**Practical – 2**

**Aim: Write a program to create a Transaction class which can be used by clients in blockchain application**

**Code:**

***Cargo.toml dependency changes***

chrono = "0.4.0" // for DateTime

***lib.rs***

pub mod client;

pub mod transaction;

***main.rs***

use ed25519\_dalek::Keypair;

use simple\_blockchain\_rs::client::Client;

use simple\_blockchain\_rs::transaction::Transaction;

fn main() {

let client = Client::new();

let receiver = Client::new();

let amount = 1.0;

let mut transaction1 = Transaction::new(client.public\_key, receiver.public\_key, amount, None);

transaction1.sign\_transaction(&client);

transaction1.print\_transaction();

println!(

"Transaction 1 signature validation: {:#?}",

transaction1.is\_valid\_transaction()

);

}

***client.rs from practical - 1***

***transaction.rs***

use crate::client::\*;

use chrono::{DateTime, Utc};

use secp256k1::Message;

/// A transaction structure that can be used to record a transaction in the blockchain.

///

/// `sender` contains the public key of the client that is sending the transaction.

/// `receiver` contains the public key of the client that is receiving the transaction.

/// `amount` contains the amount of money that is being sent.

/// `signature` contains the signature of the transaction.

/// `timestamp` contains the time at which the transaction was created.

#[derive(Debug)]

pub struct Transaction {

pub sender: key::PublicKey,

pub receiver: key::PublicKey,

time: DateTime<Utc>,

pub amount: f64,

signature: Option<String>,

}

impl Transaction {

/// This method creates a new transaction.

pub fn new(

sender: key::PublicKey,

receiver: key::PublicKey,

amount: f64,

signature: Option<String>,

) -> Self {

if sender == receiver {

panic!("Sender and receiver cannot be the same.");

}

Self {

sender,

receiver,

time: Utc::now(),

signature,

amount,

}

}

/// This method serializes the transaction into a string.

pub fn serialize(&self) -> String {

format!(

"{}{}{}{}",

self.sender, self.receiver, self.amount, self.time,

)

}

/// This method calculates the hash of the transaction using SHA256.

pub fn calculate\_hash(&self) -> Vec<u8> {

crypto\_hash::digest(crypto\_hash::Algorithm::SHA256, &self.serialize().as\_bytes())

}

/// This method signs the transaction using the private key of the signer.

pub fn sign\_transaction(&mut self, signer: &Client) {

self.signature = Some(signer.sign(&self.calculate\_hash()).to\_string());

}

/// This method prints the signature of the transaction.

pub fn print\_transaction(&self) {

println!("sender: {}", self.sender.to\_string());

println!("receiver: {}", self.receiver.to\_string());

println!("time: {:?}", self.time);

println!("amount: {:?}", self.amount);

if let Some(signature) = &self.signature {

println!("signature: {}", signature);

}

println!("");

}

/// This method verifies the signature of the transaction.

pub fn is\_valid\_transaction(&self) -> bool {

let secp = Secp256k1::verification\_only();

let unsigned\_transaction\_hash =

Message::from\_slice(self.calculate\_hash().as\_slice()).unwrap();

if let Some(\_) = &self.signature {

return secp

.verify\_ecdsa(

&unsigned\_transaction\_hash,

&Signature::from\_str(

self.signature.as\_ref().unwrap\_or(&String::new()).as\_str(),

)

.unwrap(),

&self.sender,

)

.is\_ok();

}

return false;

}

}

**Single transaction data with validation check**

Text, application

Description automatically generated