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 []. We use S3 as a storage unit and its own versioning system. The data is kept in encrypted format and available for anyone to download. Write permissions are configured to be granted only to the authenticated users of CrypDist. In the future, Akamai services [] is planned to be configured for the project for more efficient usage.
* For blockchain and its synchronisation over the clients in the network, it is crucial to share a global clock. Clocks in clients are configured according to the NIST Internet Time Servers. We use the server in Macon, Georgia ([nist1-macon.macon.ga.us](http://nist1-macon.macon.ga.us)).
* For package management, we implemented the project by using Apache Maven [].

1. **User Interface**
2. **Data Structures and Algorithms**
   1. **Blockchain Structure**

Blockchain is a distributed ledger in which transactions are recorded and cannot be changed later. It is replicated among peers and for consistency, all peers must have the same list of transactions. Each block has up to 4 transactions and each transaction contains a data summary, a URL link to the actual data, and digital signature of the peer who uploads the data. All blocks are identified by a unique hash key, and all of them point to the previous one by containing its hash. Hash keys are chosen according to block mining algorithm such that they depend on the previous hash values, so changing one block would affect the later ones. This ensures immutability of the structure. To validate a blockchain, just checking the last hash value with the majority of peers is enough in this context.

The data structure also has the ability to fork. That means if two peers generate different blocks at the same time which have the same previous hash values, the structure produces two branches. For achieving consensus among peers, the longest branch is chosen as the valid one.

* 1. **Block Mining Algorithm**

The process of finding a valid hash key for a block is called block mining. Each peer executes this algorithm at the same time, and the first one to produce a valid hash becomes the winner. By that way, each peer can have a contribution to the system, so scalability is improved. A valid hash key has the constraint that first byte of the key must be zero. This is for ensuring that the hash keys produced from the different data cannot be the same, or the probability of them being equal is almost zero.

* 1. **Digital Signature Algorithm**

For security purposes, peers should sign the transactions they upload such that if they contain malicious data, peers can be backtracked. There is a key pair in the system which consists of a public key and a private key. When a peer authenticates himself, the server gives him the private key. When he sends a transaction, he encrypts his username by the private key and other peers can validate it by using the public key. So when the transaction is put into the blockchain, username of the peer can be included in the block as a digital signature.

* 1. **Parallel Download Algorithm**

When last hash of the blockchain is not the same with the majority of the peers, the peer must download the blocks from others to upload a new transaction. At first, he receives the full key set from all of them and takes the ones in the majority. Then he decides which blocks he needs to get from others and according to that he sends the particular requests.

* 1. **Safety and Progress Properties**

For concluding that system works as desired, safety and progress properties are ensured. Safety property of the system is that at least one of two peers which have different versions of the blockchain at the same time cannot upload data. This ensures the consistency of transactions in the system. This property is provided by the blockchain validation routine such that if the last hash of the blockchain is not the same with the majority, then the peer can upload data to system until he updates his blockchain.

Progress property is that if a peer wants to execute a valid transaction, the transaction will be executed and recorded into blockchain eventually. This is ensured by putting the transactions into a pending queue before they are validated. When they are validated, they are executed and put into a priority queue according to their timestamps and when there are 4 transactions in the queue, the block is generated. If number of transactions does not become 4 for a long time, the block is generated with less number of transactions.

