CS4398 SM1 Project Description - CrypoSim

Eric Brun

Cynthia Cordova

Taylor Cano

Mason Cruze

Description by Eric Brun

Table of Contents:

1. Overview:

This program is a basic simulation modelling the behavior of a blockchain system for creating a distributed record system. The blockchain is used to record transactions of an imaginary Cryptocurrency called **crypto**. As records are added to the blockchain and blockchains are updated, new crypto is generated.

2. The Model – Viewer – Controller (MVC) System:

The overall program is designed around the principle of Model-Viewer-Controller (MVC). Under MVC, the program is divided into three main components, the Model component which manages the essential internal data of the program, the Viewer which handles methods and data needed to create a graphical display of data from the Model, and the Controller which coordinates interaction between the Model and Viewer components.

Due to the graphical library we chose to use (Simple Fast Multimedia Library, aka SFML), the Viewer is also responsible for taking “raw” user inputs from the mouse and keyboard and creating appropriate data to be sent to the Controller. The Controller decides what to do with the passed input data from the Viewer.

3. The Model

The Model is responsible for holding and operating on all the data associated with the simulation. This includes data on how much cryptocurrency exists, the number of transactions that occurred, the rate of time by which the simulation updates, and all classes and data related to different Nodes in the Network and the copy of the Blockchain that each Node has.

3.1 Transactions

A **Transaction** is the data being recorded in the Blockchain. A Transaction is used to record a monetary transaction between two users (a Sender and a Receiver) of the cryptocurrency. The Sender is giving crypto from their account balance to the Receiver. For the sake of simplicity in the simulation, individual users are represented as Nodes.

A Transaction instance has the following data:

- Timestamp: records the moment in time the Transaction instance was created
- CryptoAmount: records the amount of crypto traded in the Transaction
- SenderID: records a string representing the Sender
- ReceiverID: records a string representing the Receiver

3.2 Blocks and the Blockchain

Blocks are the basic component of the Blockchain class. Each instance of the Block class contains an index value, a “nonce” value, a hash string generated from an encryption algorithm which uses the nonce value to increase decryption difficulty, a hash from the “previous” neighboring Block in the chain, a timestamp recording when the Block instance was created, and a Transaction instance containing the data pertaining to the particular transaction that this Block is recording.

The creation of a new Block requires an encryption process where the hash of the block is generated by feeding all other data in the Block (including the hash of the most recent Block at the end of the Blockchain) into an encryption algorithm. This simulation uses the SHA256 encryption algorithm. Once the Block is created, no data in the Block can be modified.

The **Blockchain** is a container of individual and distinct Blocks. A Block is distinct if at least one of the 4 attributes in the Block's Transaction is different from any other Block. When adding a new Block to the chain, the Blockchain checks if the Block is a duplicate or not. If the Block is distinct, then the Blockchain pushes the new Block to the back of a vector container of Blocks.

3.3 Nodes and the Network

In a real world blockchain implementation, the blockchain is distributed over multiple user machines which are connected by a network. Each user in the network has their own copy of a blockchain and can be notified of updates to the chain by other members of the network. This creates a "decentralized" record system where copies of the record are distributed across many machines, in contrast to a centralized record system where all data is located on a single machine.

For simplicity, we've created a basic model of an online network of users. Individual users are Nodes and the container responsible for managing the Nodes is the Network.

The **Node** class contains the following attributes:

- Blockchain: a Blockchain instance
- ID: a string used to identify this Node instance
- Balance: a value tracking how much crypto the Node has available
- A list of all other Nodes in the Network (Neighbors)
- A reference to the DataManager of the simulation

Each Node can add a new block to its Blockchain. When a Node adds a Block to its Blockchain, it also prompts all other Nodes connected to it to update their Blockchains. This results in any new Block in the chain being distributed out to all other Nodes in the Network.

