1. **Introduction**

In today’s technology genomics research is widely used for discovery of mutations and by that way, cures for human diseases can be further improved. Data sharing is crucial for this kind genomics research; however the data must be safe from being removed from existing databases by financial and political reasons, required by the organizations []. Our project, CrypDist, provides a way to access genomics data more securely. It is a decentralized distributed system which uses a distributed ledger called blockchain to record the URL links of datasets.

Blockchain is a cryptographic data structure which ensures immutability of data and avoids third-party access. It basically provides synchronization of the data links among many users and it also includes data summaries. The mentioned data is not kept in the blockchain because of its size.

1. **Packages and Tools**

* There is a local database in each of the machines which is developed by using PostgreSQL package [].
* For managing logging information, Log4j package is used which belongs to Apache Software Foundation [].
* For testing purposes, a stub server is used for uploading the data which is provided by Amazon Web Services []. In the future, Akamai services [] is planned to be configured for the project for more efficient usage.
* //TODO Timestamp server
* For build automation, we implemented the project by using Apache Maven [].

1. **User Interface**
2. **Hardware/Software Mapping**
3. **Software Architecture**
   1. **Subsystem Decomposition**

Figure x shows the subsystem decomposition of the system. Class diagrams are drawn after that for clarity.

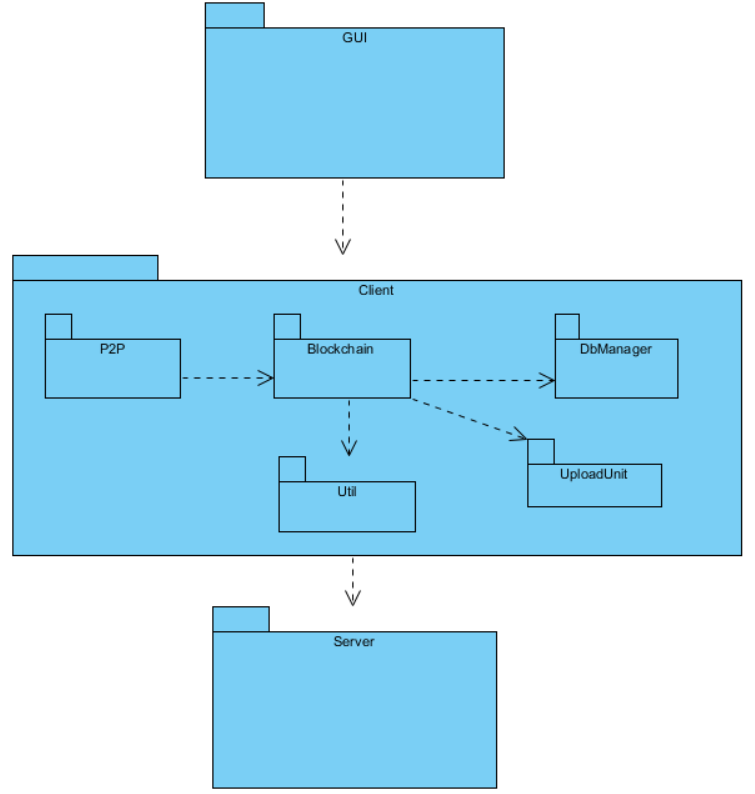
****

Figure x – Subsystem Decomposition

The system has a 3-tier architectural style. The top layer includes the graphical user interface of the system. The middle layer includes the Client interfaces where client is a peer in the system. Blockchain of peer is managed by the Blockchain subsystem and it is kept in a local database which is managed by the DbManager subsystem. By the P2P subsystem, peer can contribute to peer-to-peer connection by other peers. UploadUnit subsystem is for uploading genome data to the off-the-shelf server and Util subsystem is used for managing client operations such as parallel download and receiving hash. Finally, the Server subsystem acts as a registrar which keeps the IP addresses and authentication credentials of the clients and peers can get those information from the registrar as necessary.

* 1. **Subsystem Interfaces**

**GUI Subsystem**

Figure x shows the GUI subsystem.

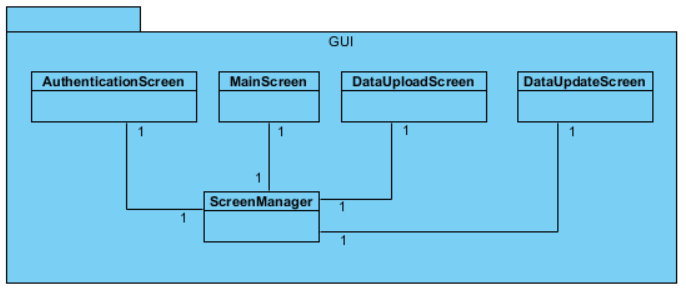


Figure x – GUI Subsystem

Clients authenticate themselves via AuthenticationScreen. In MainScreen, they can query for the data in the blockchain. They can upload new data by DataUploadScreen and provide new versions for it by DataUpdateScreen. ScreenManager provides the interface between Client and GUI subsystems that it forwards the requests coming from the user interface to the bottom layer.

