DLT5400: DLT Implementation and Internals

Due on 19th July 2021

Assignment

Instructions

- This is an individual assignment and carries 100% of the final DLT5400 grade.
- The firm submission deadline is 19th July 2021. Hard copies are not required to be handed in.
- A soft-copy of the report and all related files must be uploaded to the VLE upload area by the same deadline. All files must be archived into a single .zip file. It is the student's responsibility to ensure that the uploaded zip file and all contents are valid.
- The report must be provided in PDF format.

Problem 1

This part of the assignment comprises 75% of the final grade and consists of the following tasks:

1. **SBB Completion**: Ensure that your SBB implementation from class is working correctly. Test it with 4 nodes (different processes). See that the node works for at least 20 blocks without a problem. For this whole assignment there is no need to test your implementation against your peers in the same study-unit.

2. **Configurable Blockchain:** You will be required to implement your code in such a manner that you can choose between the following options that you would have implemented at compile/development time: (i) Proof-of-Work or Proof-of-Turn; (ii) Account or UTXO model; and (iii) JSON or a Well-defined byte encoding. New options are discussed below.

3. **Blocktime Calculation**: For the Proof-of-Work implementation discussed in lectures, find a block hash target number of consecutive "0"s which results in an average block time of around 10 seconds. Calculate the average block times for varying number of consecutive "0"s to find the target consecutive "0"s that is closest to 10 seconds (stored as milliseconds). This value will be referred to as $< YOUR\_BLOCK\_TIME >$ throughout this assignment. This value will be used in later parts in this assignment. Document the process of finding the target number of zeros and resultant average block time.

4. **Proof-of-Turn:** Emulate a Proof-of-Stake algorithm using the following Proof-of-"Turn" algorithm which is used to decide which node gets to mine the next block:

```
Min(
  LS64B(
    Hash(
      Concat(<Node's Public Key>, <Previous Block Hash>)
    )
  )
)
```

   For each node: calculate the hash of its public key concatenated with the previous block hash, truncate the resultant hash to its lowest 64 bits. The turn then belongs to the node with the minimum/smallest resultant 64 bit number calculated above.

   If two nodes result in the same 64 bit value, then any can be chosen. The block produced after should then be the deciding block to decide which chain is valid based upon a lower generated 64 bit number.

   For the first $< PreviousBlockHash >$ use a value of "0".

   Nodes should wait $< YOUR\_BLOCK\_TIME >$ milliseconds since the last block before confirming a new block to be the next block in the chain.

5. **Account Model DB:** So far we've implemented an account model paradigm — where a single wallet address holds an amount of cryptocurrency and can be used to receive and send any amounts of cryptocurrency. This requires the program to traverse through the chain of blocks whenever we want to calculate the balance of an account. Implement Account Model functionality that stores balances in a database (any database is fine, even in-memory).

   If the balance has not yet been calculated then the chain should be traversed to calculate the balance and thereafter the balance should be saved in the database. Any transactions that take place for

Problem 1 continued on next page...

future transactions should update the database's balance (besides the transactions being added into new blocks).

6. **UTXO Model:** We've focused on a blockchain implementation that uses an account model. Here you are required to implement a UTXO model. A coin when mined will be allocated to the miner in a transaction that is marked as unspent. When the miner wants to spend this coin they will need to do this by spending the unspent transaction, this will mark the original transaction as spent and a new transaction will be generated and sent to the recipient who is then the owner of the new unspent transaction. The same process is used when this recipient spends the unspent transaction and for all other transactions.

   Implement: (i) a version that requires traversing the chain to establish whether a transaction is un/spent; and (ii) a version that makes use of a database to keep track of un/spent transactions.

7. **Design a Well-Defined Byte Encoding:** You need to design a well-defined byte encoding to be used as a protocol.

   The $< STX >< LEN >< PAYLOAD >< ETX >$ format used in class can be used.

   However, the $< PAYLOAD >$ format for messages that contain JSON objects should be changed to rather than use an inner JSON payload, to use a well defined structured byte protocol.

8. **Evaluation:** You are to evaluate each of the following configurations by recording and analysing CPU usage (and associated timings):

PoW or PoT	**Account DB or UTXO DB**	**JSON or Byte Encoding**
PoW	Account DB	JSON
PoT	Account DB	JSON
PoW	UTXO DB	JSON
PoT	UTXO DB	JSON
PoW	Account DB	Byte Encoding
PoT	Account DB	Byte Encoding
PoW	UTXO DB	Byte Encoding
PoT	UTXO DB	Byte Encoding

9. **Report:** Provide documentation for: (i) the average block time and target consecutive "0"s calculation including resultant average times and final parameters used; (ii) The protocols used. (iii) Any flaws that you see with aspects of the systems. (iv) Analysis and thoughts regarding the evaluation metrics extracted. (v) Any parts of the requirements that you did not implement. (vi) A description of what a merkle tree is, and how it could be used in the implementations to provide better efficiency.

Problem 2

This part of the assignment comprises 25% of the final grade and consists of the following tasks:

1. **Smart Contract Upload and Retrieval:** Create a Substrate Pallet that can be used to upload and store a smart contract (represent as a byte array) in the node.

   You should define a function similar to: $upload\_contract(origin, sc : Vec < u8 >)$

   This function will allow for a user to pass in a smart contract represented as the byte vector *sc*.

   The function should save the smart contract in storage, and return to the user a 32-bit integer which is the unique smart contract address.

   A second function similar to: $retrieve\_contract(origin, sc : u32)$ should be implemented which requires as input a unique smart contract's 32-bit address (as returned from $upload\_contract$).

   Note: you do not need to abide strictly by the function signatures/definitions above.

2. **Limited JVM-like Interpreter:** You should implement a function similar to $execute\_contract(origin, sc : u32)$

   The function will execute the smart contract code associated with the *sc* reference passed in (as uploaded in the first problem).

   You are to implement JVM bytecode and design added bytecode operations to support execution of the following code:

   ```
   if (blockNumber % 2 == 0) {
           emit "You win";
   } else {
           emit "You lose";
   }
   ```

   blockNumber should return back an unsigned 32-bit integer representing the current block being mined/forged. You will need to design a new bytecode operator to support this.

   $emit < string >$ should emit a message to the user. You will need to design new bytecode for this.

   You should be able to use other 32-bit int JVM instructions for the rest of the code.

   This is the whole code of the smart contract. It's a single function that gets executed whenever the smart contract is called.

   Though a UI is not explicitly required, you do need to demonstrate functionality of the Pallet functions working. You may want to consider using the front-end template as demomnstrated in class.

3. **Report:** Write a brief report including: (i) a description of the bytecode operations you implemented; and (ii) listing any functionality that you did not implement.