ v indexe i označuje, že uzol i je sledovaný, inak „false.“ Tento uzol tiež získa zoznam transakcií (zoznam ponúk), ktoré môže vysielať svojim sledujúcim. Za vytvorenie počiatočných transakcií/zoznamu návrhov nebudete

zodpovedať. Predpokladajme, že všetky transakcie sú platné a že neplatné transakcie nemožno vytvoriť. Poskytnutá trieda Node má nasledujúce API:

```
public interface Node {

    // POZNÁMKA: Node je rozhranie a nemá konštruktor.
    // Vaša trieda TrustedNode.java však vyžaduje 4-argumentový
    // konštruktor, ako je definované v Simulation.java
    // Tento konštruktor dáva vášmu uzlu informácie o simulácii
    // vrátane počtu kôl, pre ktoré bude bežať.

    /**
     * {@code followees [i]} je pravda iba ak tento uzol nasleduje uzol {@code i}
     */
    void followeesSet(boolean[] followees);

    /** inicializovať návrhový zoznam transakcií */
    void pendingTransactionSet(Set<Transaction> pendingTransactions);

    /**
     * @return návrhy, ktoré pošlem mojim nasledovníkom. Pamätajme: Po finálovom
     *         kole sa správanie {@code getProposals} zmení a malo by vrátiť
     *         transakcie, pri ktorých bol dosiahnutý konsenzus.
     */
    Set<Transaction> followersSend();

    /** príjmy kandidátov z iných uzlov */
    void followeesReceive(Set<Candidate> candidates);
}
```

Pri testovaní sa môžu uzly, na ktorých beží váš kód, stretnúť s množstvom (až 45%) škodlivých uzlov, ktoré nespolupracujú s vaším konsenzuálnym algoritmom. Uzly vášho návrhu by mali byť schopné vydržať čo najviac škodlivých uzlov a stále dosiahnuť konsenzus. Škodlivé uzly môžu mať svojvoľné správanie. Napríklad, škodlivý uzol môže:

- byť funkčne mŕtvy a nikdy v skutočnosti nevysielať žiadne transakcie,
- neustále vysiela svoju vlastnú skupinu transakcií a nikdy neprijíma transakcie, ktoré sú mu dané,
- meniť správanie medzi kolami, aby sa zabránilo detekcii.

K dispozícii máte nasledujúce súbory:

`Node.java`	základné rozhranie pre vašu triedu TrustedNode
`TrustedNode.java`	Kostra triedy TrustedNode. Váš kód by ste mali vyvinúť na základe šablóny, ktorú tento súbor poskytuje.
`Candidate.java`	jednoduchá trieda na opísanie kandidátskych transakcií, ktoré váš uzol dostane

`ByzantineNode.java`	veľmi jednoduchý príklad škodlivého uzla
`Simulation.java`	základný generátor grafov, ktorý môžete použiť na spustenie vlastných simulácií s rôznymi parametrami grafu a na otestovanie triedy TrustedNode
`Transaction.java`	trieda Transaction, pričom transakcia je iba obalom okolo jedinečného identifikátora (t. j. platnosť a sémantika transakcií sú pre toto priradenie irelevantné)

Graf uzlov bude mať nasledujúce parametre:

- pravdepodobnosť párovej konektivity náhodného grafu: napr. {.1, .2, .3},
- pravdepodobnosť, že uzol bude nastavený ako podvodný: napr. {.15, .30, .45},
- pravdepodobnosť, že bude oznámená každá z počiatočných platných transakcií: napr. {.01, .05, .10},
- počet kôl v simulácii napr. {10, 20}.

Zamerajte sa na vývoj robustnej triedy TrustedNode, ktorá bude pracovať so všetkými kombináciami parametrov grafu. Na konci každého kola váš uzol uvidí zoznam transakcií, ktoré mu boli vysielané.

Každý test sa meria na základe:

- Ako veľká skupina uzlov dosiahla konsenzus. Sada uzlov sa počíta iba ako majúca dosiahnutý konsenzus, ak majú všetky rovnaký zoznam transakcií.
- Veľkosť súboru, na ktorom sa dosahuje konsenzus. Mali by ste sa usilovať o dosiahnutie konsenzu čo najviac transakcií.
- Čas vykonania, ktorý by mal byť rozumný (ak kód trvá príliš dlho, na hodnotení sa to odrazí).