**Client Subsystem**

* **P2P Subsystem**

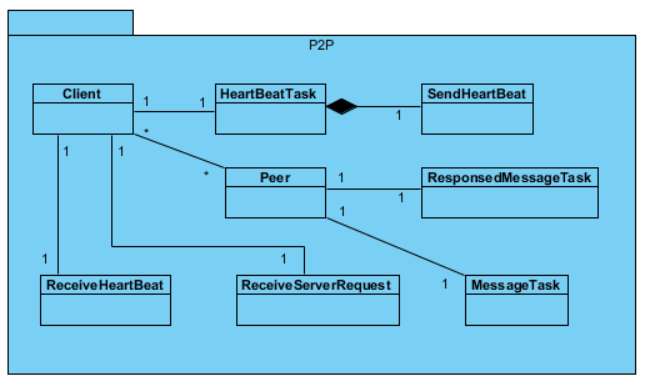


Figure x – P2P Subsystem

* **Blockchain Subsystem**

Figure x shows the Blockchain subsystem.

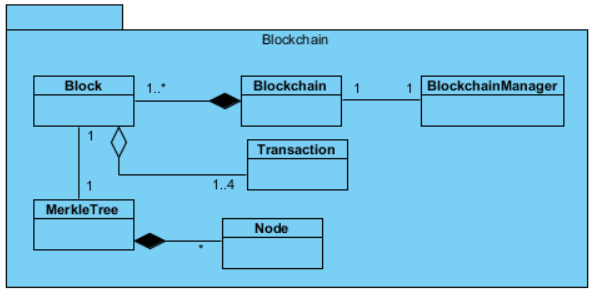
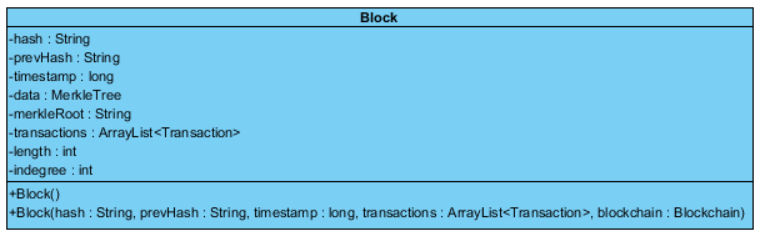


Figure x – Blockchain Subsystem

**Block Class**



**Interfaces:** Serializable

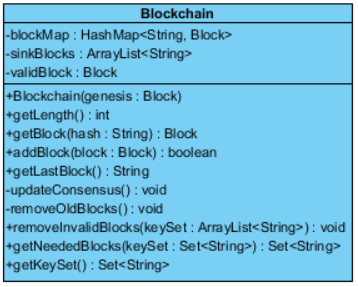
**Attributes**

* + **private String hash:** Unique hash key of the block
  + **private String prevHash:** Hash key of the previous block
  + **private long timestamp:** Creation time of the block
  + **private MerkleTree data:** The data structure which holds the transaction signatures
  + **private String merkleRoot:** Root signature of the merkle tree
  + **private ArrayList<Transaction> transactions:** Transactions in the block
  + **private int length:** Number of previous blocks in the chain
  + **private int indegree:** Number of blocks which point to the block

**Constructors**

* + **public Block():** Constructs the genesis block
  + **public Block(String hash, String prevHash, long timestamp, ArrayList<Transaction> transactions, Blockchain blockchain)**

**Blockchain Class**



**Interfaces:** Serializable

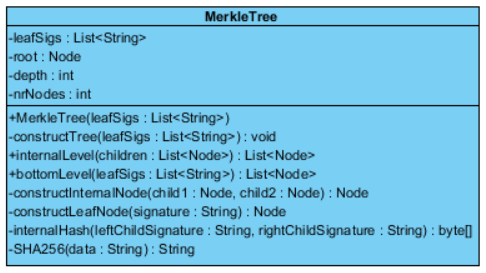
**Attributes**

* + **private HashMap<String, Block> blockMap:** Maps hashes to blocks
  + **private ArrayList<String> sinkBlocks:** The end blocks of each branch
  + **private Block validBlock:** The end block of the valid (longest) branch

**Operations**

* + **public Blockchain(Block genesis):** Genesis block is hard-coded.
  + **public int getLength():** Returns length of the valid branch.
  + **public Block getBlock(String hash):** Returns the block with the given hash.
  + **public boolean addBlock(Block block):** Adds the block to the blockchain and returns an ACK or negative ACK.
  + **public String getLastBlock():** Returns the hash of the last block.
  + **private void updateConsensus():** Updates the current branch according to the longest chain rule.
  + **private void removeOldBlocks():** Removes the blocks which are in the old branches.
  + **public void removeInvalidBlocks(ArrayList<String> keySet):** Removes the blocks which majority of the peers do not have.
  + **public Set<String> getNeededBlocks(Set<String> keySet):** Returns the blocks which majority of the peers have, but the current peer does not.
  + **public Set<String> getKeySet():** Returns the block hashes.

