Mojaloop: 3rd Party Payment Initiation Proposal for Credit Transactions

Version: 5.0

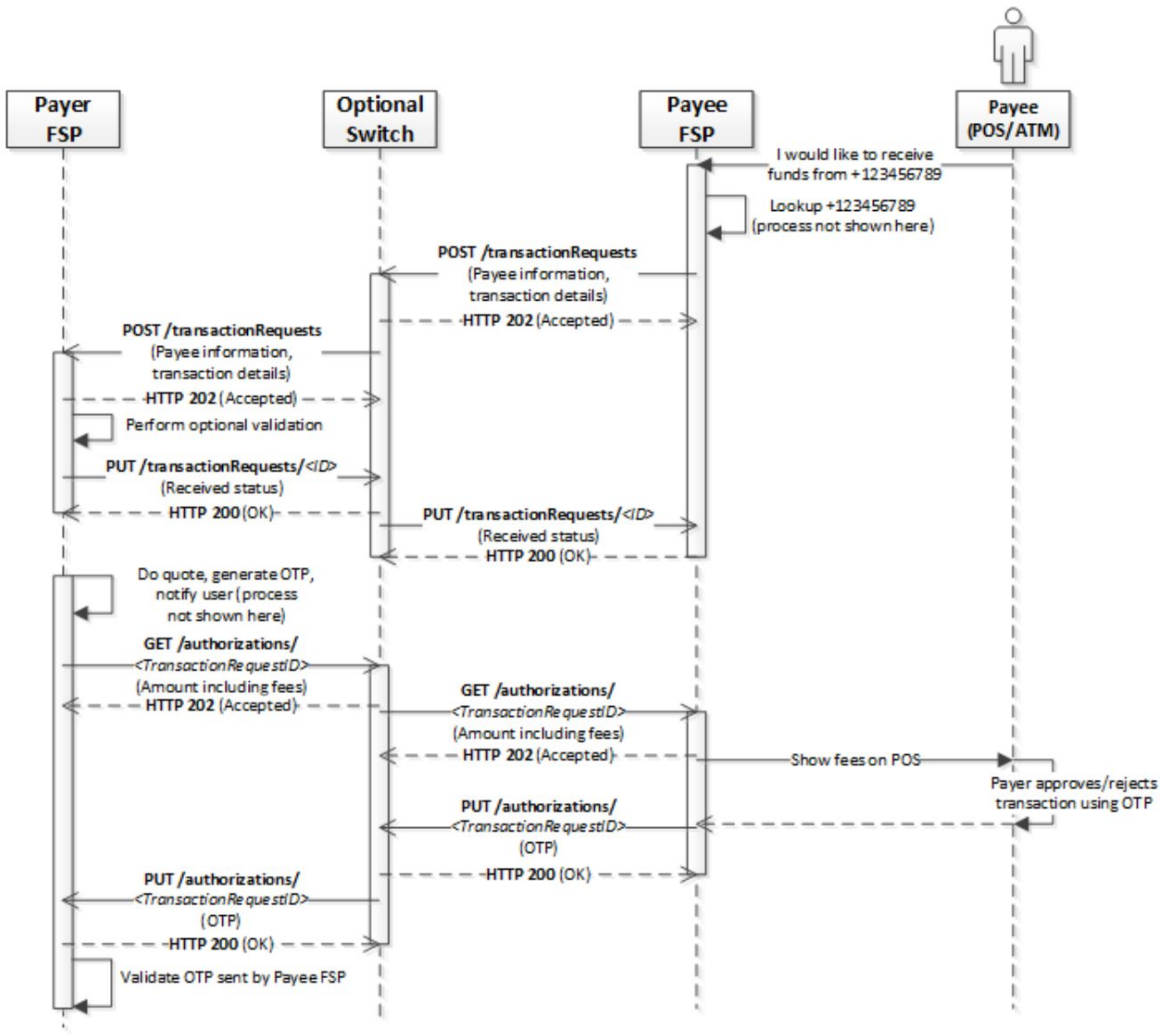
Introduction

Mojaloop is open source software for creating digital payment platforms that connect all customers, merchants, banks, and other financial providers in a country's economy. Its original design assumed that all participating financial service providers (FSPs) keep track of value that customers own by running ledgers. The examples of such ledgers are bank checking/savings accounts, digital wallets, crypto exchange accounts, etc. Today, there are real time payment networks which allow participations from services which don't hold values. The purpose of this document is to propose a set of technical changes that need to be done within the framework of existing Mojaloop design in order to allow participation of such non-value holding parties.

Payment Initiation Service Provider (PISP)

The Mojaloop RTP system facilitates payments flowing between its *participants* - from one financial service provider (FSP) to another.

All supported payment flows originate from a payer or payee *party* which holds a financial account at one of participant FSPs. As payers, such parties either ask their payer-side FSP to send transactions over to the payee's FSP - while as payees, *parties* can initiate a transfer request that flows back to payer and its FSP for approval. There are a number of real-world flows supported by it now, but all its payment flows originate by FSPs on either payer or payee side and all these participants are value-holding institutions with their own ledgers. An example of one such flow, originating from the payee's FSP side (ATM in this case) is here:


Payer FSP
Optional Switch
Payee FSP
Payee (POS/ATM)
I would like to receive funds from +123456789
Lookup +123456789 (process not shown here)
POST /transactionRequests (Payee information, transaction details)
HTTP 202 (Accepted)
POST /transactionRequests (Payee information, transaction details)
HTTP 202 (Accepted)
Perform optional validation
PUT /transactionRequests/<ID> (Received status)
HTTP 200 (OK)
PUT /transactionRequests/<ID> (Received status)
HTTP 200 (OK)
Do quote, generate OTP, notify user (process not shown here)
GET /authorizations/<TransactionRequestID> (Amount including fees)
HTTP 202 (Accepted)
GET /authorizations/<TransactionRequestID> (Amount including fees)
HTTP 202 (Accepted)
Show fees on POS
Payer approves/rejects transaction using OTP
PUT /authorizations/<TransactionRequestID> (OTP)
HTTP 200 (OK)
PUT /authorizations/<TransactionRequestID> (OTP)
HTTP 200 (OK)
Validate OTP sent by Payee FSP

Even though Mojaloop is already flexible enough to cover a number of different money transfer scenarios, it assumes that all network *participants* (FSPs) are holding value of some sort, basically maintaining account ledgers of their own. In addition to such value holding FSPs, this document proposes paths for adding support for **non-value-holding participants**. In PSD2 terminology, such 3rd-party payment initiation services which are referred to as “**Payment Initiation Service Providers**” or **PISPs**. This proposal ‘borrows’ PSD2 terminology which defines PISP as:

> *“PISP is a service to initiate a payment order at the request of the payment service user with respect to a payment account held at another payment service provider.”*

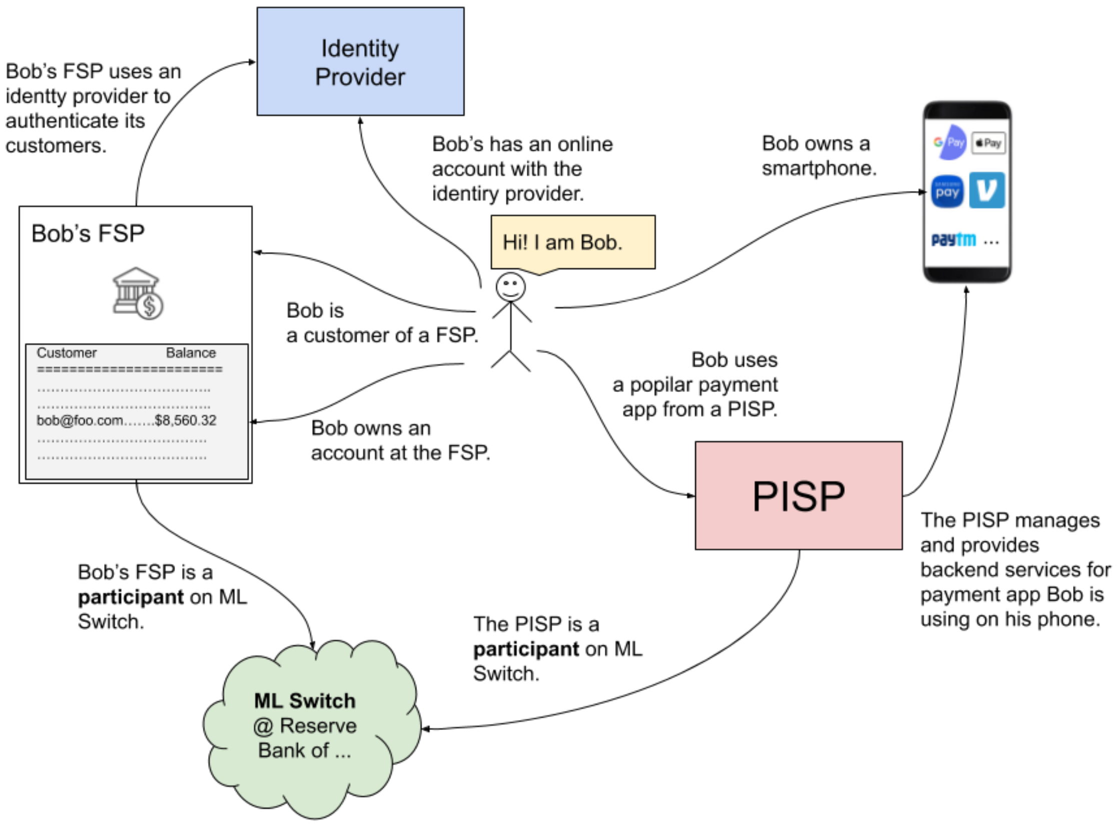
For example, entities such as GPay and PayTM, operate in such non-value holding capacity within India’s UPI payment network.

