Project 3 - Block Chain

Thursday, December 17, 2020 9:36 PM

In this assignment you will implement a node that's part of a block-chain-based distributed consensus protocol. Specifically, your code will receive incoming transactions and blocks and maintain an updated block chain.

Download Starter Code Here

Files provided:

Block.java	Stores the block data structure.
BlockHandler.java	Uses BlockChain.java to process a newly received block, create a new block, or process a newly received transaction.
ByteArrayWrapper.java	A utility file which creates a wrapper for byte arrays such that it could be used as a key in hash functions. (See TransactionPool.java)
Transaction.java	This is similar to Transaction.java as provided in Project 1 except for introducing functionality to create a coinbase transaction. Take a look at Block.java constructor to see how a coinbase transaction is created.
TransactionPool.java	Implements a pool of transactions, required when creating a new block.
UTXO.java	From Project 1.
UTXOPool.java	From Project 1.
TxHandler.java	From Project 1 with some changes.

File to be modified:

BlockChain.java

The BlockChain class is responsible for maintaining a block chain. Since the entire block chain could be huge in size, you should only keep around the most recent blocks. The exact number to store is up to your design, as long as you're able to implement all the API functions.

Since there can be (multiple) forks, blocks form a tree rather than a list. Your design should take this into account. You have to maintain a UTXO pool corresponding to every block on top of which a new block might be created.

Assumptions and Guidelines:

- A new genesis block would not be mined. If you receive a block which claims to be a genesis block (parent is a null hash) in the `addBlock(Block b)` function, you can return false.
- If there are multiple blocks at the same height, return the oldest block in `getMaxHeightBlock()` function.
- Assume for simplicity, a coinbase transaction of a block is available to be used in the next block mined on top of it. (This is contrary to the actual Bitcoin protocol when there is a gap of 100 blocks only after which the coinbase transaction can be used).
- Maintain only one global Transaction Pool for the block chain and keep adding transactions to it on receiving transactions and keep removing transactions from it if a new block is received or created. This might cause some transactions to be lost. For example, a block is received on Chain A including transaction Tx1. We remove Tx1 from the transaction pool. Now suppose chain B offshoots chain A. Do not put Tx1 back in the pool, although ideally it should be put back in. This is to simplify your work as well as our work in testing. (Miners are not responsible for including transactions in the blocks. If a transaction is lost, it is the responsibility of the transaction owner to re-broadcast it in the network).
- The coinbase value is kept as constant in our entire block chain (= 25 bitcoins) whereas we know it changes every four years.
- When checking for validity of a newly received block, just checking if the transactions form a valid set is enough. The set need not be a maximum possible set of transactions. Also, you should not check for hash of the block to contain specific zeros (no proof of work here).

Download Autograder Here

An autograder is provided to you to self check the correctness of your program. Follow the following steps:

Step 1. Copy **BlockChain.java** into **grading** folder.

Step 2. While holding down **Windows key** press **R** (or click **Start**) and then type **cmd**. Press **Enter**.

Step 3. Move into the folder that contains the **project**-related classes by typing this command:

```
cd C:\grading
```

Step 4. Compile the classes with this command (**Replace ; with : for Linux or Mac OS**):

```
javac -cp blockChainGrader.jar;algs4.jar;rsa.jar;. DropboxTestBlockChain.java
```

Step 5. Finally, run the tests with this command:

```
java -cp blockChainGrader.jar;rsa.jar;. DropboxTestBlockChain
```

Download Autograder Here

An autograder is provided to you to self check the correctness of your program. Follow the following steps:

Step 1. Copy **BlockChain.java** into **grading** folder.

Step 2. While holding down **Windows key** press **R** (or click **Start**) and then type **cmd**. Press **Enter**.

Step 3. Move into the folder that contains the **project**-related classes by typing this command:

```
cd C:\grading
```

Step 4. Compile the classes with this command (**Replace ; with : for Linux or Mac OS**):

```
javac -cp blockChainGrader.jar;algs4.jar;rsa.jar;. DropboxTestBlockChain.java
```

Step 5. Finally, run the tests with this command:

```
java -cp blockChainGrader.jar;rsa.jar;. DropboxTestBlockChain
```

There are 27 tests in total and the test files are located in *files* folder.

Example output:

Process a block with no transactions
==> passed

...