Niektoré rady:

- Váš uzol nebude poznať topológiu siete a mal by sa snažiť pracovať všeobecne. To znamená, že si musí uvedomiť, ako rôzna topológia môže mať vplyv na to, ako chceme zahrnúť transakcie podľa obrazu konsenzu.
- Váš kód TrustedNode môže predpokladať, že všetky transakcie, ktoré vidí, sú platné – simulácia pošle iba platné transakcie (počiatočné aj medzi kolami) a iba simulačný kód má schopnosť vytvárať platné transakcie.

- Ignorujte patologické prípady, ktoré sa vyskytujú s extrémne nízkou pravdepodobnosťou, napríklad keď sa dôveryhodný uzol spáruje iba so škodlivými uzlami. Testovacie prípady by takéto scenáre nemali mať.

Fáza 3 – blockchain

V tejto fáze budete implementovať uzol, ktorý je súčasťou distribuovaného konsenzuálneho protokolu založeného na blokových reťazcoch. Váš kód bude konkrétne prijímať prichádzajúce transakcie a bloky a udržiavať aktualizovaný blockchain.

Poskytnuté súbory:

`Block.java`	Ukladá štruktúru dát bloku.
`BlockHandler.java`	Používa BlockChain.java na spracovanie novo prijatého bloku, vytvorenie nového bloku alebo na spracovanie novo prijatej transakcie.
`ByteArrayWrapper.java`	Súbor obslužného programu, ktorý vytvára wrapper pre bajtové polia, takže ich možno použiť ako kľúč v hašovacích funkciách.
`Transaction.java`	Podobné ako v prípade Transaction.java, ako je uvedená vo Fáze 1, s výnimkou zavedenia funkcií na vytvorenie transakcie coinbase. Prezrite si konštruktor Block.java a uvidíte, ako sa vytvára coinbase.
`TransactionPool.java`	Implementuje pool transakcií, ktorý sa vyžaduje pri vytváraní nového bloku.
`UTXO.java`	Z fázy 1.
`UTXOPool.java`	Z fázy 1.

Vytvorte verejnú funkciu `UTXOPoolGet()` v súbore `HandleTxs.java`, ktorý ste vytvorili vo fáze 1 a skopírujte súbor do svojho kódu pre fázu 3.

Trieda Blockchain je zodpovedná za udržiavanie blokového reťazca. Pretože celý blockchain môže mať obrovskú veľkosť, mali by ste si ukladať iba posledné bloky. Presné číslo, ktoré sa má uložiť, závisí od vášho návrhu, pokiaľ budete môcť implementovať všetky funkcie API.

Pretože môže byť (viac) forkov, bloky tvoria skôr strom ako zoznam. Váš návrh by to mal brať do úvahy. Musíte udržiavať UTXO pool zodpovedajúci každému bloku, nad ktorým by sa mohol vytvoriť nový blok.

Súbor, ktorý sa má upraviť:

`BlockChain.java`

```java
// Blockchain by mal na uspokojenie funkcií udržiavať iba obmedzené množstvo uzlov
// Nemali by ste mať všetky bloky pridané do blockchainu v pamäti
// pretože by to spôsobilo pretečenie pamäte.

public class BlockChain {
    public static final int CUT_OFF_AGE = 12;

    /**
     * vytvor prázdny blockchain iba s Genesis blokom. Predpokladajme, že
     * {@code genesisBlock} je platný blok
     */
    public BlockChain(Block genesisBlock) {
        // IMPLEMENTOVAŤ
    }

    /** Získaj maximum height blok */
    public Block getMaxHeightBlock() {
        // IMPLEMENTOVAŤ
    }

    /** Získaj UTXOPool na ťaženie a nový blok na vrchu max height blok */
    public UTXOPool getMaxHeightUTXOPool() {
        // IMPLEMENTOVAŤ
    }

    /** Získaj pool transakcií na vyťaženie nového bloku */
    public TransactionPool getTransactionPool() {
        // IMPLEMENTOVAŤ
    }

    /**
     * Pridaj {@code block} do blockchainu, ak je platný. Kvôli platnosti by mali
     * byť všetky transakcie platné a blok by mal byť na
     * {@code height > (maxHeight - CUT_OFF_AGE)}.
     *
     * Môžete napríklad vyskúšať vytvoriť nový blok nad blokom Genesis (výška bloku
     * 2), ak height blockchainu je {@code <=
     * CUT_OFF_AGE + 1}. Len čo {@code height > CUT_OFF_AGE + 1}, nemôžete vytvoriť
     * nový blok vo výške 2.
     *
     * @return true, ak je blok úspešne pridaný
     */
    public boolean blockAdd(Block block) {
        // IMPLEMENTOVAŤ
    }

    /** Pridaj transakciu do transakčného poolu */
    public void transactionAdd(Transaction tx) {
        // IMPLEMENTOVAŤ
    }
}
```