Party Authentication and Transaction Authorization

PISP Participants on Mojaloop RTP Network

At the Mojaloop Switch level, all PISPs will operate as new type of *participants* within Mojaloop network. The Switch operator will need to explicitly register such service providers and manage their identity and X.509 client certificates. This document won't get into further specifics of participant authorization assuming that Mojaloop's existing documentation covers this topic well.

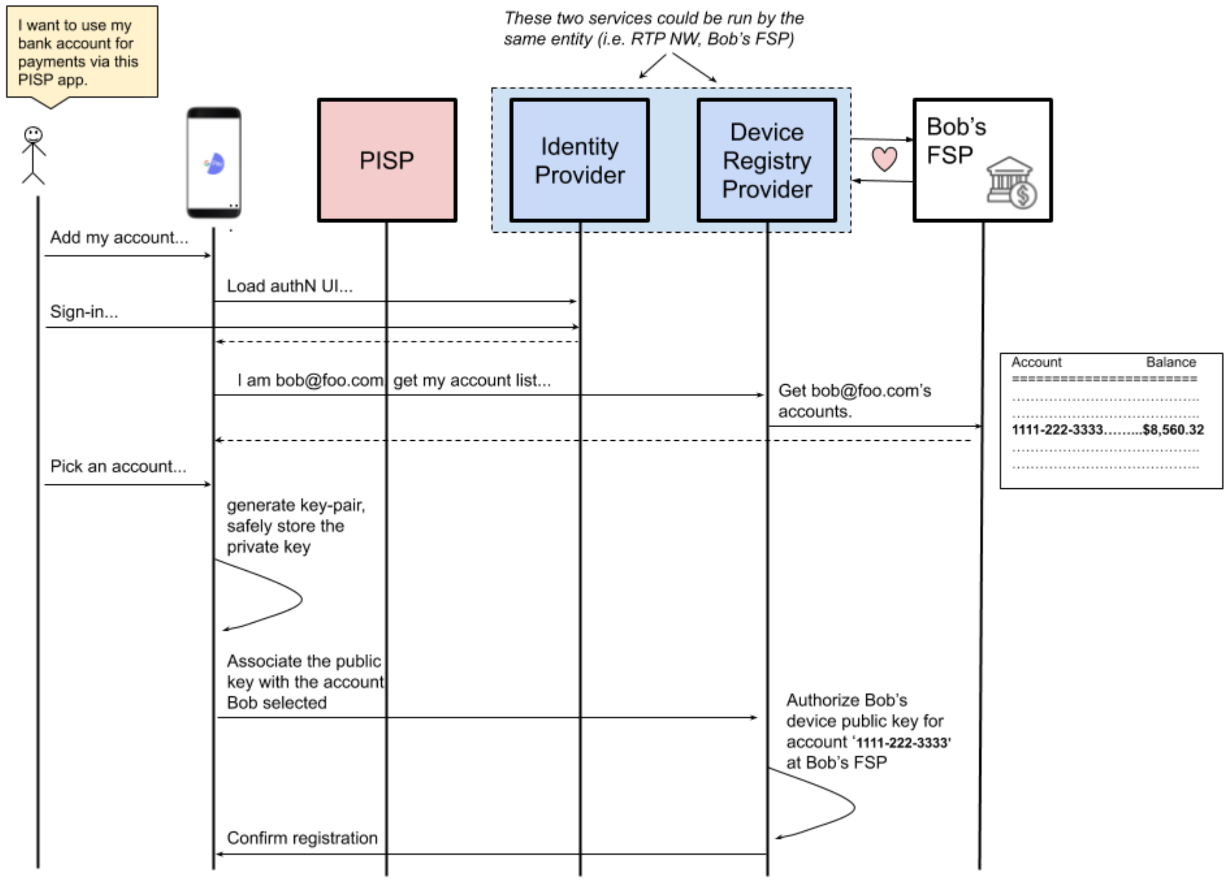
The PISPs might be also subjected to additional call restrictions on Mojaloop Switch API surface, meaning that such *participants* should be able to call only selected methods exposed by the Switch. For example, given that conceptually PISPs hold no value of their own, it does not make sense for such participants to be able to directly call `/transfer` or `/bulkTransfer` (see Mojaloop API documentation for more details).


Identity Provider
Bob's FSP uses an identty provider to authenticate its customers.
Bob's has an online account with the identiry provider.
Bob owns a smartphone.
Hi! I am Bob.
Bob's FSP
Customer Balance
bob@foo.com.......$8,560.32
Bob is a customer of a FSP.
Bob owns an account at the FSP.
Bob uses a popilar payment app from a PISP.
PISP
The PISP manages and provides backend services for payment app Bob is using on his phone.
Bob's FSP is a **participant** on ML Switch.
The PISP is a **participant** on ML Switch.
ML Switch @ Reserve Bank of ...

Registration: User Authentication and Device-Account Pairing

Currently, the Mojaloop Switch operator runs as a closed network where information is exchanged between participant FSP's and the Switch's end-points as Mojaloop API calls. Every transaction initiation or request originates from a ledger-holding FSP - either from the payer or the payee side. The existing Mojaloop design assumes that FSPs will manage end-user (party) authentication independently from their Mojaloop RTP affairs.

With PISPs as new types of participants in this network, there are several new trust related questions which this document will try to answer. The first order of business for end-users would be to register “pair’ PISP app and end-user phone with a financial account held at the FSP.


I want to use my bank account for payments via this PISP app.
These two services could be run by the same entity (i.e. RTP NW, Bob's FSP)
PISP
Identity Provider
Device Registry Provider
Bob's FSP
Add my account...
Load authN UI...
Sign-in...
I am bob@foo.com get my account list...
Get bob@foo.com's accounts.
Account Balance
1111-222-3333.........$8,560.32
Pick an account...
generate key-pair, safely store the private key
Associate the public key with the account Bob selected
Authorize Bob's device public key for account '1111-222-3333' at Bob's FSP
Confirm registration

FSP End-User Authentication in PISP Apps

This proposal assumes that there is a new service called **Identity Provider** (IP) which authenticates end-users (parties) online. The FSPs delegate user authentication to the IP and have a mechanism for mapping IP's representation of the end-user identities to the FSP's internal representation of its customers. There are several choices of IP models that could exist on different RTP networks:

a) Use **FSP's end-user authentication** mechanism as the Identity Provider. This is a simple obvious choice and almost certainly FSP already have some sort of mechanism for online authentication of their own. In this model, each PISP app would need to know at least URL for authentication endpoint for each FSP. The Mojaloop Switch could maintain such data within its participant registration records and provide it to PISPs.

b) Use **3rd party identity provider service** which FSP trusts. The most obvious choice for the identity provider platform are providers based on OAuth2. There are a number of world-wide available identity providers such as Facebook Connect and Google Identity. FSP could allow the use of those IPs for accessing its own site, or maybe just trust them as a reliable source for providing user identifiers that FSP can map to its internal customers. For example in a given network, a 3rd-party identity provider could be trusted to provide a verified citizen ID or its proxy ID which FSP can directly map to its own customers. From the Switch operator side, the upside of this model is that the operator does not have to keep track of all participating FSP authentication end-points and instead just rely on FSPs to accept several sanctioned identity provider platforms.

c) Use **government-run citizens' digital identity provider**. In the counties where there is such a thing as a unique citizen ID and online identity provider service which authenticates its citizens, FSPs could accept its authentication and map it to its internal customer representation. *SingPass* in Singapore and *Aadhaar* in India are good examples of such services. In this model, Mojaloop Switch probably does not have to directly deal with the identity provider at all.

Assuming we have an IP in place (whichever above flavor above or variation of them the switch's environment picks), we can attempt to answer the main question of this chapter. The PISP service operator could integrate with the Identity Provider in a few different ways. First, there could be dedicated shared APIs between them. More likely, the UI flow from the Identity Provider could be directly embedded within the PISP app through mobile app mechanisms like WebView or intent-redirects. In the latter case, user action in PISP app to add an account to your phone would take you to UI which is built and served by the Identity Provider. Once the user is authenticated, the PISP app will receive an authentication token which would be used for further steps in mobile device registration.

