OFFLINE PAYMENT PROTOCOL FOR MOBILE DEVICES

Dhruv Rauthan

2019A7PS0095G

f20190095@goa.bits-pilani.ac.in

Problem Statement:

To build software which enables offline point-to-point monetary transactions with the use of secure and authorized hardware in mobile devices. This will allow users to make digital payments even while they are temporarily offline and disconnected from the Internet. This software will make use of the OPS protocol.

Related Work:

Towards a Two-Tier Hierarchical Infrastructure: An Offline Payment System for Central Bank Digital Currencies

https://www.researchgate.net/publication/347300453_Towards_a_Two-Tier_Hierarchical_Infrastructure_An_Offline_Payment_System_for_Central_Bank_Digital_Currencies

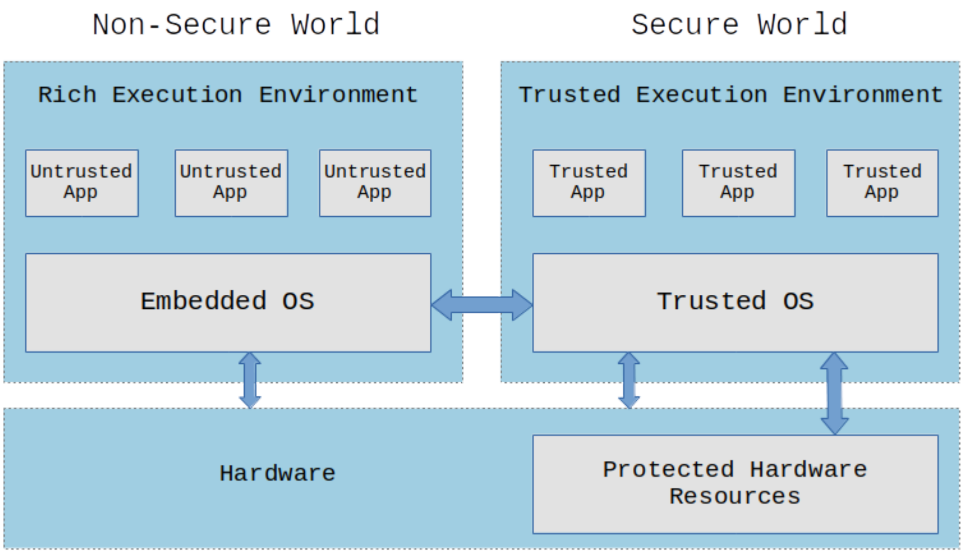
Introduction:

There has been a growing and spreading interest in blockchain technology and its implementations over the past few years. One such application is the ability to make anonymous and secure digital payments. This has prompted agencies all over the world to look into a kind of “central bank money” with which to make these payments. This money is also known as Central Bank Digital Currency (CDBC). Apart from the usual security, reliability and verifiability of the transactions when connected to the internet, it is also important to develop an “offline” capability in which users need not be connected to the internet to make transactions. This creates a system in which users can exchange money in any scenario and place. Hence, the Offline Payment System (OPS) protocol is proposed to grant such a facility. An overview of this protocol is given below.

Trusted Execution Environment:

A Trusted Execution Environment (TEE) is an environment to execute code which has its own physical hardware resources as well as its own software consisting of a separate operating system and trusted applications (TAs) which run on this OS. In a TEE, one has a higher level of trust, security and isolation as compared to the normal OS running on the mobile device. Both of the operating systems run parallelly with each other. Only the TAs have full access to the main processor and all the peripherals of the device. The TAs communicate with external "untrusted" applications (UAs) with the help of APIs provided by the TEE OS.

To prevent external user programs tampering with the hardware, certain private keys are embedded into the chips during manufacturing, which cannot be changed in any way. These keys are used to sign the trusted applications in the device. The hardware does not allow other programs not signed with the key, to access privileged features. Whenever a new application is attested, it is loaded from the untrusted part of the device to the trusted part. The GP model is used in most Android mobile devices today.



TEE helps protect this protocol from financial crimes and other related risks by providing a secure hardware to maintain the security of CBDC and retaining the benefits of offline payments.

OPS Components:

OPS Server TA:

The TA deployed on the server which helps the client to register and set up their online accounts and later helps them manage said accounts

OPS Sender TA:

TA deployed within the TEE on the sender’s device and provides OPS protocol functionalities to access and manage the client’s offline balance securely.

OPS Sender UA:

The UA deployed on the sender’s device which provides the UI for the user which communicates with the TA and also with the server to register the UA and the TA

OPS Receiver UA:

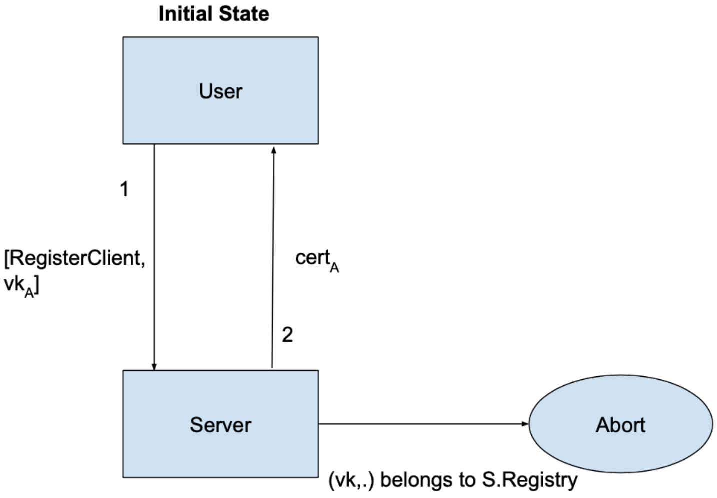
UA deployed on the receiver’s device which provides similar functionalities to the sender UA such as providing the UI to receive and verify the offline payments made to them. This does not interact with the TEE for verification.

Interestingly, we do not need to have a TA on the receiver’s device. Hence, a person can receive money offline without setting up their TA. However, they will need to convert the offline balance to online as soon as they are connected to the server. This means that they cannot use the offline balance in further transactions. This is explained further in the claim and collect protocols.

Client Setup:

Every user needs to be registered with the server before participating in any transactions, online or offline. This involves establishing cryptographic keys and certificates and also initialising the TEE software.

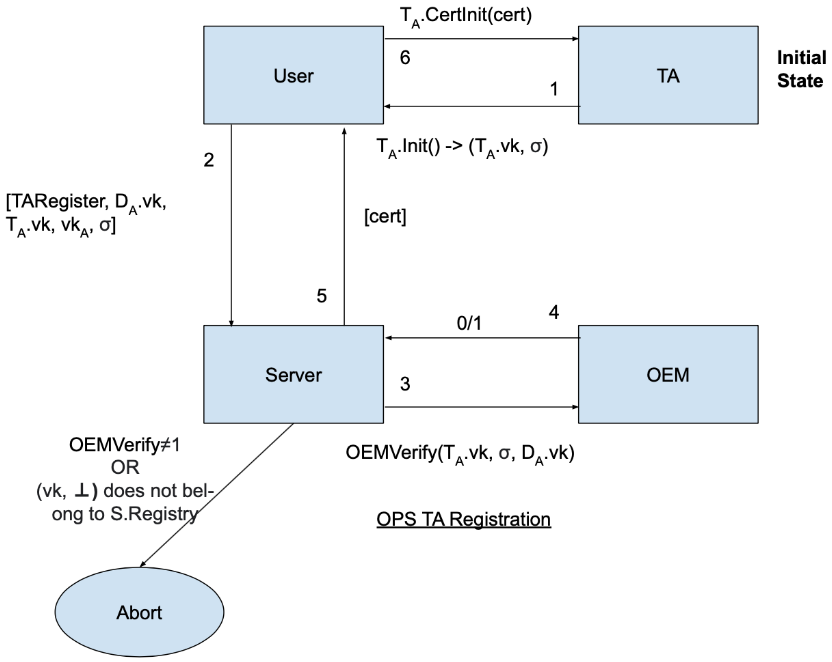
The client generates key signing pairs and sends these to the server. The server checks whether the client is already registered or not, and responds accordingly. If they are not yet registered, the server sends back a certificate assigned to the key signing pair to the client. The client uses this certificate later to authenticate its registration with the server.