The algorithm works as follows:

1. This Node adds a Block to its Blockchain. The new Block is at the end of the chain.
2. For every neighbor Node connected to this Node, this Node sends to the Neighbor Nodes a reference to the last Block in this Node's chain.
3. Each Neighbor node compares Blocks in its chain to the last Block in this Node's chain. If no duplicate Blocks exist, the Neighbor Node adds this Block to its chain and then notifies its Neighbors of the change.

The **Network** class is a container of Nodes. The Network ensures each Node is connected to every other Node. The Network provides a layer of interface between client code and the collection of Nodes to make operations such as adding a new Node to the Network easier.

The process for adding a new Node to the Network is the following:

1. Create a new Node
2. Assign an appropriate ID to the Node
3. Connect the Node to the DataManager
4. For each Node already in the Network, connect each Node to this Node
5. Push this Node to the back of the vector of Nodes

3.4 The Driver

The **Driver** class is the component of the simulation responsible for operating on the simulation after enough time passes. This is achieved by the Driver's methods pollForAction() and takeAction().

Driver::pollForAction() checks if enough time has passed since the last time the Driver performed an action on the simulation. If enough time has passed, then the Driver executes takeAction().

Driver::takeAction() randomly picks from a number of possible actions to take, and then executes that action. Currently there are two actions the Driver may take:

1. Do nothing: The Driver does nothing for this action cycle. This is to add a little unreliability to the rate at which things occur in the simulation, to model how users in a real cryptocurrency system may choose to make a transaction or not make a transaction at any time.
2. Create a new Transaction: This is accomplished with Driver::addTransaction().
3. Add a new Node to the Network using Driver::addNewNode().

The process for addTransaction() is as follows:

1. The Driver randomly picks two different Nodes in the Network. One of these two Nodes is chosen to be a Sender in a Transaction and the other is the Receiver.
2. The Driver gets a random amount of crypto from the Sender Node's balance.
3. The Driver gets the current system time.
4. The Driver creates a Transaction from the above data and sends the Transaction to the Receiver Node to be added to the Blockchain.
5. The Driver notifies the DataManager that a new Transaction was created, to increment the counter of total Transactions.

Adding transactions are the primary means by which the Driver modifies the simulation.

The process for adding new Nodes to the Network is simpler:

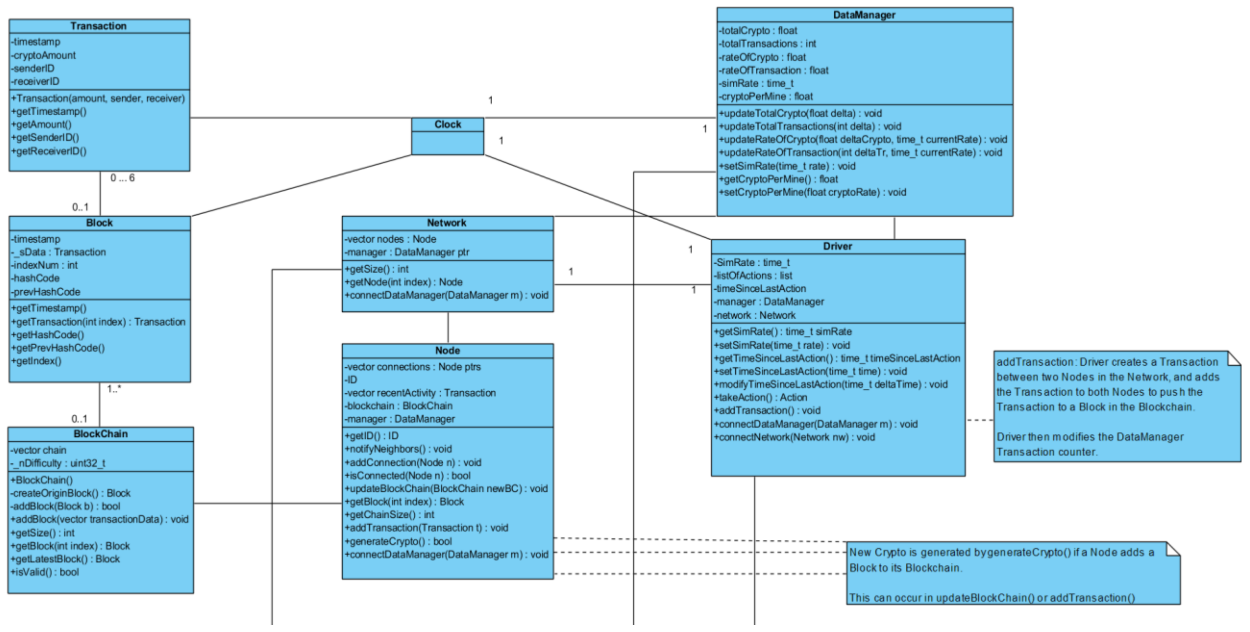
1. The Driver calls the Network's addNewNode() method.
2. The Driver notifies the DataManager of the change in the node counter.