PISP App Device Pairing with FSP Financial Account

Similarly to the start of the previous chapter, let's assume that there is yet another service called **Device Registry Provider** (DRP). The DRP's main purpose will be to keep track of registered instances of PISP apps+devices and corresponding financial accounts from FSPs. The DRP will provide following service to PISPs:

1. **Fetch the list of financial accounts** from the FSP for an authenticated user token (provided by the Identity Provider), and then

2. **Record device-generated public key and associate it with selected financial accounts**. The user device will create public-private keypair where the private key will be safely stored with the help of hardware-back mechanism.

Since FSPs will need to use DRP service during Mojaloop transaction authorization call sequence, there are several options on where such service could reside at:

a) **FSPs-operated DRP services.** In this option, each participating FSP would run their own DRP service. An FSPs would initiate public-private key creation on the device for the selected FSP-hosted account and record the generated public part of the keypair as the authorized key for signing the response of a transaction's challenge. Potentially, a skeleton DRP service could be provided as a part of Mojaloop SDK to simplify the work for and also ensure uniform and proper implementation on the FSP side.

b) **The Switch Operator runs "central" DRP service** (or another single "central" authority per RTN). The simplification for FSPs here is that The Switch will be responsible for hosting device/account registration services. Additionally, the Switch could be also implicitly validating transaction authorizations (challenge-response) before routing their replies to FSPs that requested them.

PISP Account Discovery

[WIP] PISP Party Lookup in Mojaloop

To understand all roles of PISPs and the DRP play in different payment flows, we need to first review how Mojaloop's account "addressing" system works. In Mojaloop, the concept of source/destination account as a transaction *party* is defined with the following info:

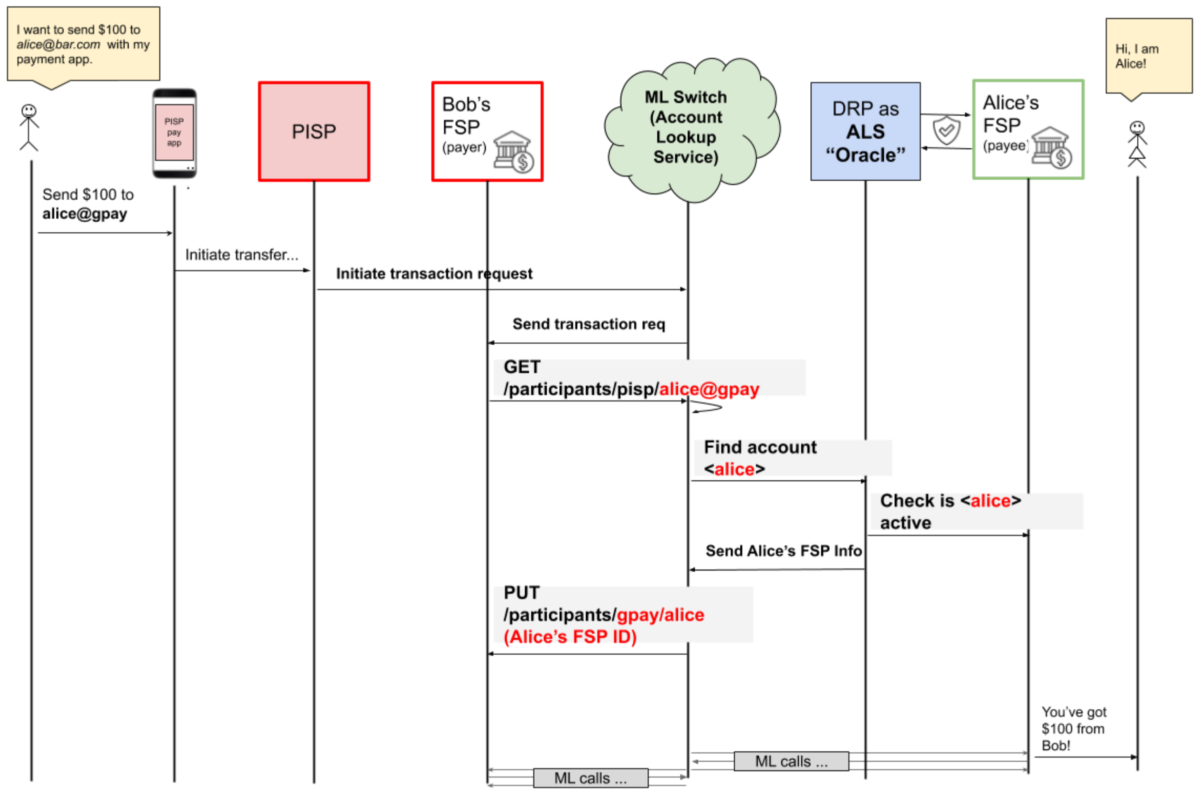
- **partyIdType** - defined a type of an account identifier
- **partyIdentifier** - account identifier (i.e. checking account #, email address, phone #, etc)
- **partySubidOrType** (optional) - additional info that further clarifies either the identifier or the identifier type

Mojaloop's Switch exposed a set of APIs calls collectively called Account Lookup Service (ALS) which are used to find *participants* (FSP) which host the targeted account. In addition to that, the ALS check for the existence of the particular account instance by directly querying qualifying FSPs. This call sequence returns additional account info such as account owner's name and *fspID.* The *fspID* identifies payee/payer institutions in financial transaction flow calls and it's used by the Switch to route API calls to appropriate *participants'* endpoints.

For example, *partyIdType* and *partyIdentifier* defined as `/MSISDN/1234567890` would represent a *party* with phone number of `1234567890`. The ALS will ping all participating parties which registered as potential handlers for type `MSISDN` to find which one actually can handle transactions for phone number `1234567890`.

Similar to that, *partyIdType/partyIdentifier* tuple `IBAN/HR5411111111222222222` would represent account # `222222222` at bank in Croatia (HR) identified within the country by its bank code of `1111111`.

The Mojaloop ALS design is flexible to accommodate adding additional party types and additional oracles which know how to resolve them. Each operating PISP could have its own addressing and address resolution service registered that way. Ideally, the Switch should run a central PISP account resolution "oracle" which will contain all knowledge needed for resolution of all participating PISPs' accounts.



Party Identifiers for PISP Accounts

Even though PISP does not run a ledger with its own value holding accounts, we would like PISP to be able to expose their accounts in Mojaloop as sort of account proxies for actual payer/payee accounts at other underlying FSPs. The PISP accounts would only be able to represent FSP accounts that have been paired with their apps as described in Authentication and Authorization section.

One option is to have each PISP should define its own *partyIdType* that is specific to its service. For example, GPay account could be addressible as:

gpay/<gpay-specific-account-id>/<gpay-specific-subtype-or-id>

Alternatively, a more generic account addressing schema that is PISP agnostic can be used too. Something like:

pisp/<account-handle>

Where <account-handle> would be recorded during the device/account pairing process. This schema gives us an opportunity to have <account-handle> which could be memorable and as such used a conveyance mechanism - for example **alice.smith@gpay**.

DRP as PISP Account Resolution Oracle for ASL

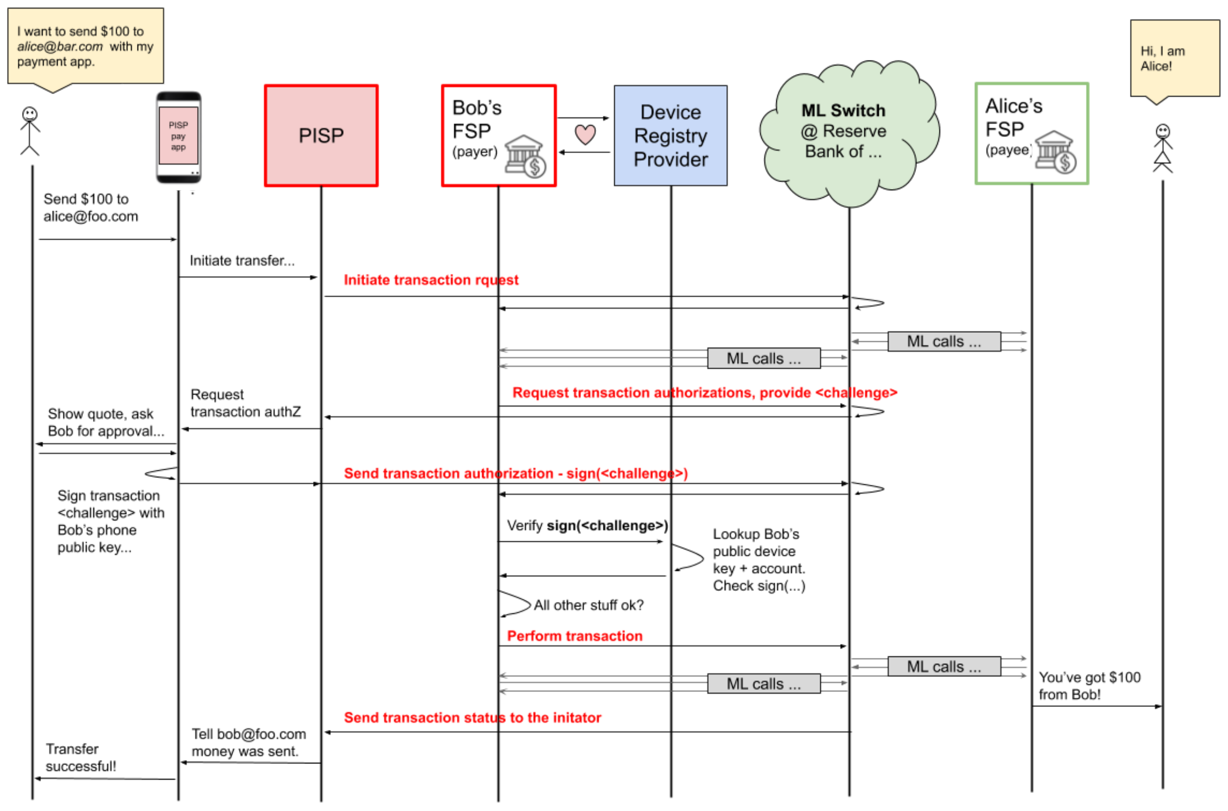
For different values of *partyIdType*, ASL performs the *party* lookup described above by routing to different "oracles" which know how to map account id into relevant FSP information. For example, *partyIdType* of MSIDSN (phone lookup) used an oracle which utilizes PathFinder service to find carrier for a given phone number. From that info, ASL will find registered *participants* that corresponds to found mobile carrier.