Client Setup

Trusted Application Registration:

After verification of the TEE and checking whether the TA is installed properly within the environment, the TA needs to be registered with the server as well. The TA generates a key signing pair and sends it to the device. The client then forwards this along with its own information to the server. The server verifies the attestation using OEMVerify and returns the certificate to the client which is again forwarded to the TA. This certificate activates the TEE to allow for offline payments and is sent during payments to verify the validity of the payment made.The server and the TA keep track of a counter ‘i’ which must be in sync with each other. This ensures that the client does not deposit/withdraw money more than once by spoofing packets.

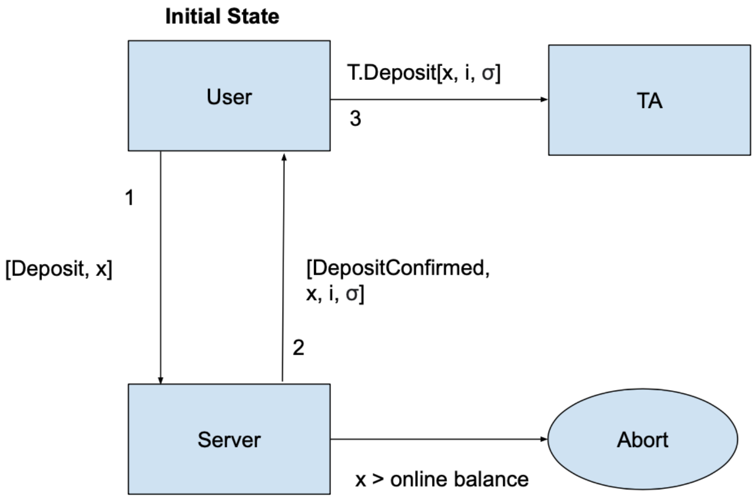


OPS TA Registration

Deposit and Withdraw Protocols:

These protocols are used to convert between online and offline balances, i.e, the online balance maintained by the server and the offline balance maintained by the TA. The client device sends a request to the server. The server checks whether they have the required online balance and sends back the confirmation to the client. This confirmation contains the counter 'i' apart from the basic amount. The client then invokes the TA which checks the counter 'i' to be in sync and then adjusts the offline balance accordingly.

The deposit protocol is shown below:

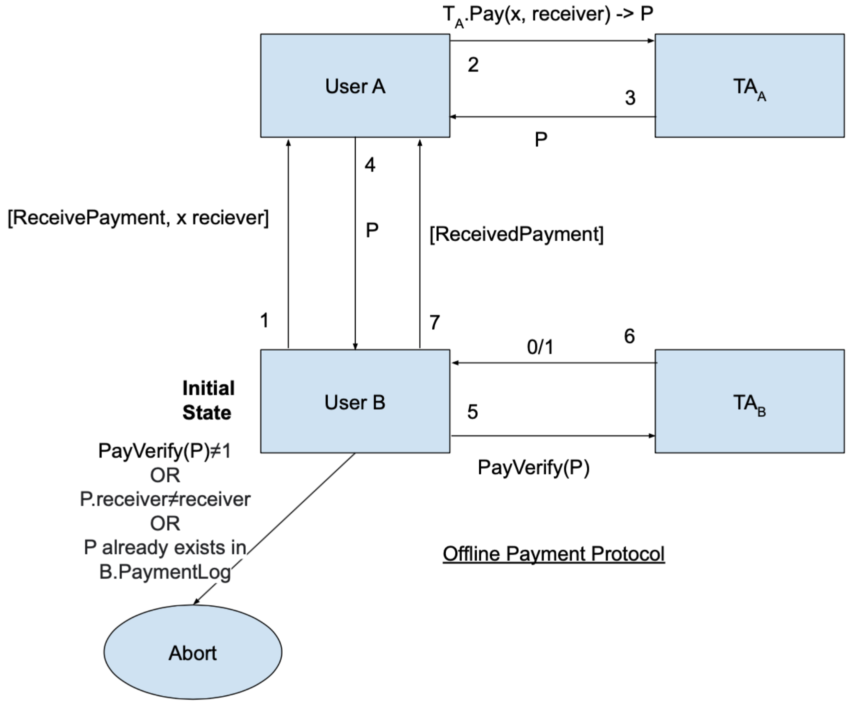


Deposit Protocol

Offline Payment Protocol:

This is the actual protocol where after all the setup, the transaction is actually taking place. The receiver sends a request to the sender. This request contains relevant information such as the amount and its certificate, which is forwarded to the sender TA. The TA checks for the availability of funds with the sender. If yes, it deducts the amount from the offline balance and sends a payment confirmation containing the sender certificate and a payment counter ‘j’ (similar to the counter ‘i’ but for payments) to the sender which is forwarded to the receiver. On receiving the payment message, it verifies the message using PayVerify (which can be done without a TA as well) and adds it to its payment log accordingly. The receiver then sends back a confirmation message to the sender.

To claim the money, the receiver can invoke either the claim or the collect protocol which are discussed below.

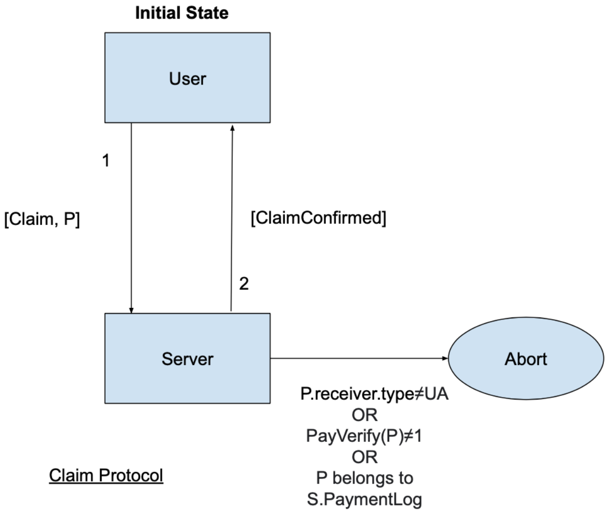


Offline Payment Protocol

Claim and Collect Protocol:

Claim:

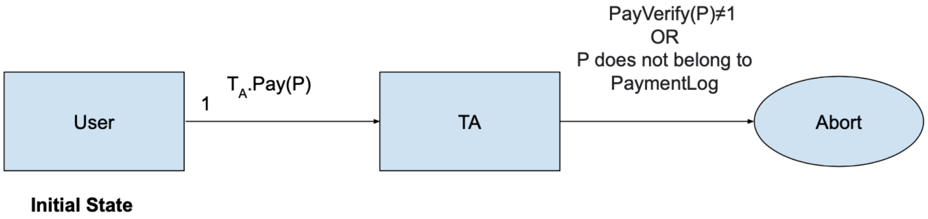
Users who are not TEE enabled and have received payments have to invoke this protocol in order to add money to their online balance. The user sends a packet to the server, containing the payment details which it had received during the offline payment protocol. The server verifies this payment and accordingly adjusts the user's online balance, sending back a confirmation to the user as well.