1. **Software Architecture**
   1. **Subsystem Decomposition**

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

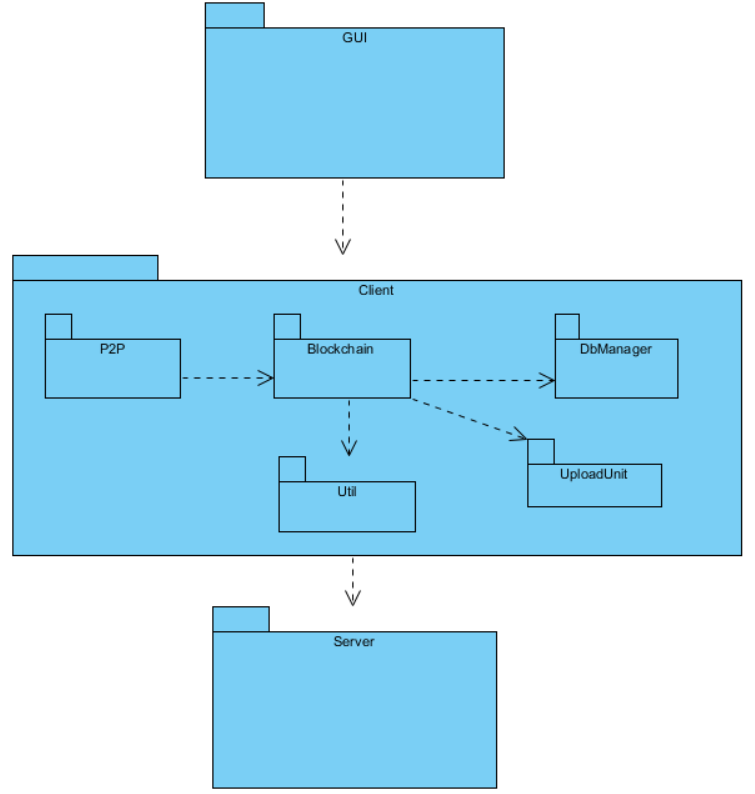
****

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.

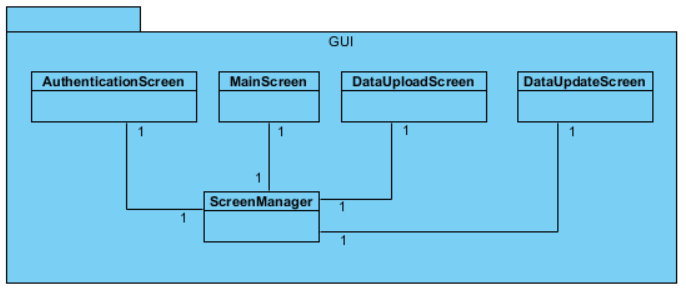


Figure x – GUI Subsystem

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 shows the P2P subsystem.

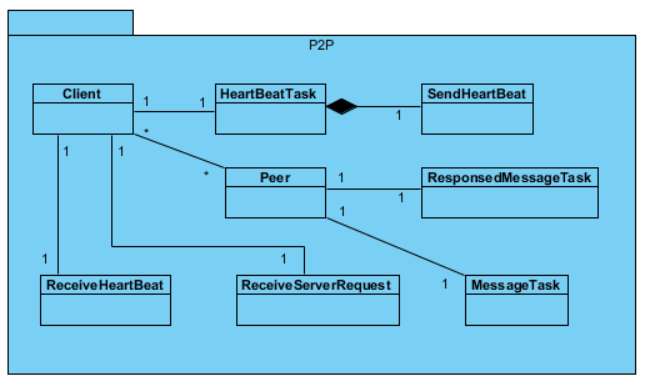
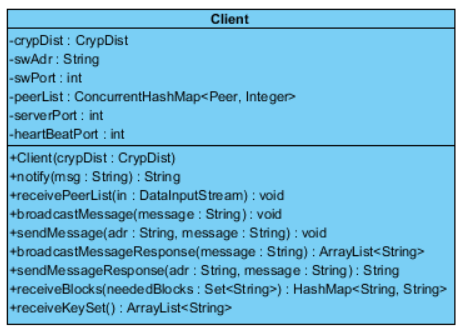


Figure x – P2P Subsystem

Via HeartBeatTask and ReceiveHeartBeat classes, Client sends heart beats periodically and receives from others. Peer class includes the information of a peer and Client keeps all peers’ information. By ReceiveServerRequest class, client receives requests from other peers. By MessageTask class, Client sends message to other peers, and receives their responses by the ResponsedMessageTask class.

**Client Class**



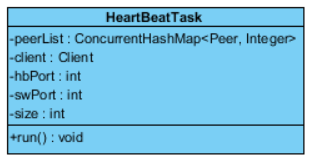
**Attributes**

* + **private CrypDist crypDist**
  + **private String swAdr:** Address of the registrar
  + **private int swPort:** Port number of the registrar
  + **private ConcurrentHashMap<Peer, Integer> peerList:** List of known peers and their not-responded heart beat counts
  + **private int serverPort:** Server port of the peer
  + **private int heartBeatPort:** Heart beat port of the peer

**Operations**

* + **public String notify(String msg):** Notifies CrypDist instance about its state with a message.
  + **public void receivePeerList(DataInputStream in):** Receives the peer list from the registrar.
  + **public void broadcastMessage(String message):** Broadcasts a message to the peers.
  + **public void sendMessage(String adr, String message):** Sends a message to a particular peer.
  + **public ArrayList<String> broadcastMessageResponse(String message):** Collects the responses for a broadcasted message.
  + **public String sendMessageResponse(String adr, String message):** Gets the response for a sent message.
  + **public HashMap<String, String> receiveBlocks(Set<String> neededBlocks):** Receives the needed blocks from other peers.
  + **public ArrayList<String> receiveKeySet():** Receives the key set from other peers.