An account lookup oracle supporting PISP could be a pretty simple one following the account identity schema from the above. On top of that, PISP Server's endpoint would just need to expose `GET /parties/<id>/<sub-id>` handler which will return relevant account information. Given that the DRP would have an opportunity to record such PISP account association during the device-account pairing process, it seems that the best choice is to have the DRP could serve as an oracle for resolving PISP-paired financial accounts.

Transaction Authorization via PISP Apps

The process of pairing the user’s device with a given financial account will record the device/account public key at the DRP as DAK_{pub}. The user’s device will create a record of that pairing process as well by saving the device’s private key DAK_{priv}, ideally in its internal secure storage hardware component. During the transaction authorization flow, the Mojaloop Switch will route all authorization requests to the user’s device via the PISP which initiated the transaction request. As a part of the transaction authorization request, the FSP will generate a *challenge* which it will expect the user device to sign with its private key.

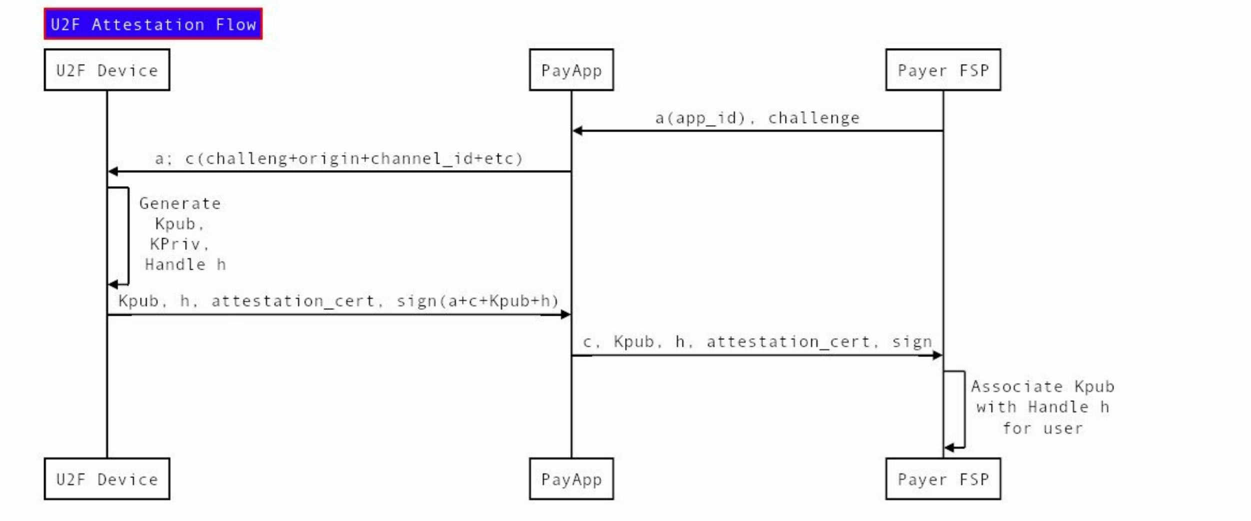
Below diagram is a conceptual depiction of this process, more details follow in the rest of the document.


I want to send $100 to alice@bar.com with my payment app.
PISP pay app
PISP
Bob's FSP (payer)
Device Registry Provider
ML Switch @ Reserve Bank of ...
Alice's FSP (payee)
Hi, I am Alice!
Send $100 to alice@foo.com
Initiate transfer...
Initiate transaction rquest
ML calls ...
ML calls ...
Request transaction authZ
Request transaction authorizations, provide <challenge>
Show quote, ask Bob for approval...
Send transaction authorization - sign(<challenge>)
Sign transaction <challenge> with Bob's phone public key...
Verify sign(<challenge>)
Lookup Bob's public device key + account. Check sign(...)
All other stuff ok?
Perform transaction
ML calls ...
ML calls ...
You've got $100 from Bob!
Send transaction status to the initator
Tell bob@foo.com money was sent.
Transfer successful!

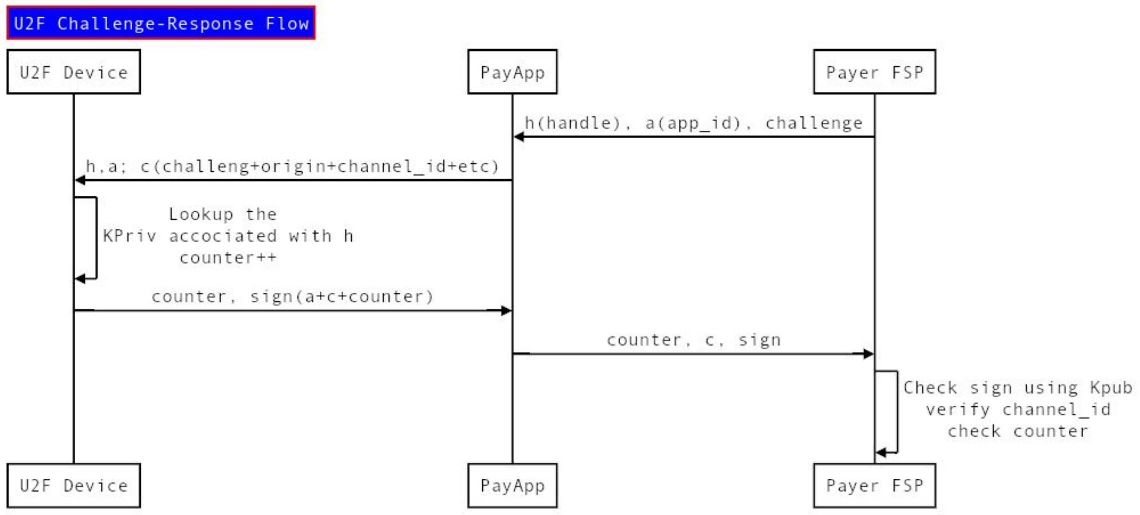
Design Proposal Details

FIDO Universal 2nd Factor (U2F) Basics

The idea behind FIDO's is about having strong hardware-backed crypto key exchange established between the end-user device and an FSP. FSPs will, upon initially authenticating its user in some other way (i.e. web flow), generate a challenge which will be passed to the payment app. The app will then ask FIDO-blessed authenticator device to create a new public-private key pair and a key handle which will be used as the key pair's unique identifier. The FIDO device will then sign the public key, the handle and the challenge with its attestation certificate (which assures the presence of a legit FIDO certified hardware). The private part of the key will never be shared outside of FIDO authenticator device - i.e. it will be kept in TPM, TrustZone or a similar secure storage hardware. Upon receiving it, FSP will verify the attestation certificate and associate the public key plus its handle with an appropriate user account. This **key handle** will be later used to set the expectations between the client device/app and FSP which key should be used for signing challenge-response requests. While the RTP Network participants will send the key handle in relevant API calls, the RTP Network's Switch itself won't care about its content, it will simply route it to the appropriate party. That said, the Switch might enforce its presence in some API calls based on their semantics.


U2F Attestation Flow
U2F Device
PayApp
Payer FSP
a(app_id), challenge
a; c(challeng+origin+channel_id+etc)
Generate Kpub, KPriv, Handle h
Kpub, h, attestation_cert, sign(a+c+Kpub+h)
c, Kpub, h, attestation_cert, sign
Associate Kpub with Handle h for user
U2F Device
PayApp
Payer FSP

While the key registration flow described above should be performed outside of the actual RTP Network API calls, the challenge-response part of the flow will be integrated within it (see the details at Transaction Request Authorization Models). The diagram below illustrates only conceptually the flow of data during FIDO challenge-response flow:



Challenge-Response Authorization Flow with FIDO U2F

In the traditional FIDO attestation flow, the relying party (either an FSP or the Switch itself) sends a challenge which the client (PISP app) then authenticates by providing crypto attestation by signing that challenge with its private key. Such flow fits nicely into both credit payment requests, originating from the payer, and requests that originate from the payee ("*pull*"). The client-side PISP app and FSP will need to agree on which key should be used for a particular challenge-response flow. The app could send their key handle expectations with the transaction request and/or the FSP key can specify which key is expected to be used to sign the challenge with its authorization request. This flow is conceptually similar to how Mojaloop performs merchant initiated payment flows which use OTP verification. The more detailed overview of the Mojaloop integration is described later in the document, but the short description below depict the relevant authentication and transaction authorization events:

As precondition steps, the payer party **device registration flow** would follow this pattern:

(R1) The PISP's client app initiates device registration with appropriate FSP. The app will use FSP-specific web UI which will drive the registration process.