Claim Protocol

Collect:

If the user who received the payment is TEE enabled, they can choose to add the money received to their offline balance as well, for further transactions. They just need to invoke the TA by sending the payment message, the TA verifies it and adjusts the offline balance accordingly, sending back a confirmation to the user.



Collect Protocol

Implementation:

Background:

We decided to implement the untrusted part of the application, while making a separate Java class to simulate for the trusted application's behaviour. This allows us to have a basic understanding of the untrusted application while also creating a skeleton app for the protocol.

The app simulates both the sender and the receiver, by storing the data of all the clients locally in a Room database.

Further, we also eliminate the need for having a separate server for now, by creating a local server containing the details for the users that we create.

Currently the app has 2 subprotocols implemented. The client setup and the offline payment protocol.

Classes:

The model classes used in this app are the following:

Client

This class represents a user of the app, who will eventually take part in transactions. It is stored and updated locally

Variables used:

Name	Type	Description
id	String	Used to uniquely identify the client during the Transaction Protocol
balance	int	Indicates current balance of client
privateKey	String	Secret key of the client
publicKey	String	Verification key of the client
certificate	String	Certificate of the client received during the Client Setup protocol
inPaymentLog	ArrayList<PaymentPacket>	Stores a list of the payments received

PaymentPacket

This class represents a payment packet P which is returned by the Pay(x, receiver) method of the TA

Variables used:

Name	**Type**	**Description**
amount	int	Transaction amount exchanged between the sender and the receiver
sender	String	Certificate of the sender
receiver	String	Certificate of the receiver
index	int	Used to store the counter j (Unused in this app)
signatureBytes	byte[]	Signature bytes of the data
sign	Signature	Signature of the variables of the class with the private key of the trusted application (T.sk)

RegisteredClient

This class is the client which is stored with the server. It is similar to the Client, except it only has the information which is required by the server

Variables used:

Name	**Type**	**Description**
id	String	Used to uniquely identify the client during the Transaction Protocol
balance	int	Indicates current balance of client
publicKey	String	Verification key of the client

RegistrationPacket

It represents the initial packet sent by the client to the server during the Client Setup protocol.

Variables used:

Name	Type	Description
type	String	String containing the value “RegisterClient” to let the server know the function of the message
clientPublicKey	String	The public key of the client who wants to be registered with the server

Server

This class represents the server in the overall protocol. Currently it only supports the functionalities for the client setup, but more methods can be incorporated into it to accommodate the other protocols.

Variables used:

Name	Type	Description
single_instance	Server	Used to ensure that the server only has one instance running throughout the application. This is similar to a Java singleton class
TAG	String	Used for debugging

mClientDatabase	ClientDatabase	The local client database. It stores values in the form Client.java objects. This variable is present in the Server class because the local as well as the server database need to be updated simultaneously
mServerDatabase	ServerDatabase	The server database which holds records of the RegisteredClient.java class
mClientDao	ClientDao	A data access object used to access the client database
mServerDao	ServerDao	A data access object used to access the server database
mKeyPair	KeyPair	Stores the public and private of the server in a KeyPair object

Methods used:

Name	**Parameters**	**Return type**	**Description**
Server	Context context	-	Constructor method used to initialise all the variables values in the class
getInstance	Context context	Server	Ensures that only a single instance of the class is present throughout the app. If an instance of the class is present

			already, it returns that instance. Otherwise, it creates a new object
checkPublicKeyExists	String publicKey	boolean	Returns true if the public key string exists in the server database
registerClient	String userID, KeyPair keyPair, TextView mProgressTextView	void	This method creates Client and RegisteredClient objects and saves them in the client and server database respectively. After adding the objects to the databases, it invokes the method to create a certificate for the client
createClientCertificate	PublicKey publicKey, Client client, TextView mProgressTextView	void	It creates and assigns a certificate for the newly registered client. It sets the certificate field and updates the client object in the local client database
checkClientIdExists	String userID	boolean	Checks if the client ID entered by the user already exists in the database or not
getClient	String id	Client	Returns the Client object corresponding to the given ID

TransactionPacket

This class represents the message packet sent by the receiver to the sender during the transaction protocol

Variables used:

Name	**Type**	**Description**
type	String	String containing the value “RequestPayment” to let the client know the function of the message
amount	int	The amount requested by the receiver of the payment
receiver	String	The receiver’s certificate received during its registration with the server

TrustedApplication

This class simulates the behaviour of a trusted application of the sender’s device. It contains the required variables and methods in a normal TA.

Variables used:

Name	**Type**	**Description**
clientID	String	The ID entered by the user during registration
privateKey	String	Secret key of the TA stored
publicKey	String	Verification key of the TA stored
balance	int	Indicates the current balance of the client as stored in the TA

i	int	Represents the counter i used to verify deposits/withdrawals (Unused in this app)
j	int	Represents the counter j used to verify the transactions (Unused in this app)

Methods used:

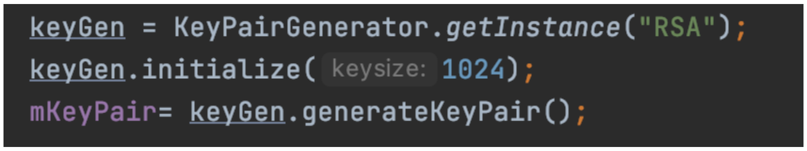
Name	**Parameters**	**Return type**	**Description**
TrustedApplication	String clientID, String privateKey, String publicKey, int balance	-	Constructor method used to initialise all the variables values in the class
pay	TransactionPacket transactionPacket, Client sender, Client receiver, TextView mProgressTextView, Context context	PaymentPacket	The method simulates TA.Pay(x, receiver). It carries out the steps defined in the protocol. If the checks pass, the balance of the sender is deducted, the value updated in the local database, and a payment packet P is created to be returned back to the receiver.

Client Setup Protocol

The user enters a random id value in the EditText field and clicks on the “Register Client” button to register the client in the local as well as the server’s database. This invokes the registerClient() method in ClientRegistrationActivity.java.

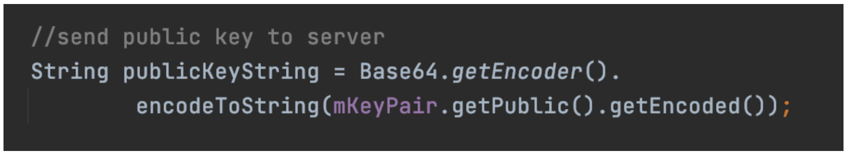
First, it is checked if the entered value exists in the local database, i.e, if it has been used for some other user. This is not a part of the protocol, but instead a part of our own implementation since we are not yet deploying the protocol on a network.

If the value does not exist in the local database, a keypair is generated for the user using Java’s KeyGen classes.



```
keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize( keysize: 1024);
mKeyPair= keyGen.generateKeyPair();
```

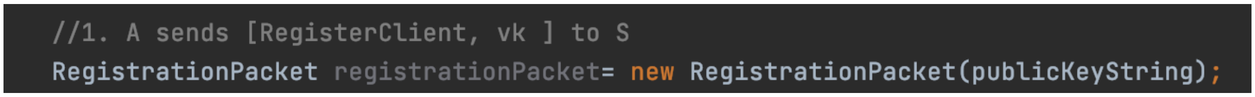
This creates an RSA keypair and assigns it to the variable mKeyPair.
The public key is converted to a string format, since the Room database in Android needs separate type converters to store special values such as KeyPair. So, we convert the keypair to a string to allow for straightforward storage of its value.



```
//send public key to server
String publicKeyString = Base64.getEncoder().
        encodeToString(mKeyPair.getPublic().getEncoded());
```

1. Client A sends [RegisterClient, vk_A] to server S

A RegistrationPacket containing the keyword “RegisterClient” and the public key (vk) of the client is created.