**HeartBeatTask Class**



**Parent Class:** TimerTask

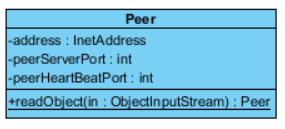
**Attributes**

* + **private ConcurrentHashMap<Peer, Integer>:** List of known peers and their not-responded heart beat counts
  + **private Client client**
  + **private int hbPort**
  + **private int swPort**
  + **private int size:** Size of the peer list

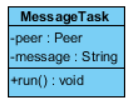
**Operations**

* + **public void run():** Sends heart beats to peers and collects their responses.

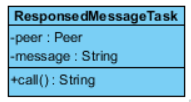
**Peer Class**



**MessageTask Class**



**ResponsedMessageTask Class**



* **Blockchain Subsystem**

Figure x shows the Blockchain subsystem.

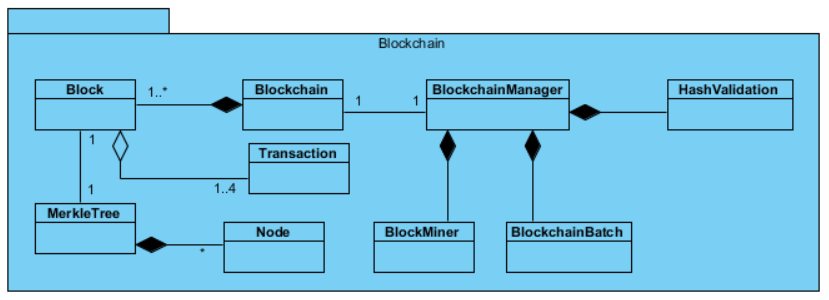
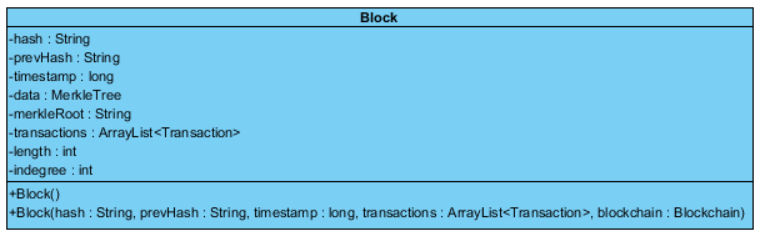


Figure x – Blockchain Subsystem

BlockchainManager class provides the interface for the subsystem. It handles the operations on the blocks and transactions which are hash validation, mining a block, achieving consensus and parallel download. Also it runs BlockchainBatch which adds the transactions to the transaction bucket from priority queue when timeout occurs.