(R2) The FSP's registration website will first authenticate the party (however they choose to do it), and then

(R3) The FSP's website will ask the authenticated party to pick one account from the list of all accounts that it owns at this particular institution, after which

(R4) The FSP's website will initiate public/private key pair creation via WebAuthentication API. Upon its creation, the FSP will internally keep the record which associate the public part of the created key, identified by its key handle, with the account selected by the party in step (R3). The device on which this registration flow is running on will safely store the created private key in its cryptographic module and identify it by its key handle.

(R5) The PISP client app, will receive the key handle with the selected account identifier and store them locally.

Once the payer party's device is associated with a financial account at the payer FSP, the payer party will be able to initiate transaction requests via its PISP. The **request authorization flow** for credit transaction would follow this pattern:

(T1) The payer party initiates new transaction requests in the PISP app. In addition to all other financial transaction details, the request will contain the registered key handle (created in step R4) which PISP app is expecting to use for authentication for accessing the payer account and for the authorization of the financial transaction request.

(T2) Upon receiving the transaction request, the payer FSP participant will start the quoting process in which the payee FSP generates the ILP packet and the ILP condition, derived from the packet. The received quote information will be also routed to the PISP by the Mojaloop Switch.

(T3) The payer FSP will request authorization of the transaction from the PISP which initiated the transaction request. The PISP participant will then route the received quote info to the party running the PISP app. When the payer party agrees to the presented transaction details, the transaction authorization will be delivered to the payer FSP in the form of FIDO response where:

- **Challenge** is the ILP condition (32-byte binary string) from the payee-generated transaction quote. Such implicit *challenge* happens to be derived from the ILP packet and as such "wraps" all relevant details of the financial transaction. Such a challenge will prevent replay attacks since the challenge is unique to a given transaction. Use of the ILP condition as the challenge will also add to nonrepudiation properties of the authorization message given that the condition is produced by the payee FSP. Therefore, by signing the ILP condition with its private key, the PISP app will perform crypto attestation of all relevant transaction properties which are generated by both the payer and the payee FSPs side.

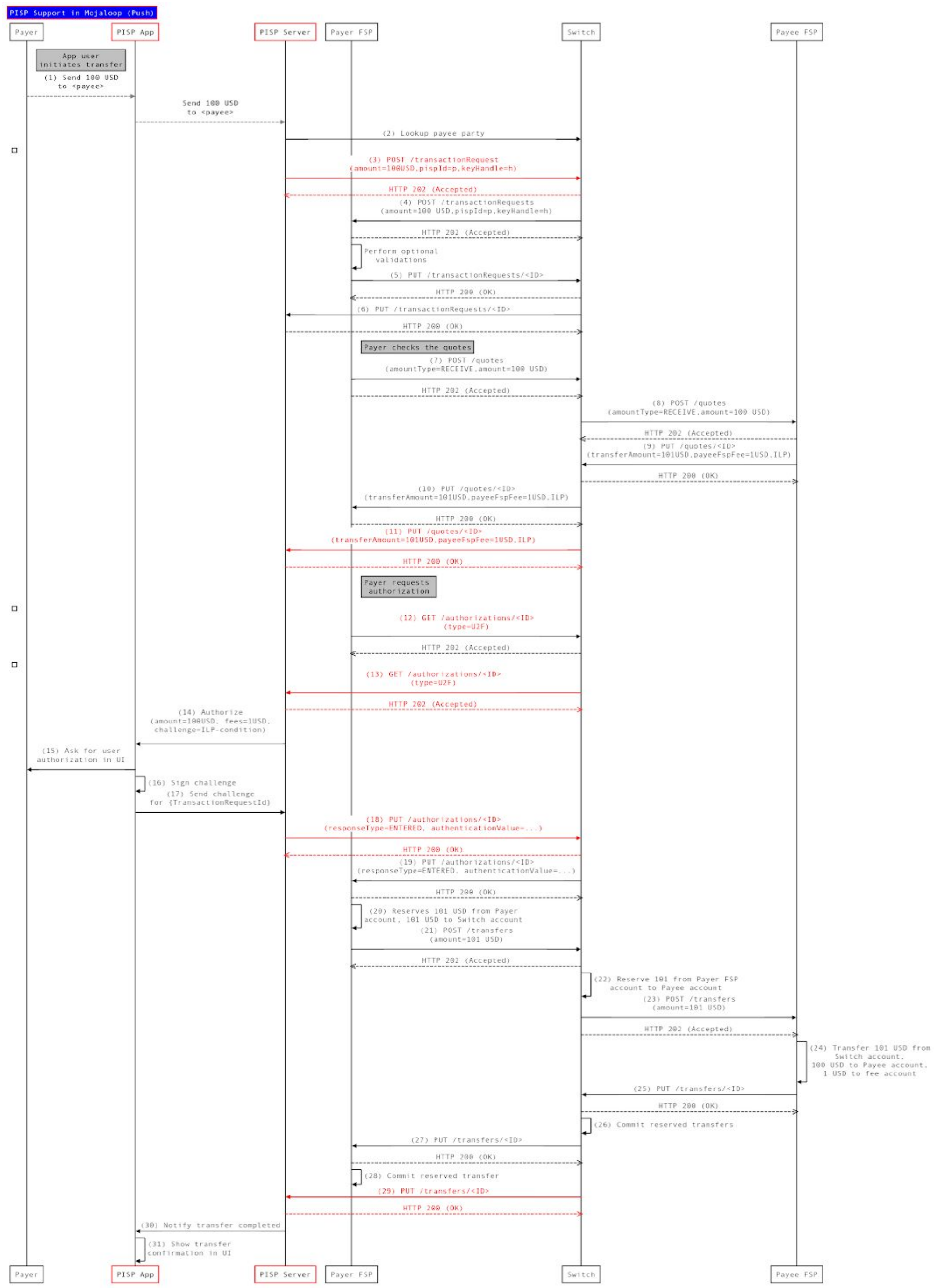
- **Response** will be FIDO-compliant signature of the *challenge* by the private key which is identified by the key handle exchanged between the payer party and its FSP in (T1).

Using PIN as Additional Knowledge Factor

Alternative to using just ILP condition as a crypto nonce, the payer FSP could also require the FIDO signature (response) also to additionally wrap a knowledge factor which the payer party would need to explicitly provide in order to authorize the transaction. The most common form of such **knowledge factor** used in banking would be in a form of **PIN**. With the PIN as a **knowledge factor**, the payer's private key (stored in the phone authenticator) as a **possession factor** and with FIDO's biometric check serving as **inherent factor**, the Mojaloop operator and it participant FSPs could easily implement 3-factor authentication scheme.

PISP-initiated Credit Transaction Requests

The sequence diagram below depicts a simple flow where a payer party initiates financial transfer to the payee via an app like GPay. There might be other flows that we want to cover (i.e. bulk transfer), but they will both follow the key principles of the 3-party participant flow described here. Even though there are no calls added to Mojaloop API, elements marked in red have slightly modified payloads or are routes to the new types of participant (PISPs).


PISP Support in Mojaloop (Push)
Payer
PISP App
PISP Server
Payer FSP
Switch
Payee FSP
App user initiates transfer
(1) Send 100 USD to <payee>
Send 100 USD to <payee>
(2) Lookup payee party
(3) POST /transactionRequest (amount=100USD,pispId=p,keyHandle=h)
HTTP 202 (Accepted)
(4) POST /transactionRequests (amount=100 USD,pispId=p,keyHandle=h)
HTTP 202 (Accepted)
Perform optional validations
(5) PUT /transactionRequests/<ID>
HTTP 200 (OK)
(6) PUT /transactionRequests/<ID>
HTTP 200 (OK)
Payer checks the quotes
(7) POST /quotes (amountType=RECEIVE,amount=100 USD)
HTTP 202 (Accepted)
(8) POST /quotes (amountType=RECEIVE,amount=100 USD)
HTTP 202 (Accepted)
(9) PUT /quotes/<ID> (transferAmount=101USD,payeeFspFee=1USD,ILP)
HTTP 200 (OK)
(10) PUT /quotes/<ID> (transferAmount=101USD,payeeFspFee=1USD,ILP)
HTTP 200 (OK)
(11) PUT /quotes/<ID> (transferAmount=101USD,payeeFspFee=1USD,ILP)
HTTP 200 (OK)
Payer requests authorization
(12) GET /authorizations/<ID> (type=U2F)
HTTP 202 (Accepted)
(13) GET /authorizations/<ID> (type=U2F)
HTTP 202 (Accepted)
(14) Authorize (amount=100USD, fees=1USD, challenge=ILP-condition)
(15) Ask for user authorization in UI
(16) Sign challenge
(17) Send challenge for {TransactionRequestId}
(18) PUT /authorizations/<ID> (responseType=ENTERED, authenticationValue=...)
HTTP 200 (OK)
(19) PUT /authorizations/<ID> (responseType=ENTERED, authenticationValue=...)
HTTP 200 (OK)
(20) Reserves 101 USD from Payer account, 101 USD to Switch account
(21) POST /transfers (amount=101 USD)
HTTP 202 (Accepted)
(22) Reserve 101 from Payer FSP account to Payee account
(23) POST /transfers (amount=101 USD)
HTTP 202 (Accepted)
(24) Transfer 101 USD from Switch account, 100 USD to Payee account, 1 USD to fee account
(25) PUT /transfers/<ID>
HTTP 200 (OK)
(26) Commit reserved transfers
(27) PUT /transfers/<ID>
HTTP 200 (OK)
(28) Commit reserved transfer
(29) PUT /transfers/<ID>
HTTP 200 (OK)
(30) Notify transfer completed
(31) Show transfer confirmation in UI
Payer
PISP App
PISP Server
Payer FSP
Switch
Payee FSP