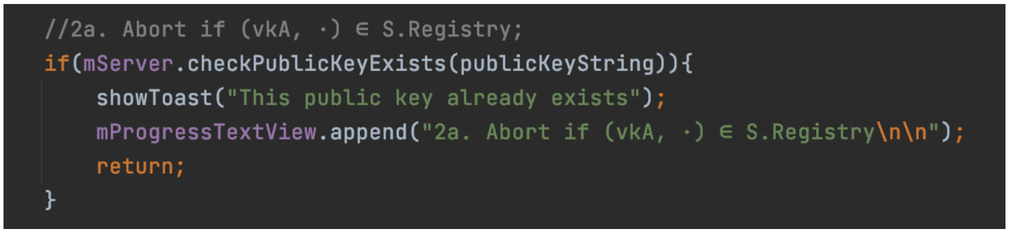


```
//1. A sends [RegisterClient, vk ] to S
RegistrationPacket registrationPacket= new RegistrationPacket(publicKeyString);
```

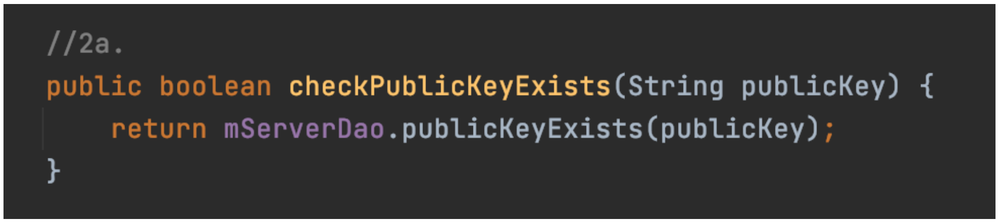
2. Upon receiving the [RegisterClient, vk_A], S performs the following steps:

(a) Abort if (vk_A, .) ∈ S.Registry

The server's checkPublicKeyExists() method is invoked. This checks the server's database if this public key is already registered with some other client by invoking the appropriate method of the ServerDao interface.



```
//2a. Abort if (vkA, ·) ∈ S.Registry;
if(mServer.checkPublicKeyExists(publicKeyString)){
    showToast("This public key already exists");
    mProgressTextView.append("2a. Abort if (vkA, ·) ∈ S.Registry\n\n");
    return;
}
```



```
//2a.
public boolean checkPublicKeyExists(String publicKey) {
    return mServerDao.publicKeyExists(publicKey);
}
```

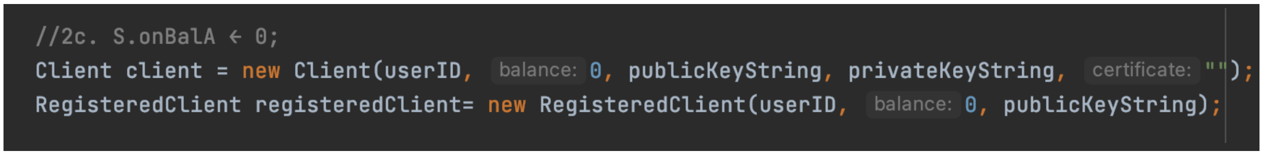
(b) Add (vk_A, ⊥) to S.Registry and (c) $S.onBal_A$ <= 0

If the public key does not exist in the database, we invoke the registerClient() method in the Server.java class.

Here, 2 objects are created

The first is the Client object, which is stored in the local database for our own reference. This object is initialized with the user ID, balance=0, the private and public key strings, and an empty string for the certificate which will be set later.

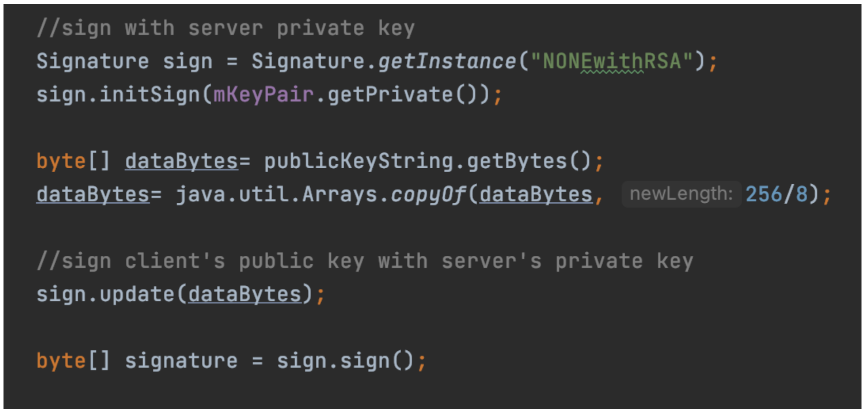
The second is the RegisteredClient object, which is stored in the server's database. This only contains the user ID, balance =0 and the public key of the client according to the protocol.



```
//2c. S.onBalA ← 0;
Client client = new Client(userID, balance: 0, publicKeyString, privateKeyString, certificate: "");
RegisteredClient registeredClient= new RegisteredClient(userID, balance: 0, publicKeyString);
```

(d) Create cert$_A$ such that cert$_A$.vk <= vk$_A$ and cert$_A$.sign <= Sign(vk$_A$, sk$_S$)

After adding the client to the registry, the client certificate generation is done by invoking the createClientCertificate() method.
The signature is create using the Signature java class

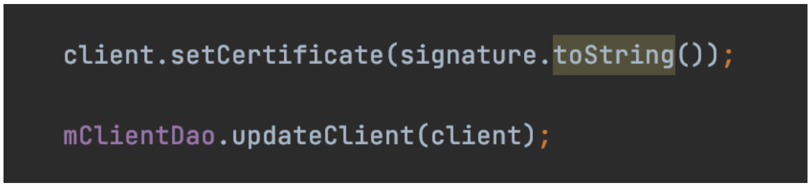


```
//sign with server private key
Signature sign = Signature.getInstance("NONEwithRSA");
sign.initSign(mKeyPair.getPrivate());

byte[] dataBytes= publicKeyString.getBytes();
dataBytes= java.util.Arrays.copyOf(dataBytes, newLength: 256/8);

//sign client's public key with server's private key
sign.update(dataBytes);

byte[] signature = sign.sign();
```

Here, the public key of the client is being signed with the private key of the server. The string is converted into an array of bytes and the signature output into a byte array "signature".

(e) Send cert$_A$ to A

Since we are not implementing the protocol on a network, this step is skipped. We directly assign the certificate to A using a setCertificate() method. The certificate is assigned to the client's certificate variable and the local database is updated accordingly.



```
client.setCertificate(signature.toString());

mClientDao.updateClient(client);
```

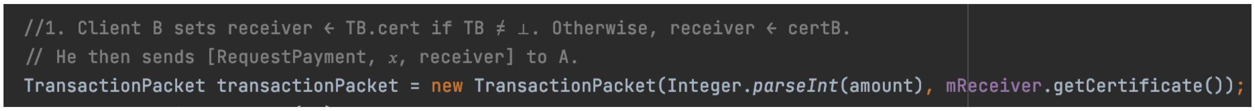
Offline Payment Protocol

We have 3 fields on the screen to fill in here. The amount that the receiver wants to request, the sender ID and the receiver ID. Once the user clicks the "Receive" button, the transaction starts. This is initiated by calling the requestPayment() method in the TransactionActivity.java class.

We follow a similar procedure to the one in the Client Setup Protocol, where the fields are checked for empty values

1. Client B sets receiver ← T_B.cert if $T_B \neq \perp$. Otherwise, receiver ← cert$_B$. He then sends [RequestPayment, x, receiver] to A.