**MerkleTree Class**



**Attributes:**

* + **private List<String> leafSigs:** Signatures of the leaves
  + **private Node root:** Root node of the tree
  + **private int depth:** Depth of the tree
  + **private int nrNodes:** Number of nodes in the tree

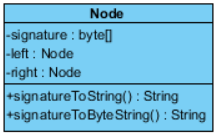
**Constructor:**

* **public MerkleTree(List<String> leafSigs)**

**Operations:**

* **private void constructTree(List<String> leafSigs)**
* **public List<Node> internalLevel(List<Node> children):** Returns an internal level whose children are the specified ones
* **public List<Node> bottomLevel(List<String> leafSigs):** Returns the bottom level of the tree.
* **private void constructInternalNode(Node child1, Node child2)**
* **private void constructLeafNode(String signature)**
* **public byte[] internalHash(String leftChildSignature, String rightChildSignature):** Computes the signature of the internal node from the child nodes.
* **public String SHA256(String data):** Computes the signature of the transaction by the SHA256 algorithm.

**Node Class**



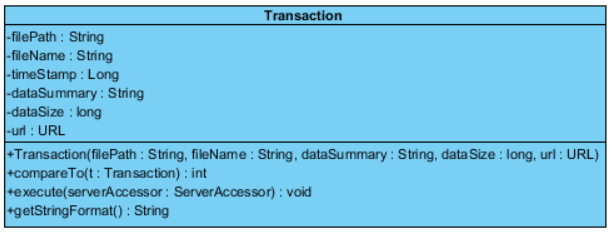
**Attributes**

* + **private byte[] signature: Signature of the transaction kept in the node**
  + **private Node left**
  + **private Node right**

**Operations**

* + **public String signatureToString()**
  + **public String signatureToByteString()**

**Transaction Class**



**Interface:** Comparable

**Attributes**

* + **private String filePath**
  + **private String fileName**
  + **private Long timeStamp**
  + **private String dataSummary:** Summary of the genomics data in the server
  + **private long dataSize:** Size of the genomics data in the server
  + **private URL url:** Link of the genomics data in the server

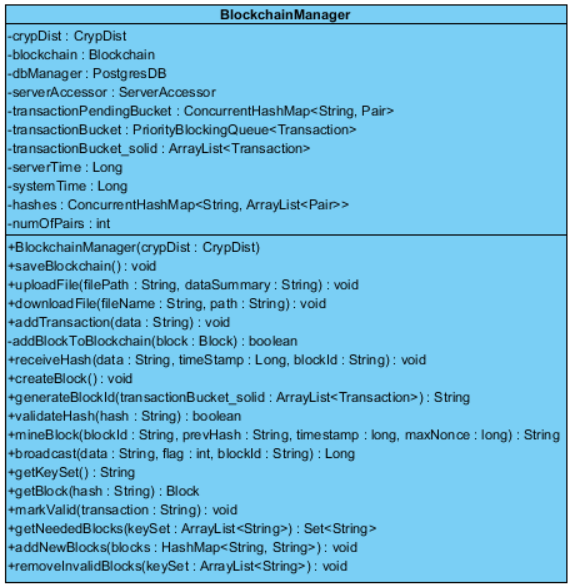
**Constructor**

* + **public Transaction(String filePath, String fileName, String dataSummary, long dataSize, URL url)**

**Operations**

* + **public int compareTo(Transaction t):** Compares to t according to timestamps.
  + **public void execute(ServerAccessor serverAccessor):** Executes the transaction.
  + **public String getStringFormat():** Returns the string format of transaction for merkle tree