Credit Transaction Flow Summary

The PISP, as a participant in the system, will be limited in function in the sense that it will be able to call only certain functions on the Switch. The basic entry method for its interaction with other participants is `POST /transactionRequests`. This method was previously reserved for use by the payee side of the transaction flow to initiate the flow but now we are introducing another potential transaction request initiator - the PISP. Once it receives a transaction request call from PISP, the Switch will be able to differentiate this call originating from there from those originating from other 'traditional' FSPs in the system. In addition listing to FSPs and other parameters of the transaction request, such calls will also contain additional information containing the identity of the caller PISP and requested authentication method will be exclusively set to FIDO (new method, see changes listed in `6.4.2.2 POST /transactionRequests`). The Switch will route the transaction request to the payer FSP which will then initiate the regular quoting call sequence. Once the quote info is gathered, the payer FSP will pass this info and ask the Switch to route transaction authentication request to the originating participant, PISP again. The PISP will be responsible to route this info to the calling app instance, which will be responsible to perform FIDO rubber stamping of the authorization request. The Switch will route this authorization info back to the payer FSP which will then initiate the actual transfer to the payee FSP. Once the payee PSP confirms that the transfer is settled on their side, the Switch will route this info both to the payer FSP and the originating PISP. The PISP will pass this info to the originating app instance and show confirmation to the end-user.

Shortly, PISP entities can initiate a new transaction request, intercept and respond to authorization requests from the payer FSP and at the end receive the notifications about the state of the initiated transaction.

Credit Transaction Flow Sequence Diagram Description

The flow described here shows a single financial transfer between the payer and the payee's FSP which is initialized by PISP app running on end-user's device.
Here are the details of the call sequence steps which are depicted in the diagram above:

1. The payer (end-user) initiates financial transfer to the payee in PISP's provided application. The application sends payment instructions to PISP server (PayApp Server in this diagram) which is a registered participant of Mojaloop Switch.

2. PayApp Server performs a payee lookup process with the Switch to determine its FSP. These steps are identical to the existing Mojaloop flow and are not described here.

3. PayApp Server initiates a new financial transaction by calling `POST /transactionRequests`. The request coming from PISPs will contain additional content identifying PISP service as ***initializerId*** and the public-private key pair handle (***payerKeyHandle)*** which app expects to use for authorization of transfer request for this particular FSP financial account. *initializerId* can be used by FSPs to limit the functional scope of what a given key can do. The request payload will also have ***authenticationType*** value set to ***FIDO***.

4. The Switch will verify this request (i.e. enforce values of fields as described in #3) and pass to payer FSP. The request will be internally marked as a special transaction which involves 3 participants - payer FSP, payee FSP and PISP. The switch will need to use this information to properly route upcoming messages.

5. The Payer FSP will verify the request (i.e. check if *appKeyHandle* is registered) and then call `PUT /transactionRequests` on the Switch let it know about the current state of the transaction request.

6. The Switch will then pass transaction state from #5 to PayApp Server which can optionally notify the originating app instance about it (i.e. in case Payer FSP had rejected the transaction request).

7-10. If the transaction request hasn't been rejected by the Payer FSP, it will initiate transaction quoting flow with the Switch and the Payee FSP.

11. The quote information will be routed to the PISP by the Switch given that the transaction request originated from it.

12. Once the Payer FSP receives the transaction quote from the Payee, it will add its own fees on top of that and request transaction authorization by calling `GET /authorizations/<ID>` on the Switch. The Payer FSP will be authorized in the form of FIDO challenge-response from requests that originate from PISPs. Check FIDO flow details. This request will pass the **retriesLeft** field which retries left before the financial transaction is rejected by the payer. The value of <ID> part of `GET /authorizations/<ID>` request path should match value of **transactionRequestID** from `POST /transactionRequests` call from step #3.

13. Given that the Switch "knows" #12 step is an authorization request for a transaction which was initialized by a PISP participant, it will route `GET /authorizations/<ID>` call to PISP's (PayApp Server in the diagram).

14. The PISP Server will notify the originating app instance about the authorization request. It will pass all relevant financial transactions and quote information.

15. The PISP payment app instance will show to the end-user relevant transaction details (i.e. fees) and ask the user for its authorization of the transfer request.

16. The payment app will interact with the client device's service which provides a **challenge** FIDO compliant signing functionality.

17. The payment app instance will pass to PISP Server signed **response**, a crypto signature with which the payer party authorizes the transaction request and related quote details. The response will be signed in the payment app by using the private key stored on the client device. The selected signing will be determined by the ***payerKeyHandle*** value exchanged in step #3.

18. PISP Server will send U2F authorization by calling PUT /authorizations on the Switch. Assuming that authorization was successfully performed by the client, in this reply the PISP Server will then set value **authenticationValue** to base64 encoded U2F *response*, value **authentication** to 'U2F**'** and value of **responseType** to 'ENTERED'.

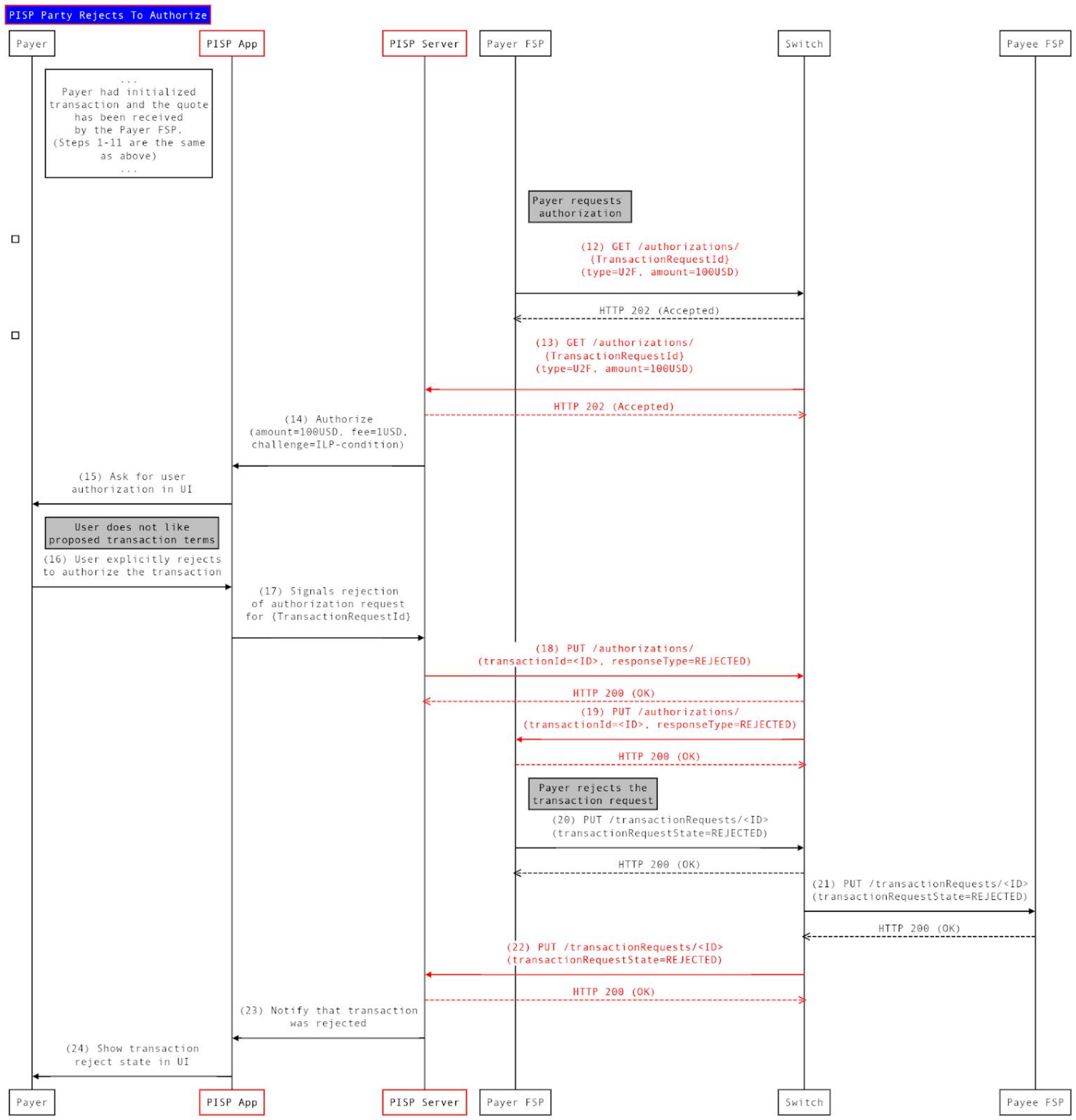
19. The switch will pass authorization info from #18 to the Payer PSF by calling PUT /authorizations on its entry point. If the authorization data passed to the payer FSP by the the party did not match the expected U2F signature, the payer will at this point reinitiate step #12 with **retriesLeft** field values reduced by one. If there are no more retries left, the payer FSP will call `PUT /transactionRequests` on the Switch with **transactionRequestState** set to 'REJECTED'. The Switch will pass rejection info by calling the same method on the PISP endpoint server.