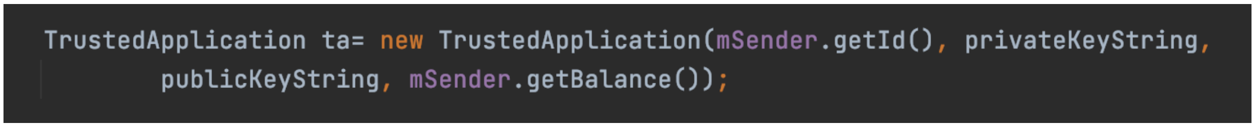
An object of the TransactionPayment.java class represents a packet [RequestPayment, x, receiver] in the protocol. Since the trusted application part is not fully functional, the receiver value is set to the certificate that B received during the Client Setup protocol.



```
//1. Client B sets receiver ← TB.cert if TB ≠ ⊥. Otherwise, receiver ← certB.
// He then sends [RequestPayment, x, receiver] to A.
TransactionPacket transactionPacket = new TransactionPacket(Integer.parseInt(amount), mReceiver.getCertificate());
```

After completing step 1, we “create” the trusted part for the sender. A simple Java class suffices for the TA, which is named TrustedApplication.java.

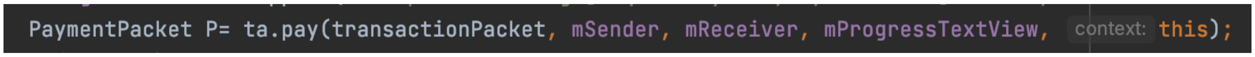
For generating the keys for the TA, a similar procedure is followed as in the Client Setup protocol. A new TrustedApplication object contains the sender’s ID, its private and public key. The other variables of the TA, such as the balance, the i counter and the j counter are all initialized to 0.



```
TrustedApplication ta= new TrustedApplication(mSender.getId(), privateKeyString,
        publicKeyString, mSender.getBalance());
```

2. Upon receiving [RequestPayment, x , receiver] from B, client A sends P <= TA.Pay(x, receiver) to B

Here, the pay() method of the TrustedApplication.java class is invoked and the return value of the payment packet P is stored in an appropriate variable.



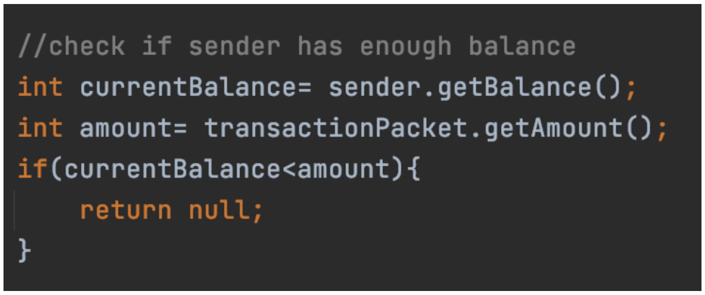
```
PaymentPacket P= ta.pay(transactionPacket, mSender, mReceiver, mProgressTextView,  context: this);
```

TA.Pay(x, receiver)

1. Abort if T.cert = ⊥ or T.bal < x

The certificate is skipped since only the essential parts of the TA have been implemented in this application.

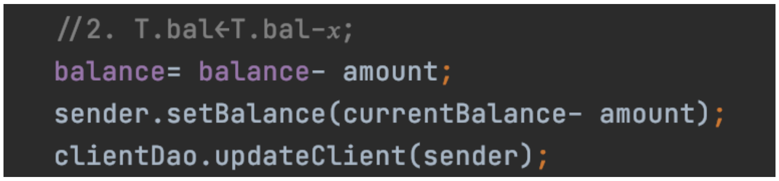
The sender’s current balance is returned using the getter method of the Client class. We compare it with the amount passed in the parameters of the method. If the sender does not have the required balance, a null value is returned which is handled appropriately in the TransactionActivity class.



```
//check if sender has enough balance
int currentBalance= sender.getBalance();
int amount= transactionPacket.getAmount();
if(currentBalance<amount){
    return null;
}
```

2. T.bal <= T.bal - x

If the sender has enough balance, it is deducted from both the TA and the Client object’s balance variables. The sender is then saved in the local database with the updated balance value.



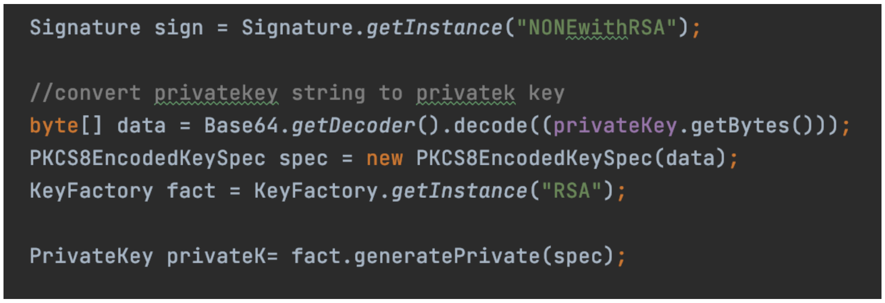
```
   //2. T.bal←T.bal-x;
   balance= balance- amount;
   sender.setBalance(currentBalance- amount);
   clientDao.updateClient(sender);
```

3. T.j <= T.j + 1

The counter is not yet implemented

4. P.amount <= x; P.sender <= T.cert; P.receiver <= receiver; P.index <= T.j and 5. Output P, where P.sig <= Sign([P.amount, P.sender. P.receiver, P.index], T.sk)

The TA’s private key needs to be converted back to a PrivateKey type variable from the String type that it has been saved in. For this, we use the following code:

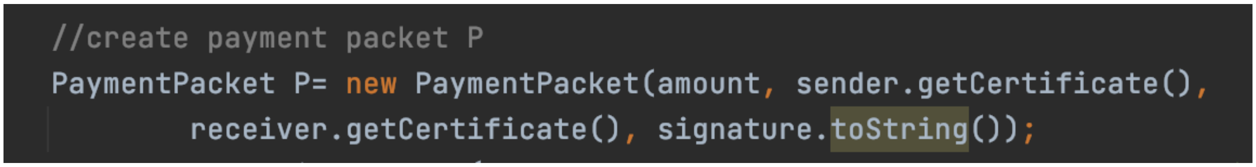


```
Signature sign = Signature.getInstance("NONEwithRSA");

//convert privatekey string to privatek key
byte[] data = Base64.getDecoder().decode((privateKey.getBytes()));
PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(data);
KeyFactory fact = KeyFactory.getInstance("RSA");

PrivateKey privateK= fact.generatePrivate(spec);
```

privateK is the private key of the TA, in its correct format.

We create an arraylist containing the amount value, the sender's certificate and the receiver's certificate. This arraylist is converted into a string object and then into an array of bytes. These data bytes are then signed with the TA's private key (T.sk).

The output P is an object of a PaymentPacket.java class. It contains the amount, the sender's certificate, the receiver's certificate and signature obtained by signing the arraylist with the private key. This P is returned from the method.



```
//create payment packet P
PaymentPacket P= new PaymentPacket(amount, sender.getCertificate(),
        receiver.getCertificate(), signature.toString());
```

In TransactionActivity.java the output PaymentPacket is received and stored in a variable P. If P is equal to null, it means that the method exited early due to the sender not having the appropriate balance. A small Toast message is shown indicating the same.