Note: If you are using IDE like eclipse, just add jar files to your build path and create a folder for all text files within the working directory of your code.

To import jar file in your Eclipse IDE, follow the steps given below.

1. Right click on your project
2. Select Build Path
3. Click on Configure Build Path
4. Click on Libraries and select Add External JARs
5. Select the jar file from the required folder
6. Click and Apply and Ok

Jar Files:

1) rsa.jar: Contains classes for using RSAKeys

2) algs4.jar: Contains some useful classes like defining priority queues, stacks, etc.

3) blockChainGrader.java: Contains classes used for grading the submitted files.

Academic Honesty:

All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you collaborate. Details posted at https://www.fullerton.edu/senate/publications_policies_resolutions/ups/UPS%20300/UPS%20300.021.pdf .

Grading for this assignment:

- Extra credit: N/A
- Late submissions less than 24 hours shall be penalized 10%.
- Late submissions greater than 24 hours shall be penalized 20%.
- No assignments shall be accepted after 48 hours.

Deliverables:

- Please hand in your source code electronically
- You must make sure that the code compiles and runs correctly.
- Write a README file which contains
  - Name and Email Address of each team member
  - Anything special about your submission that we should take note of.
- One team submit **ONE copy** to Canvas
- Only submit BlockChain.java and README files

Project 3 rubric			
Criteria	**Ratings**		**Pts**
Test 1: processBlock() Process a block with no transactions	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 2: processBlock() Process a block with a single valid transaction	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 3: processBlock() Process a block with many valid transactions	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 4: processBlock() Process a block with some double spends	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 5: processBlock() Process a new genesis block	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 6: processBlock() Process a block with an invalid prevBlockHash	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 7: processBlock() Process blocks with different sorts of invalid transactions	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts

Test 8: processBlock() Process multiple blocks directly on top of the genesis block	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 9: processBlock() Process a block containing a transaction that claims a UTXO already claimed by a transaction in its parent	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 10: processBlock() Process a block containing a transaction that claims a UTXO not on its branch	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 11: processBlock() Process a block containing a transaction that claims a UTXO from earlier in its branch that has not yet been claimed	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 12: processBlock() Process a linear chain of blocks	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 13: processBlock() Process a linear chain of blocks of length CUT_OFF_AGE and then a block on top of the genesis block	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 14: processBlock() Process a linear chain of blocks of length CUT_OFF_AGE + 1 and then a block on top of the genesis block	**0.3 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.3 pts
Test 15: createBlock() Create a block when no transactions have been processed	**0.4 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.4 pts
Test 16: createBlock() Create a block after a single valid transaction has been processed	**0.4 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.4 pts

Test 16: createBlock() Create a block after a single valid transaction has been processed	**0.4 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.4 pts
Test 17: createBlock() Create a block after only invalid transactions have been processed	**0.4 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.4 pts
Test 18: createBlock() Create a block after a valid transaction has been processed, then create a second block	**0.4 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.4 pts
Test 19: createBlock() Create a block after a valid transaction has been processed that is already in a block in the longest valid branch	**0.4 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.4 pts
Test 20: createBlock() Create a block after a valid transaction has been processed that uses a UTXO already claimed by a transaction in the longest valid branch	**0.4 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.4 pts
Test 21: createBlock() Create a block after a valid transaction has been processed that is not a double spend on the longest valid branch and has not yet been included in any other block	**0.4 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.4 pts
Test 22: Combination Process a transaction, create a block, process a transaction, create a block, ...	**0.5 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.5 pts
Test 23: Combination Process a transaction, create a block, then process a block on top of that block with a transaction claiming a UTXO from that transaction	**0.5 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.5 pts
Test 24: Combination Process a transaction, create a block, then process a block on top of the genesis block with a transaction claiming a UTXO from that transaction	**0.5 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.5 pts
Test 25: Combination Process multiple blocks directly on top of the genesis block, then create a block	**0.5 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.5 pts
Test 26: Combination Construct two branches of approximately equal size, ensuring that blocks are always created on the proper branch	**0.5 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.5 pts
Test 27: Combination Similar to previous test, but then try to process blocks whose parents are at height < maxHeight - CUT_OFF_AGE	**0.5 pts** **Full Marks** passed	**0.0 pts** **No Marks** failed	0.5 pts
			Total Points: 10.0