20. - 28. The Payer FSP initiates transfer processing in the way it currently does this with the Switch. The Switch will call Payee FSP to notify it about the transfer. Once Payee FSP confirms that it performed the transfers, the switch will commit reserved resources to appropriate accounts.

29. Once the Switch has notified the Payer FSP about the performed transfer and the settlement on the Switch side, the Switch will also pass the same notification to PISP Server by invoking PUT /transfers/<ID> call on its entry point service.

Rejected Authorization Flow

The following diagram is a variant of flow that starts the same as the end-to-end successful flow described above, but in which the payer party explicitly rejects transaction terms in PISP app UI.


PISP Party Rejects To Authorize
Payer
PISP App
PISP Server
Payer FSP
Switch
Payee FSP
... Payer had initialized transaction and the quote has been received by the Payer FSP. (Steps 1-11 are the same as above) ...
Payer requests authorization
(12) GET /authorizations/ {TransactionRequestId} (type=U2F, amount=100USD)
HTTP 202 (Accepted)
(13) GET /authorizations/ {TransactionRequestId} (type=U2F, amount=100USD)
HTTP 202 (Accepted)
(14) Authorize (amount=100USD, fee=1USD, challenge=ILP-condition)
(15) Ask for user authorization in UI
User does not like proposed transaction terms
(16) User explicitly rejects to authorize the transaction
(17) Signals rejection of authorization request for {TransactionRequestId}
(18) PUT /authorizations/ (transactionId=<ID>, responseType=REJECTED)
HTTP 200 (OK)
(19) PUT /authorizations/ (transactionId=<ID>, responseType=REJECTED)
HTTP 200 (OK)
Payer rejects the transaction request
(20) PUT /transactionRequests/<ID> (transactionRequestState=REJECTED)
HTTP 200 (OK)
(21) PUT /transactionRequests/<ID> (transactionRequestState=REJECTED)
HTTP 200 (OK)
(22) PUT /transactionRequests/<ID> (transactionRequestState=REJECTED)
HTTP 200 (OK)
(23) Notify that transaction was rejected
(24) Show transaction reject state in UI
Payer
PISP App
PISP Server
Payer FSP
Switch
Payee FSP

1. - 11. These steps, not shown in the diagram above, are identical to steps described in the flow described in Credit Transaction Flow Sequence Diagram Description section.

12. The Payer FSP requests authorization by calling `GET /authorizations/<ID>` on the Switch.

13. The Switch “routes” the authorization request to the PISP participant’s endpoint server.

14. The PISP Server will ask the originating PISP app instance for the authorization request. It will pass all relevant financial information to the app.

15. The PISP app instance will show to the end-user relevant transaction details (i.e. fees) and ask the user to authorize the transfer.

16. The user decided that the transaction should not proceed and explicitly rejects the authorization request in the PISP app’s UI.

17. The PISP app passes transaction authorization rejection info to PISP Server.

18. PISP Server signals to the Switch that authorization was rejected by the payee party by calling PUT /authorizations and setting value of **responseType** field to ‘REJECTED’.

19. The Switch will “route” authorization rejection to the Payer FSP by calling PUT /authorizations on its endpoint server.

20. Given that the authorization is rejected, the Payer FSP will signal to the Switch that it rejects the transaction request by calling `PUT /transactionRequests` and passing he value of **transactionRequestState** field set to ‘REJECTED’.

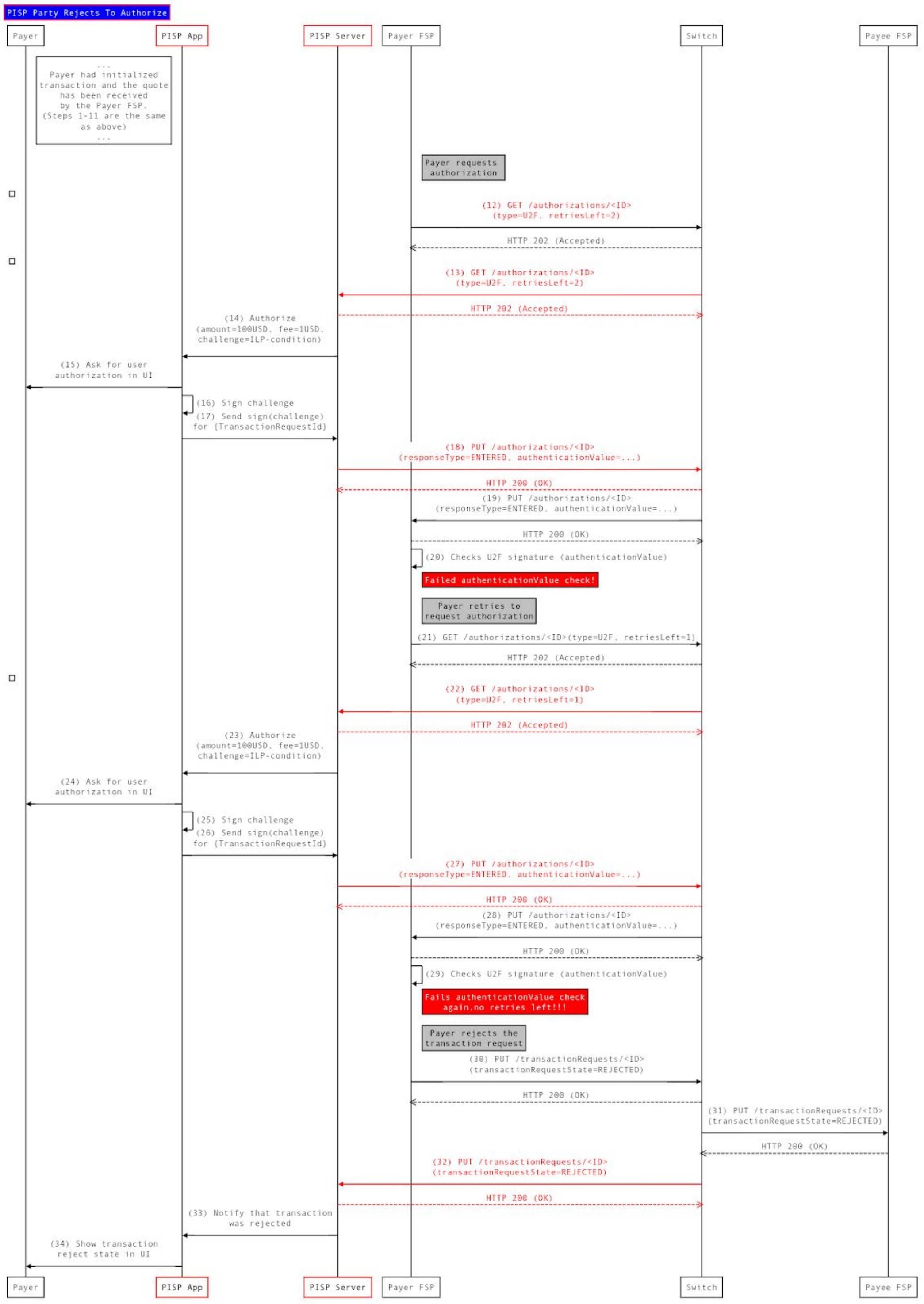
21. The Switch will pass transaction request rejection info to the Payee FSP by calling the same method on its endpoint server, and

22. The Switch will also pass transaction request rejection info to PISP’s server.

23. - 24. The PISP sever can notify the calling PISP app instance that the transaction request is was rejected and no longer considered valid.

Failed Authorization Flow

In this sequence diagram, the payer party (end-user) is attempting to authorize the transaction but the authorization provided by its PISP app instance isn't matching expectations of the Payer FSP checks.