**Block Class**



**Interfaces:** Serializable

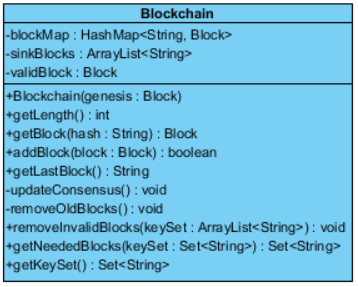
**Attributes**

* + **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

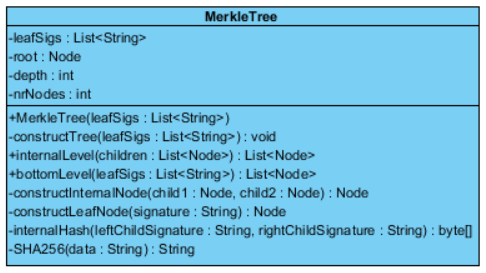
**Attributes**

* + **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

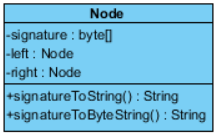
**Constructor:**

* **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**



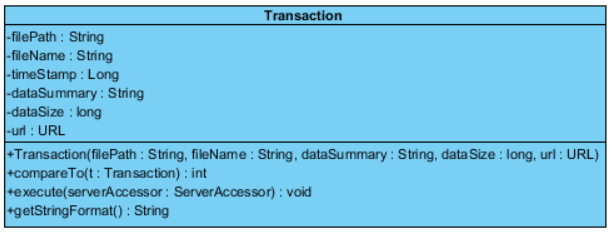
**Attributes**

* + **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

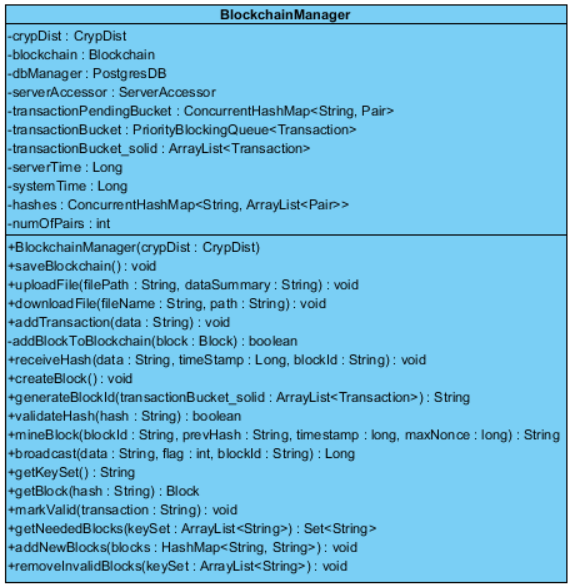
**Constructor**

* + **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

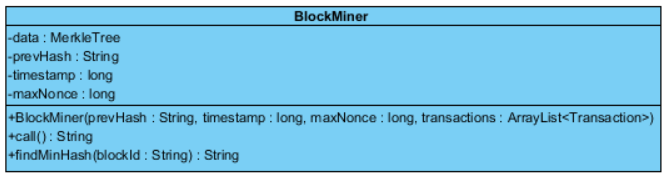
**Constructor**

* + **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 to store the hashes generated for it
  + **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**



**Parent class:** Callable<String>

**Attributes**

* + **private MerkleTree data**
  + **private String prevHash**
  + **private long timestamp**
  + **private long maxNonce:** Maximum possible value for nonce

**Constructor**

* + **public BlockMiner(String prevHash, long timestamp, long maxNonce, ArrayList<Transaction> transactions)**

**Operations**

* + **public String call():** Finds a valid hash by trying possible values for nonce
  + **public String findMinHash(String blockId):** Returns the minimum hash among the ones received for the block
* **DbManager Subsystem**

Figure x shows the DbManager subsystem.

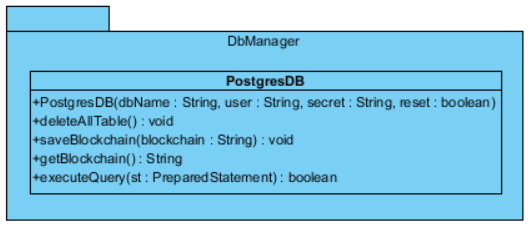


Figure x – DbManager Subsystem

PostgresDB class saves and gets the blockchain and pending transactions to the local database.

* **UploadUnit Subsystem**

Figure x shows the UploadUnit subsystem.

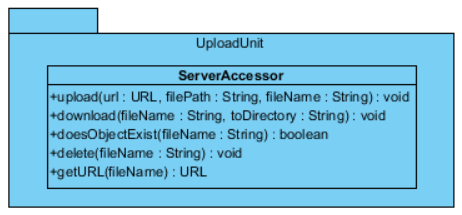


Figure x – UploadUnit Subsystem

ServerAccessor class adapts the Amazon server to the CrypDist system. It manages data upload and download and returns the links to system accordingly.

* **Util Subsystem**

Figure x shows the UploadUnit subsystem.

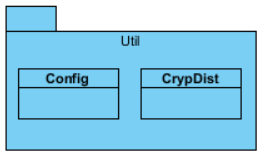
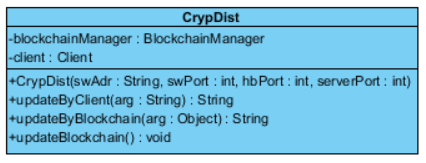


Figure x – Util Subsystem

Config class includes the constant fields for the system such as ACK numbers, message flags, server address and port number, and so on.

**CrypDist Class**



**Operations**

* + **public String updateByClient(String arg):** Performs the appropriate operation according to input of the Client where the input consists of the IP address of the Client and a flag indicates the type of operation.
  + **public String updateByBlockchain(Object arg):** Performs the appropriate operation according to input of the BlockchainManager.
  + **public void updateBlockchain():** Updates the blockchain by downloading the needed blocks from other peers.