3.5 The DataManager

The DataManager class is used to manage various counters tracking what's occurred in the simulation. The DataManager has the following attributes:

- totalCrypto: tracks how much crypto has been generated so far in the sim
- totalTransactions: tracks how many Transactions have occurred in the sim
- cryptoPerMine: how much new Crypto is generated every time a Block is added to a Node's Blockchain
- baseSimRate: tracks the base rate of time used by the sim for update cycles
- timePassed: tracks how much time has passed in total in the sim

3.6 All Together


Transaction
-timestamp
-cryptoAmount
-senderID
-receiverID
+Transaction(amount, sender, receiver)
+getTimestamp()
+getAmount()
+getSenderID()
+getReceiverID()
0 ... 6
0..1
Clock
1
1
DataManager
-totalCrypto : float
-totalTransactions : int
-rateOfCrypto : float
-rateOfTransaction : float
-simRate : time_t
-cryptoPerMine : float
+updateTotalCrypto(float delta) : void
+updateTotalTransactions(int delta) : void
+updateRateOfCrypto(float deltaCrypto, time_t currentRate) : void
+updateRateOfTransaction(int deltaTr, time_t currentRate) : void
+setSimRate(time_t rate) : void
+getCryptoPerMine() : float
+setCryptoPerMine(float cryptoRate) : void
1
Block
-timestamp
-_sData : Transaction
-indexNum : int
-hashCode
-prevHashCode
+getTimestamp()
+getTransaction(int index) : Transaction
+getHashCode()
+getPrevHashCode()
+getIndex()
1..*
0..1
Network
-vector nodes : Node
-manager : DataManager ptr
+getSize() : int
+getNode(int index) : Node
+connectDataManager(DataManager m) : void
1
Driver
1
1
-SimRate : time_t
-listOfActions : list
-timeSinceLastAction
-manager : DataManager
-network : Network
+getSimRate() : time_t simRate
+setSimRate(time_t rate) : void
+getTimeSinceLastAction() : time_t timeSinceLastAction
+setTimeSinceLastAction(time_t time) : void
+modifyTimeSinceLastAction(time_t deltaTime) : void
+takeAction() : Action
+addTransaction() : void
+connectDataManager(DataManager m) : void
+connectNetwork(Network nw) : void
addTransaction: Driver creates a Transaction between two Nodes in the Network, and adds the Transaction to both Nodes to push the Transaction to a Block in the Blockchain.
Driver then modifies the DataManager Transaction counter.
Node
-vector connections : Node ptrs
-ID
-vector recentActivity : Transaction
-blockchain : BlockChain
-manager : DataManager
+getID() : ID
+notifyNeighbors() : void
+addConnection(Node n) : void
+isConnected(Node n) : bool
+updateBlockChain(BlockChain newBC) : void
+getBlock(int index) : Block
+getChainSize() : int
+addTransaction(Transaction t) : void
+generateCrypto() : bool
+connectDataManager(DataManager m) : void
BlockChain
-vector chain
-_nDifficulty : uint32_t
+BlockChain()
-createOriginBlock() : Block
-addBlock(Block b) : bool
+addBlock(vector transactionData) : void
+getSize() : int
+getBlock(int index) : Block
+getLatestBlock() : Block
+isValid() : bool
New Crypto is generated by generateCrypto() if a Node adds a Block to its Blockchain.
This can occur in updateBlockChain() or addTransaction()