PISP Party Rejects To Authorize
Payer
PISP App
PISP Server
Payer FSP
Switch
Payee FSP
... Payer had initialized transaction and the quote has been received by the Payer FSP. (Steps 1-11 are the same as above) ...
Payer requests authorization
(12) GET /authorizations/<ID> (type=U2F, retriesLeft=2)
HTTP 202 (Accepted)
(13) GET /authorizations/<ID> (type=U2F, retriesLeft=2)
HTTP 202 (Accepted)
(14) Authorize (amount=100USD, fee=1USD, challenge=ILP-condition)
(15) Ask for user authorization in UI
(16) Sign challenge
(17) Send sign(challenge) for {TransactionRequestId}
(18) PUT /authorizations/<ID> (responseType=ENTERED, authenticationValue=...)
HTTP 200 (OK)
(19) PUT /authorizations/<ID> (responseType=ENTERED, authenticationValue=...)
HTTP 200 (OK)
(20) Checks U2F signature (authenticationValue)
Failed authenticationValue check!
Payer retries to request authorization
(21) GET /authorizations/<ID>(type=U2F, retriesLeft=1)
HTTP 202 (Accepted)
(22) GET /authorizations/<ID> (type=U2F, retriesLeft=1)
HTTP 202 (Accepted)
(23) Authorize (amount=100USD, fee=1USD, challenge=ILP-condition)
(24) Ask for user authorization in UI
(25) Sign challenge
(26) Send sign(challenge) for {TransactionRequestId}
(27) PUT /authorizations/<ID> (responseType=ENTERED, authenticationValue=...)
HTTP 200 (OK)
(28) PUT /authorizations/<ID> (responseType=ENTERED, authenticationValue=...)
HTTP 200 (OK)
(29) Checks U2F signature (authenticationValue)
Fails authenticationValue check again,no retries left!!!
Payer rejects the transaction request
(30) PUT /transactionRequests/<ID> (transactionRequestState=REJECTED)
HTTP 200 (OK)
(31) PUT /transactionRequests/<ID> (transactionRequestState=REJECTED)
HTTP 200 (OK)
(32) PUT /transactionRequests/<ID> (transactionRequestState=REJECTED)
HTTP 200 (OK)
(33) Notify that transaction was rejected
(34) Show transaction reject state in UI
Payer
PISP App
PISP Server
Payer FSP
Switch
Payee FSP

1. - 11. These steps, not shown in the diagram above, are identical to steps described in the flow described in Credit Transaction Flow Sequence Diagram Description section.

12. The Payer FSP requests authorization by calling `GET /authorizations/<ID>` on the Switch. In this example, this request will pass the **retriesLeft** field with value *2.* This will communicate the PISP server and the PISP app how many times the user can try to authorize the transaction before the transaction request fails.

13. The Switch "routes" the authorization request to the PISP participant's endpoint server.

14. The PISP Server will ask the originating PISP app instance for the authorization request. It will pass all relevant financial information to the app.

15. The PISP app instance will show to the end-user relevant transaction details (i.e. fees) and ask the user to authorize the transfer.

16. The user approves the transaction and its PISP app generates U2F signature to authorize it.

17. The PISP app passes transaction authorization signature to PISP Server.

18. The PISP Server will then send U2F authorization details to the Switch by calling PUT /authorizations. In this call, the PISP Server will then set value **authenticationValue** to base64 encoded U2F *response*, value **authentication** to 'U2F**'** and value of **responseType** to 'ENTERED'.

19. The Switch will then "route" these authorization details to the Payer FSP by calling PUT /authorizations on its endpoint server.

20. The Payer FSP checks the received **authenticationValue** but its value does not meet its expectation.

21. - 28. Given that the current transaction has 2 retries left, the Payer FSP will re-requests authorization by calling `GET /authorizations/<ID>` this time with **retriesLeft** field set to value *1.* The rest of the steps 22. - 28. are identical to steps are identical to steps 12-19 - the request authorization is routed to the app, the app sends U2F signature as response back to the Payer FSP.

29. The Payer FSP checks the received **authenticationValue** and it again does not meet its expectation. Given that this was the last authentication attempt retry, it decides to reject the transaction request.

30. Given that the authorization is rejected by the Payer FSP, this participant will signal that to the Switch by calling `PUT /transactionRequests` and passing the value of **transactionRequestState** field set to ‘REJECTED’.

31. The Switch will pass transaction request rejection info to the Payee FSP by calling the same method on its endpoint server, and

32. The Switch will also pass transaction request rejection info to PISP’s server.

33. - 34. The PISP sever can notify the calling PISP app instance that the transaction request is was rejected and no longer considered valid.

Mojaloop API Change Details

Note: This section is neither final nor comprehensive design doc, but it does illustrate the point that there isn't all that much to change in order to support PISPs and all scenarios brought up this doc.

This section of the document should be read as diff of Mojaloop API. It's a draft of exact changes that need to be done to the switch specification. This proposal adds new values/fields to existing calls and data typed (marked in **bold** below), but there are no new API calls added to Mojaloop on top of that.

"FSPIOP-Originator" HTTP Request Header Field

Mojaloop already defines header fields for source and destination participants as `FSPIOP-Source` and `FSPIOP-Destination`. To clearly identify requests originating from PISP participants and enable their proper verifications and routing, we will add new `FSPIOP-Originator` http header field.

6.4.2.2 POST /transactionRequests

Name	Cardinality	Type	Description
transactionRequestId	1	CorrelationId (UUID)	Common ID between the FSPs for the transaction request object, decided by the Payee FSP. The ID should be reused for resends of the same transaction request. A new ID should be generated for each new transaction request.
payee	1	Party	Information about the Payee in the proposed financial transaction.
payer	1	PartyIdInfo	Information about the Payer type, id, sub-type/id, FSP Id in the proposed financial transaction.
initiatorId	**0..1**	**Party**	**Information identifying PISP in the proposed financial transaction.**
payerKeyHandle	**0..1**	**KeyHandle**	**Information identifying registered U2F key with payer FSP.**
amount	1	Money	Requested amount to be transferred from the Payer to Payee.

transactionType	1	TransactionType	Type of transaction.
note	0..1	Note	Reason for the transaction request, intended to the Payer.
geoCode	0..1	GeoCode	Longitude and Latitude of the initiating Party. Can be used to detect fraud.
authenticationType	0..1	AuthenticationType	OTP, QRCODE **or U2F**
expiration	0..1	DateTime	Can be set to get a quick failure in case the peer FSP takes too long to respond. Also, it may be beneficial for Consumer, Agent, Merchant to know that their request has a time limit.
extensionList			Optional extension, specific to deployment.

6.6.2.1 GET /authorizations/<ID>

The <ID> in the URI should contain the transactionRequestID (see Table 14), received from the POST /transactionRequests. This request requires a query string (see Section 3.1.3 for more information regarding URI syntax) to be included in the URI, with the following key-value pairs:

Key-value pair	Description
authenticationType=<Type>	<Type> value is a valid authentication type from the enumeration. **When requesting authorization for transaction requests originating from PISPs, the value of this query parameter will be set to U2F.**
retriesLeft=<NrOfRetries>	<NrOfRetries> is the number of retries left before the financial transaction is rejected. <NrOfRetries> must be expressed in the form of the data type Integer (see Section 7.2.5). retriesLeft=1
amount=<Amount>	<Amount> is the transaction amount that will be withdrawn from the Payer’s account.
currency=<Currency>	<Currency> is the transaction currency for the amount that will be withdrawn from the

	Payer’s account. The <Currency> value must be expressed in the form of the enumeration CurrencyCode (see Section 7.5.5).

(bold content is added/modified)

6.6.3.1 PUT /authorizations/<ID>

Name	Cardinality	Type	Description
authenticationInfo	**0..1**	**AuthenticationInfo**	**OTP, QRCODE or U2F.**
responseType	1	AuthorizationResponse	Enum containing response information; if the customer entered the authentication value, rejected the transaction, or requested a resend of the authentication value.

(bold content is added/modified)

7.3.3 AuthenticationValue

Table 36 contains the data model for the element AuthenticationValue.

Name	Cardinality	Type	Description
AuthenticationValue	**1**	**Depending on AuthenticationType, if:** *** OTP - OtpValue** *** QRCODE - String(1..64)** *** U2F: String(1..64), base64 encoded signed challenge.**	**Contains the authentication value. The format depends on the authentication type used in the AuthenticationInfo complex type.**
Counter	**0..1**	**Int64**	**Sequential counter used for cloning detection. Present only for U2F authentication.**

(bold content is added/modified)

7.4.1 AuthenticationInfo

Table 69 contains the data model for the complex type AuthenticationInfo.

Name	Cardinality	Type	Description
authentication	**1**	AuthenticationType	Type of authentication
authenticationValue	**1**	AuthenticationValue	Authentication value

7.5.2 AuthenticationType

Table 88 contains the allowed values for the enumeration AuthenticationType:

Name	Description
OTP	One-time password generated by the Payer FSP.
QRCODE	QR code used as One Time Password.
U2F	**U2F challenge-response, where payer FSP verifies if the response provided by end-user device matches the previously registered key.**

(bold content is added/modified)