Predpoklady a rady:

- Nový Genesis blok sa nebude ťažiť. Ak dostanete blok, ktorý o sebe tvrdí, že je blokom genesis (rodič je nulový hash), vo funkcii `blockAdd(Block b)`, môžete vrátiť hodnotu false.
- Ak je v rovnakej výške viac blokov, vráťte najstarší blok vo funkcii `getMaxHeightBlock()`.
- Pre jednoduchosť predpokladajme, že je k dispozícii coinbase transakcia bloku, ktorá sa môže minúť v nasledujúcom bloku ťaženom nad ňou (To je v rozpore so skutočným Bitcoin protokolom, keď pred jeho uplatnením existuje „dospelosť“ 100 potvrdení).
- Udržiavajte iba jeden globálny pool transakcií pre blockchain a pri prijímaní transakcií doň neustále pridávajte transakcie a odstráňte z neho transakcie, ak je prijatý alebo vytvorený nový blok. Je v poriadku, ak dôjde k zrušeniu niektorých transakcií počas reorganizácie blockchainu, tj. keď sa bočná vetva stane novou najdlhšou vetvou. Konkrétne transakcie prítomné v pôvodnej hlavnej vetve (a teda odstránené z poolu transakcií), ktoré však chýbajú v bočnej vetve, sa môžu stratiť.
- Hodnota coinbase je udržiavaná na konštantnej hodnote 6.25 bitcoinov, zatiaľ čo v skutočnosti sa znižuje na polovicu zhruba každé 4 roky.
- Pri kontrole platnosti novo prijatého bloku stačí iba skontrolovať, či sú transakcie z platnej množiny. Táto sada nemusí predstavovať maximálny možný počet transakcií. Tiež nemusíte robiť žiadne kontroly funkčnosti.

Extra bod: 1 bod naviac získajú tí, ktorí umožnia v blockchaine používať multisig transakcie a zároveň ich overenie v štandardnom procese ťažby nových blokov.

Odovzdanie

Dokumentáciu a zdrojový kód implementácie študent odovzdáva v elektronickom tvare do AISu v určenom termíne. Aj na toto zadanie je možné využiť Late days. Všetky kódy prejdú kontrolou originality, v prípade vysokej percentuálne zhody bude študentovi začaté disciplinárne konanie.

Každý program bude testovaný automatizovane pomocou unit testov. Podmienkou je aj aktívne odprezentovanie programu študentom na cvičení podľa harmonogramu.

Súčasťou riešenia je aj dokumentácia, ktorá musí obsahovať najmä:

a) formálne náležitosti ako sú titulná strana, číslovanie strán,
b) blokový návrh (koncepcia) fungovania riešenia,
c) navrhnutý mechanizmus komunikácie v rámci vlastnej blockchain siete,
d) vysvetlené jednotlivé fázy a ako sa s nimi študent popasoval,
e) používateľská príručka,
f) voľbu implementačného prostredia
g) iné záležitosti, ktorými sa študent chce pochváliť
h) záver a v ňom zhodnotenie vlastného príspevku a čo som sa naučil/a.

Hodnotenie

Celé riešenie - max. 15 bodov, z toho:

- max. 5 bodov za riešenie úlohy vo fáze 1; 15 unit testov každý za 0,33b
- max. 1 bod za riešenie úlohy vo fáze 2; 2 unit testy každý za 0,5b
- max. 6,75 bodov za riešenie úlohy vo fáze 3; 27 unit testov každý za 0,25b
- max. 2,25 bodu za výslednú dokumentáciu.

Bonusy

- max. 1 bod za MaxFeeHandleTxs vo fáze 1; 3 unit testy každý za 0,33b
- max. 1 bod za MultiSig transakcie vo fáze 3; 3 unit testy každý za 0,33b
- max. 2 body za GUI a vizualizačný nástroj, ktorý uľahčí interakciu so zadaním
- max. 5 bodov za prácu v jazyku Python a prerobenie všetkých zdrojových kódov vrátane unit testov na Python 3.