**Server Subsystem**

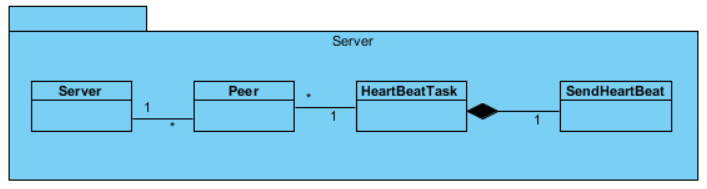
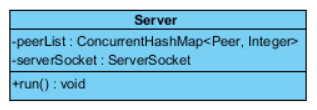


Figure x – Server Subsystem

**Server Class**



**Parent Class:** Thread

**Attributes**

* + **private ConcurrentHashMap<Peer, Integer> peerList:** List of peers and their not-responded heart beat counts
  + **private ServerSocket serverSocket**

**Operations**

* + **public void run():** Accepts new peer, receives their ports and sends them peer list. Also sends heart beats to peers periodically.

The other classes are the same with the ones in the Client subsystem.

* 1. **Pattern Applications**

**Façade Design Pattern**

For encapsulating individual subsystems, Façade pattern is applicable. By providing the subsystem interface by only one class, other classes can be abstracted. For Client subsystem, CrypDist class is used for this purpose such that ScreenManager class can call the appropriate methods according to the GUI events. For Blockchain subsystem, BlockchainManager class encapsulates the system. For Server subsystem, clients can reach via the Server class.

**Adapter Design Pattern**

Figure x shows the Adapter design pattern. Since the server is used as an off-the-shelf component, a generic ServerAccessor class is used for the method declarations. By including AmazonAccessor class, Amazon-specific implementations can be used. In the future, Akamai servers can be integrated easily by adding a class for AkamaiAccessor.

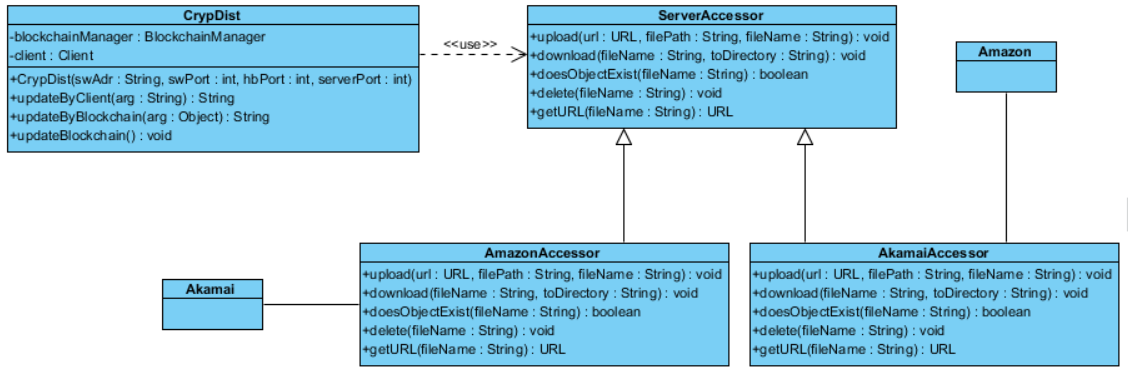


Figure x – Adapter Design Pattern

**Singleton Design Pattern**

Blockchain object is replicated among the peers. So, each local program contains exactly one Blockchain object. It is encapsulated in the BlockManager class, so that class should have also a singleton instance. So, singleton pattern is used for that purpose.

1. **Impact of Engineering Solutions**

The solutions we provide in the project is mainly based on the field of distributed systems. According to our design goals, solutions should aim to make communications without using a centralized control. By that way, single point of failure and congestion in the network can be eliminated up to some point.

However there are also some issues concerning with this approach. One of them is user incentives such that since each peer should also act as a server, they should willingly give their resources. In systems such as Bitcoin, coins are used for that purpose. In CrypDist, when new data is uploaded, all peers can access to that and get benefits accordingly, so this is the user incentive of the system.

By that system, genomics data become more easily accessible, so in that way mutations and cures for diseases can be discovered. In the environmental and societal context, it supports health for humans and society.

Economically speaking, the uploaded data is free, so this should encourage more people for the genomics research.

1. **Contemporary Issues**

Global Alliance for Genomics and Health is a recently established organization which aims to enable sharing of genomics data effectively. When there is centralized control on the data, because of political reasons, the servers may become passive just like the Wellcome Trust server. For that purpose, current projects aim to distribute data to multiple servers and databases to avoid single point of failure. Cancer Gene Trust Network is an example for this such that it aims to share data in real time. In our project, Akamai services can be used for that purpose, in the future.

The main purpose of the project is to enable researchers to access the data securely by combining blockchain technology with genomics data distribution. In that way, the links for the data cannot be corrupted by third party access.

1. **User’s Manual**

**References**

<http://science.sciencemag.org/content/352/6291/1278.full>

<https://wellcomecollection.org/what-we-do/wellcome-trust>