**BlockchainManager Class**



**Attributes**

* + **private CrypDist crypDist**
  + **private Blockchain blockchain**
  + **private PostgresDb dbManager**
  + **private ServerAccessor serverAccessor**
  + **private ConcurrentHashMap<String, Pair> transactionPendingBucket:** Bucket which holds pending transactions (the transactions which are not validated yet)
  + **private PriorityBlockingQueue<Transaction> transactionBucket:** Priority queue which holds the transactions, which are validated but not added into a block yet, according to the order of their timestamps
  + **private ArrayList<Transaction> transactionBucket\_solid:** Transactions which will be added into a block soon (first transactions in the priority queue)
  + **private Long serverTime:** Time received from UTC server
  + **private Long systemTime:** Time received from local computer
  + **private ConcurrentHashMap<String, ArrayList<Pair>> hashes:** Hashes received for a block
  + **private int numOfPairs:** Number of active peers in the system who can validate a transaction or generate a hash

**Constructor**

* + **public BlockchainManager(CrypDist crypDist)**

**Operations**

* + **public void saveBlockchain():** Saves blockchain into the local database.
  + **public void uploadFile(String filePath, String dataSummary):** Uploads a file with given path and data summary to the server
  + **public void downloadFile(String fileName, String path):** Downloads a file from the server
  + **public void addTransaction(String data):** Adds a transaction to the system
  + **public boolean addBlockToBlockchain(Block block):** Add the block to the blockchain
  + **public void receiveHash(String data, Long timestamp, String blockId):** Receives a new hash for the block with the given ID
  + **public void createBlock():** Creates a new block with the first transactions in the priority queue
  + **public String generateBlockId(ArrayList<Transaction> transactionBucket\_solid):** Generates an ID for the block with the given transactions
  + **public boolean validateHash(String hash):** Checks if the given hash is the same as the hash of the last block
  + **public String mineBlock(String blockId, String prevHash, long timestamp, long maxNonce):** Finds a valid hash for the block with the given data
  + **public Long broadcast(String data, int flag, String blockId):** Broadcasts a message according to the flag (either transaction, hash or parallel download). Returns the time of broadcasting.
  + **public String getKeySet():** Returns the string representation of hash set in the blockchain
  + **public Block getBlock(String hash):** Gets the block with the given hash from the blockchain
  + **public void markValid(String transaction):** Increments the ACK count for the given transaction and if it reaches to majority, add it to the pending transaction bucket.
  + **public Set<String> getNeededBlocks(ArrayList<String> keySet):** Returns the hash set of the blocks which does not exist in the local blockchain, but majority of the blockchains.
  + **public void addNewBlocks(HashMap<String, String> blocks):** Adds the new blocks, which are received from other peers, to the blockchain.
  + **public void removeInvalidBlocks(ArrayList<String> keySet):** Removes the blocks which exist in the local blockchain, but not the majority.

**BlockMiner Class**

**BlockchainBatch Class**

**HashValidation Class**

* **DbManager Subsystem**

Figure x shows the DbManager subsystem.

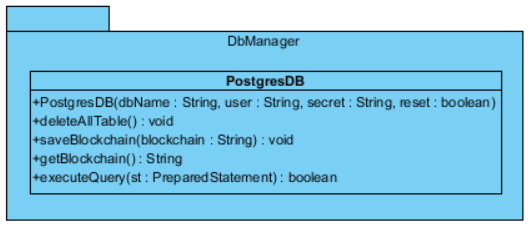


Figure x – DbManager Subsystem

* **UploadUnit Subsystem**

Figure x shows the UploadUnit subsystem.

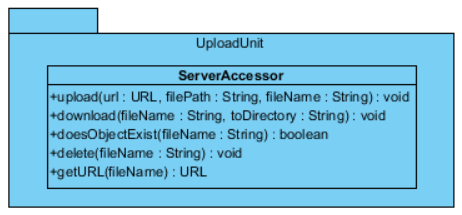


Figure x – UploadUnit Subsystem

* **Util Subsystem**

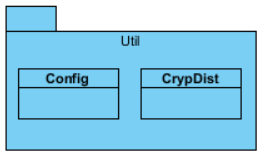


Figure x – Util Subsystem

**Server Subsystem**

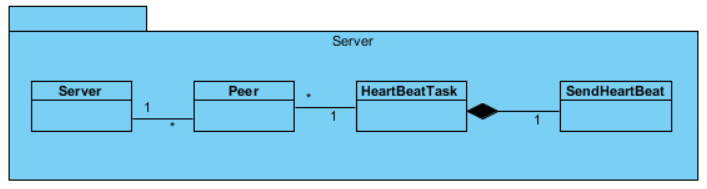


Figure x – Server Subsystem

* 1. **Design Patterns**

1. **Data Structures and Algorithms**
   1. **Blockchain Structure**
   2. **Block Mining Algorithm**
   3. **Digital Signature Algorithm**
   4. **Hash Choosing Algorithm**
   5. **Parallel Download Algorithm**
2. **Impact of Engineering Solutions**
3. **Contemporary Issues**
4. **User’s Manual**