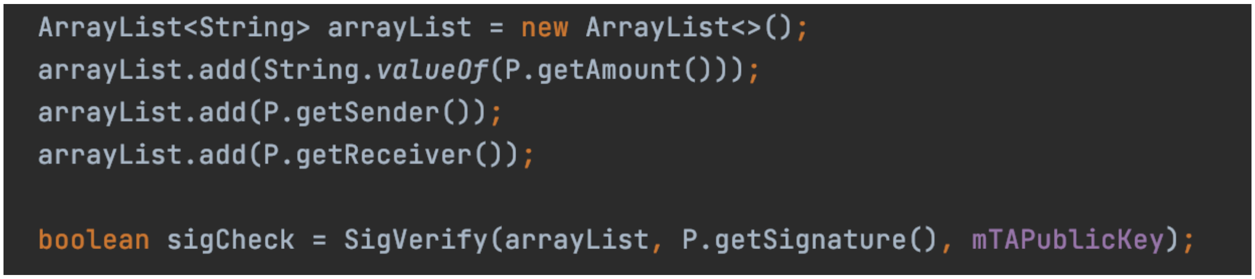
The payment packet P is to be sent back to the receiver here, but this step has not been implemented yet due to no network of devices.

3. Upon receiving P from A, client B performs the following steps:

(a) Abort if any of the following conditions is true:

- *PayVerify(P) ≠ 1,*

We create a new ArrayList with the required values of the amount, the sender's certificate and the receiver's certificate. We then pass this to the SigVerify function along with the signature of the PaymentPacket P and the public key of the TA.



```
ArrayList<String> arrayList = new ArrayList<>();
arrayList.add(String.valueOf(P.getAmount()));
arrayList.add(P.getSender());
arrayList.add(P.getReceiver());

boolean sigCheck = SigVerify(arrayList, P.getSignature(), mTAPublicKey);
```

The PayVerify function is as follows:

Return 1 if and only if all of the following conditions hold:

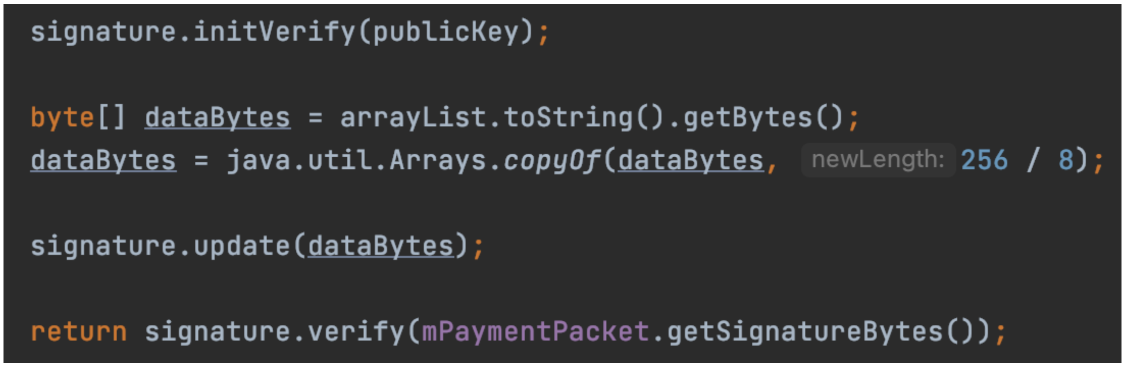
1. TACertVerify(P.sender) =? 1, and

This is to be implemented in the TA

2. SigVerify([P.amount,P.sender,P.receiver,P.index],P.sig,P.sender.vk) =? 1.

We convert the public key string of the TA into a PublicKey type variable with a similar method as used above for the private key

Now, to verify the signature



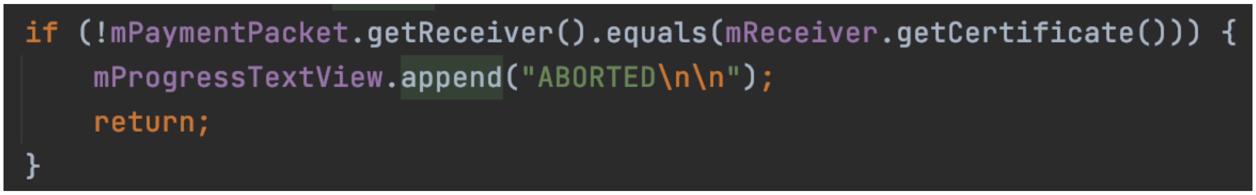
```
signature.initVerify(publicKey);

byte[] dataBytes = arrayList.toString().getBytes();
dataBytes = java.util.Arrays.copyOf(dataBytes,  newLength: 256 / 8);

signature.update(dataBytes);

return signature.verify(mPaymentPacket.getSignatureBytes());
```

We initialize the signature with the public key of the TA and update the data bytes with the ArrayList we passed initially.

The signature.verify() method returns an appropriate boolean value according to the verification of the signature.

- *P.receiver ≠ receiver, or*

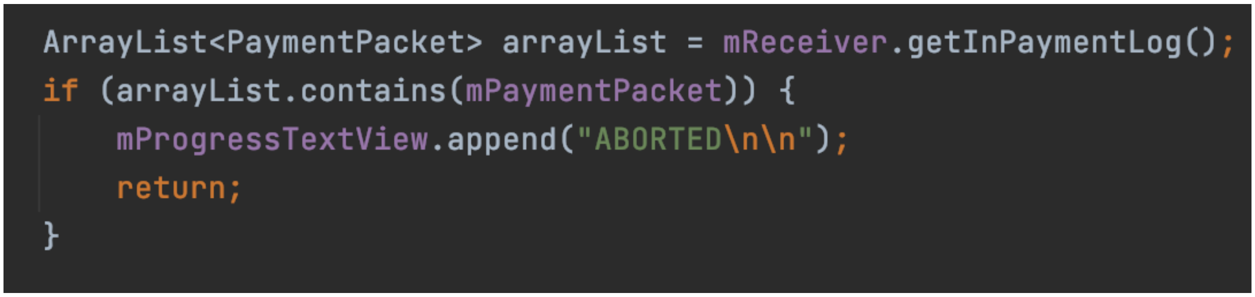
We simply check whether the certificate of the receiver matches the one in the PaymentPacket P and update the TextView accordingly



```
if (!mPaymentPacket.getReceiver().equals(mReceiver.getCertificate())) {
    mProgressTextView.append("ABORTED\n\n");
    return;
}
```

- *P ∊ B.inPaymentLog;*

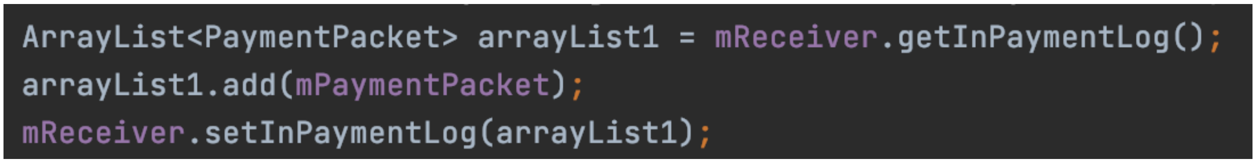
We get the ArrayList using a getter method of the Client class. If the inPaymentLog list already contains the PaymentPacket P we abort the transaction.



```
ArrayList<PaymentPacket> arrayList = mReceiver.getInPaymentLog();
if (arrayList.contains(mPaymentPacket)) {
    mProgressTextView.append("ABORTED\n\n");
    return;
}
```

(b) B adds P to B.inPaymentLog and sends [ReceivedPayment] to A;

Here, we simply add the PaymentPacket to the list and update the receiver.



```
ArrayList<PaymentPacket> arrayList1 = mReceiver.getInPaymentLog();
arrayList1.add(mPaymentPacket);
mReceiver.setInPaymentLog(arrayList1);
```