All of these components together make up the simulation within the Model. Every cycle of the main program loop the Model sends an update to the Driver about how much time has passed since the last loop, and the Driver checks if it's time to take an action. If the Driver takes an action it either adds a Node to the Network or adds a Transaction to the Blockchain of one of the Nodes, which triggers all Nodes in the Network to update their Blockchains. By modifying the Blockchains each Node generates a small amount of Crypto which increases their crypto balance and increases the total supply of Crypto.

3.7 Model Event Processing

Event Type	Description of Action
EventClose	Model creates a NoticeClose to send to Viewer
EventPause	Model sets a pause flag and stops running the sim until another EventPause is processed. Model creates a NoticePause to send to Viewer.
EventSpeedChange	Model modifies the update rate based on the data contained in the EventSpeedChange instance and creates a NoticeSpeedChange to send to the Viewer

The algorithm for the Model to parse events is the following:

1. The Model checks if its EventQueue is not empty
2. While the EventQueue is not empty, the Model pops the front Event in the Queue and checks the type of Event

3. The Model acts according to current type of Event and the above table.

4 The Controller

The Controller coordinates interactions between the Viewer and the Model. The Controller receives Input instances from the Viewer. Depending on what kind of Input is being processed, the Controller creates a corresponding type of Event instance and sends it to the Model to instruct the Model to modify itself in some way.

Input	Event
InputClose	EventClose
InputSpacebar	EventPause
InputChangeSpeed	EventChangeSpeed

5 The Viewer

The Viewer handles displaying data from the Model on screen and processing user inputs from the keyboard and mouse. The Viewer uses the Simple Fast Multimedia Library (SFML) for graphics and input processing.

5.1 Viewer Input Processing

User Input	**SFML Event**	**Input Type**
Escape key	Event.type == sf::Event::KeyReleased and Event.key.code == sf::Keyboard::Escape	InputClose
SFML Window is closed	Event.type == sf::Event::Closed	InputClose
Spacebar is pressed	Event.type == sf::Event::KeyReleased and Event.key.code == sf::Keyboard::Spacebar	InputSpacebar
User clicks the pause button	Event.type == sf::MouseButtonReleased and mouse coordinates are within button bounds	InputPause and InputChangeSpeed(PAUSE)
User clicks slow down button	Event.type == sf::MouseButtonReleased and mouse coordinates are within button bounds	InputChangeSpeed(LOW)
User clicks speed up button	Event.type == sf::MouseButtonReleased and mouse coordinates are within button bounds	InputChangeSpeed(HIGH)

Raw input is parsed by Viewer.parseSFEvent() which is called by Viewer.pollWindow(). The Viewer polls the window to check for input for a set amount of time every loop iteration before “timing out” and moving on to other tasks.

5.2 Viewer Graphical Update Loop

The Viewer’s primary control loop consist of the following tasks, in order:

1. pollWindow() : check the window for any new inputs
2. update() : process any Notices in the viewer’s queue
3. updateDisplay() : make any changes to window and draw changes to the window

5.3 Viewer Notice Processing

Notice Type	**Description of Action**
NoticePause	Viewer modifies the color of the pause button to reflect the current state of the Sim
NoticeClose	Viewer closes the window
NoticeSpeedChange	Viewer updates the display for the current update rate

5.4 Viewer updateDisplay() method

The Viewer’s updateDisplay method is a straightforward algorithm. When called, it clears the window, updates the strings of each Text object in the UI to display any changes in data from the Model’s DataManager, then redraws each element of the UI to the window before refreshing the window.