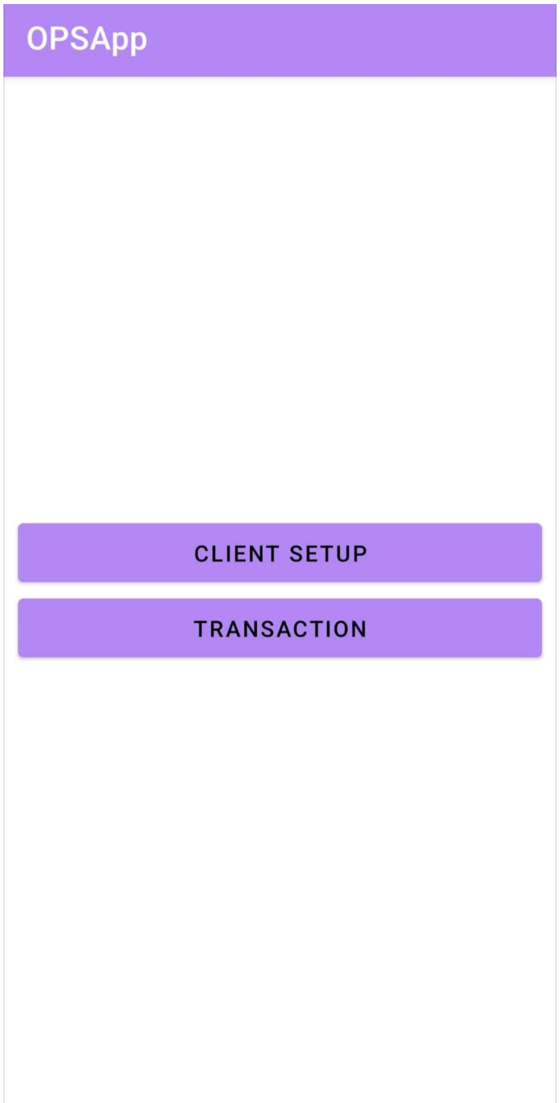
(c) If P.receiver.type = “TA”, then B calls TB.Collect(P);

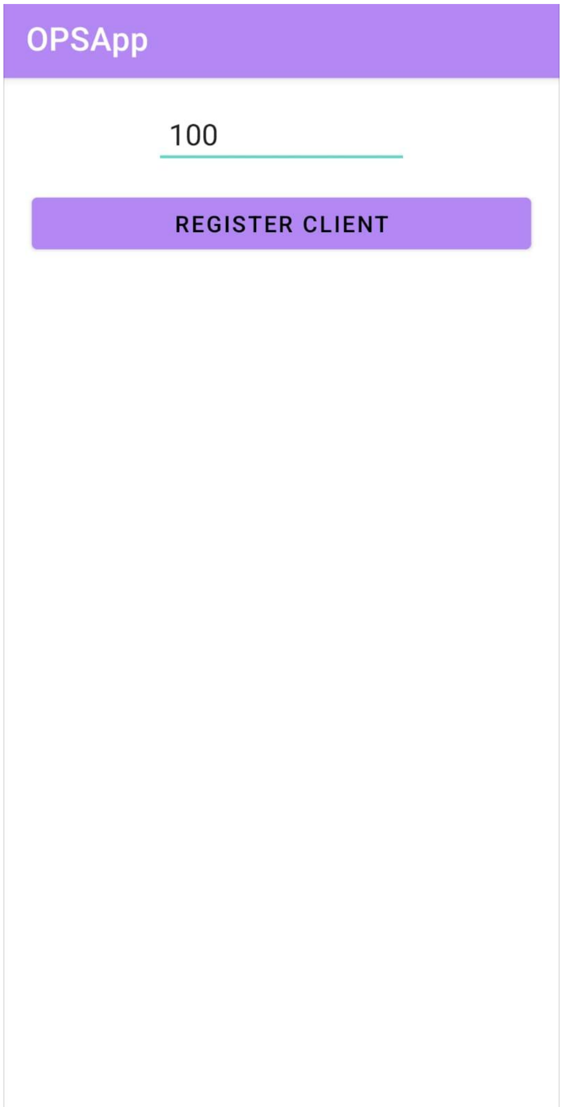
(d) Otherwise, B engages in the Claim protocol with S (Figure 9) as soon as B is online.

The Claim and Collect protocols have not been implemented yet.

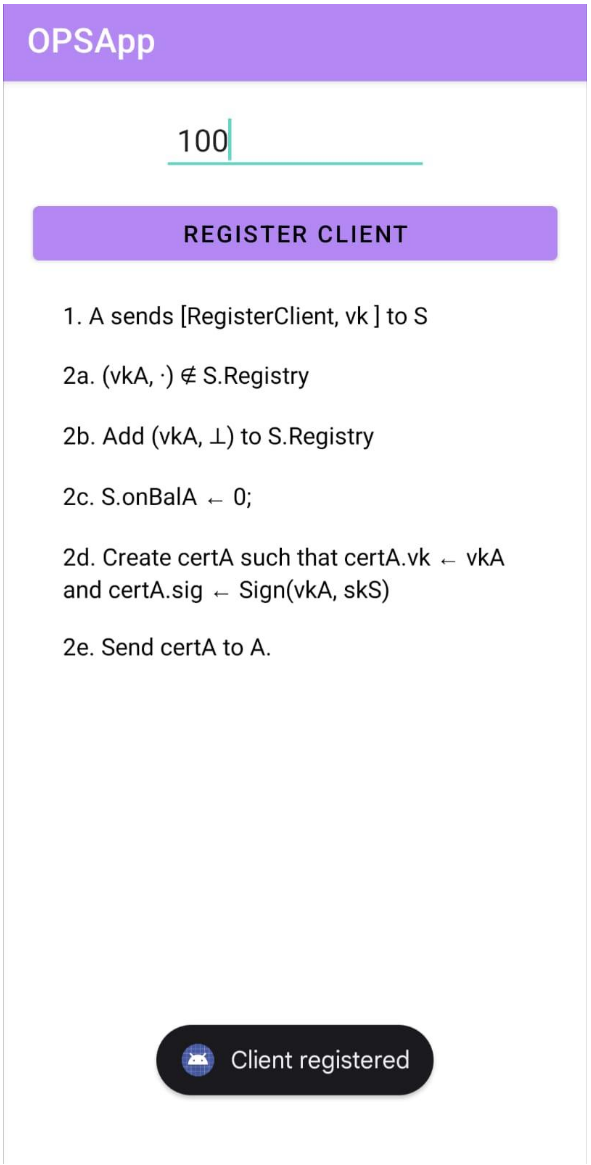
Source Code and Screenshots:

https://github.com/dhruvrauthan/OPSApp





100



100

1. A sends [RegisterClient, vk] to S

2a. $(vkA, \cdot) \notin S.Registry$

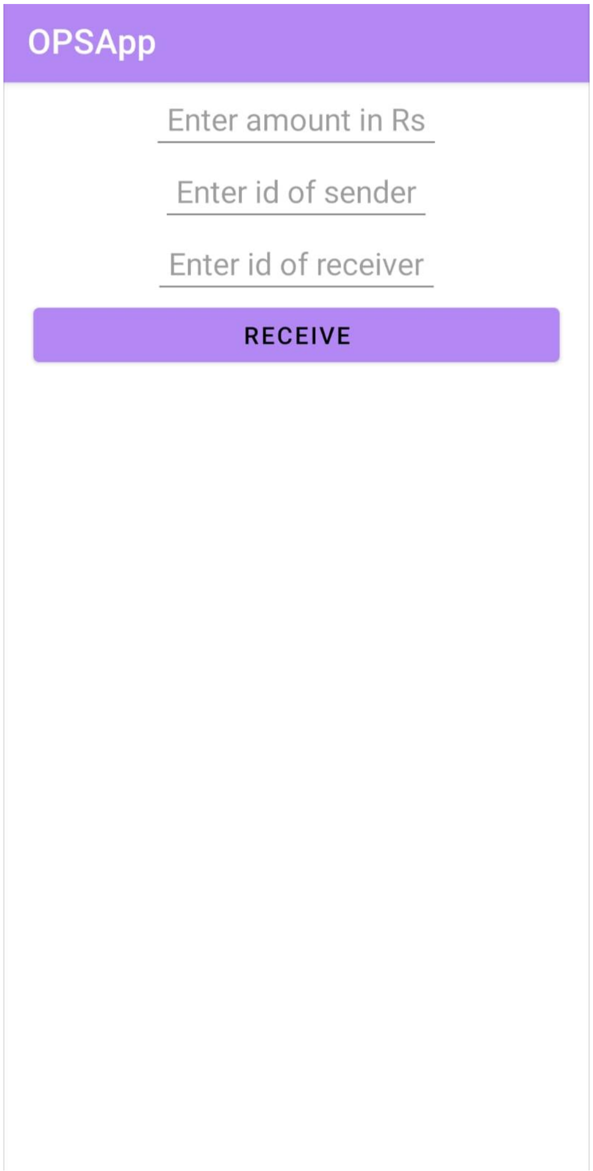
2b. Add $(vkA, \perp)$ to S.Registry

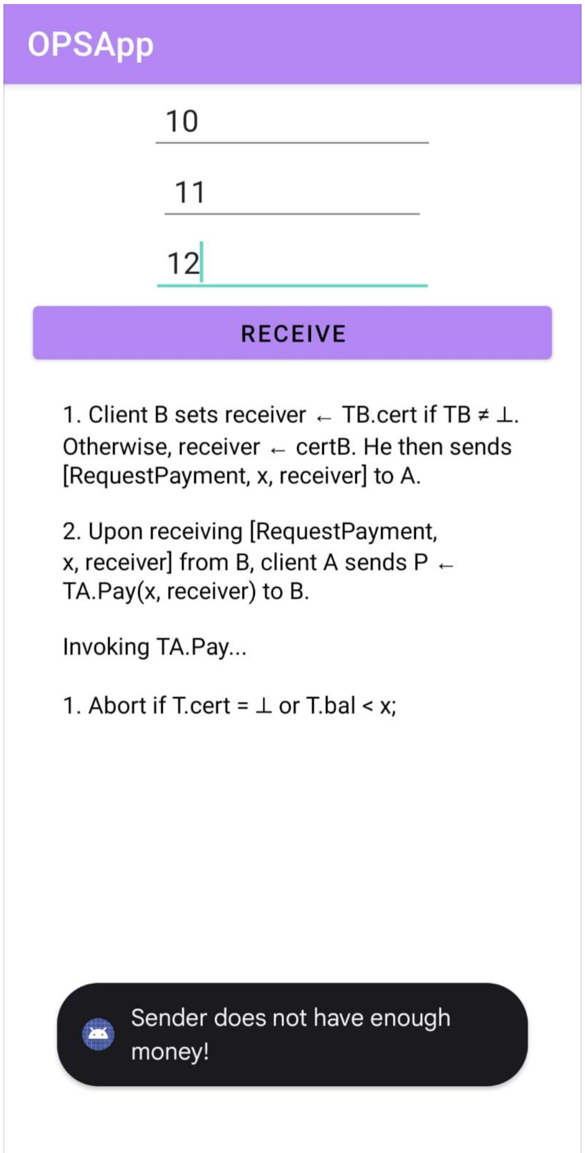
2c. S.onBalA ← 0;

2d. Create certA such that certA.vk ← vkA and certA.sig ← Sign(vkA, skS)

2e. Send certA to A.

Client registered





10

11

12

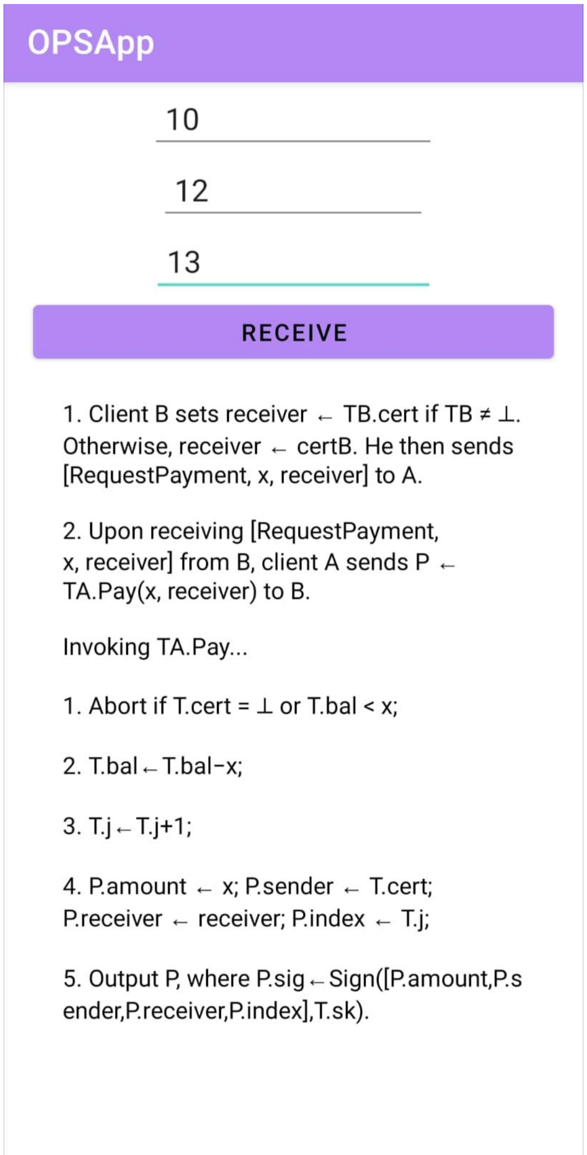
1. Client B sets receiver ← TB.cert if TB ≠ ⊥. Otherwise, receiver ← certB. He then sends [RequestPayment, x, receiver] to A.

2. Upon receiving [RequestPayment, x, receiver] from B, client A sends P ← TA.Pay(x, receiver) to B.

Invoking TA.Pay...

1. Abort if T.cert = ⊥ or T.bal < x;

Sender does not have enough money!



10

12

13

1. Client B sets receiver ← TB.cert if TB ≠ ⊥. Otherwise, receiver ← certB. He then sends [RequestPayment, x, receiver] to A.

2. Upon receiving [RequestPayment, x, receiver] from B, client A sends P ← TA.Pay(x, receiver) to B.

Invoking TA.Pay...

1. Abort if T.cert = ⊥ or T.bal < x;

2. T.bal ← T.bal−x;

3. T.j ← T.j+1;

4. P.amount ← x; P.sender ← T.cert; P.receiver ← receiver; P.index ← T.j;

5. Output P, where P.sig ← Sign([P.amount,P.sender,P.receiver,P.index],T.sk).

Conclusion:

Stable internet connectivity is yet a dream for many and hence, offline payments are an integral part of digital payment systems to be implemented in the future. Authentication and security of these payments are an important feature to be implemented in an offline network.

The Offline Payment Protocol introduces protocols to envision a mechanism to enable such transactions. We explored the proposed protocol and made our own mobile payment application implementing two of the sub protocols, the client setup and the transaction protocol.

Digital payments are definitely the future of transactions, and this is an important step in that direction.

Future Work:

The Deposit/Withdraw and the Claim/Collect protocols are not present in the app. The sub protocols involving the TA have not been implemented yet due to the additional overhead required for the setting up of the TEE and the custom TAs.

References:

- https://developer.android.com/reference/java/security/Signature
- https://developer.android.com/reference/java/security/KeyPair
- https://developer.android.com/reference/java/security/KeyPairGenerator
- https://developer.android.com/training/data-storage/room
- https://www.javatpoint.com/java-digital-signature