Resource Oriented Micropayment Pool

We have designed and implemented an off-chain “Resource Oriented Micropayment Pool” that is purpose-built

For video streaming. It allows a user to create an off-chain micropayment pool that any other user can withdraw from using off-chain transactions, and is double-spend resistant. It is much more flexible compared to off-chain payment channels. In particular, for the video streaming use case, it allows a viewer to pay for video content pulled from multiple caching nodes without on-chain transactions. By replacing on-chain transactions with off-chain payments, the built-in “Resource Oriented Micropayment Pool” significantly improves the scalability of the blockchain.

The following scenario and diagram provide a comprehensive walkthrough of how the Resource Oriented Micropayment Pool works in application.

**● Step 0.** A provider of a resource makes available the *resourceId* to identify the resource and the cost of the resource available to be purchased by potential receivers and potentially how long they are willing to wait to redeem payment.

* Is this advertising left entirely to the parties involved? What about non-video content case?
* How is the block time calculated so that the expiration of the pool can be understood by the parties? Is/could this be made available by the explorer API?

**● Step 1.** Micropayment pool creation: As the first step, Alice publishes an on-chain transaction to create a micropayment pool with a time-lock and a slashable collateral.

*CreatePool(resourceId, deposit, collateral, duration)*

A couple things to be noted. To create the pool, Alice needs to specify the “Resource ID” *resourceId* that uniquely represents the digital content she intends to retrieve. It may refer to a video file, or a live stream.

* To confirm. Is *resourceID* the entire video file and does resourceID+reserve\_seq uniquely identify the video segment? If so, if Alice requests *resourceID(0)* from Bob but fails to receive it, does she wait or request *resourceID(0)* from Carol?

The *deposit* amount needs to be at least the total value of the resource to be retrieved. For instance, if the resource is a video file which is worth 10 tokens, then the deposit has to be at least 10 tokens.

The *collateral* is required to discourage Alice from double spending. If a double spending attempt from Alice is detected by the validators of the blockchain, the collateral will be slashed. Later in the blogpost we will show that if *collateral > deposit*, the net return of a double spend is always negative, and hence any rational user will have no incentive to double spend.

The duration is a time-lock similar to that of a standard payment channel. Any withdrawal from the payment pool has to be before the time-lock expires.

* Is it possible for Alice to release the Funding Pool prior to the time-lock expiration? Even if the API doesn’t implement it, it’s interface is documented(Auto Release)?

The blockchain returns Alice the Merkle proof of the *CreatePool( )* transaction after it has been committed to the blockchain, as well as *createPoolTxHash*, the transaction hash of the *CreatePool()* transaction.

* How is the Merkle proof attained by Alice in response to Pool creation? I don’t see a mechanism for this(missing something)?

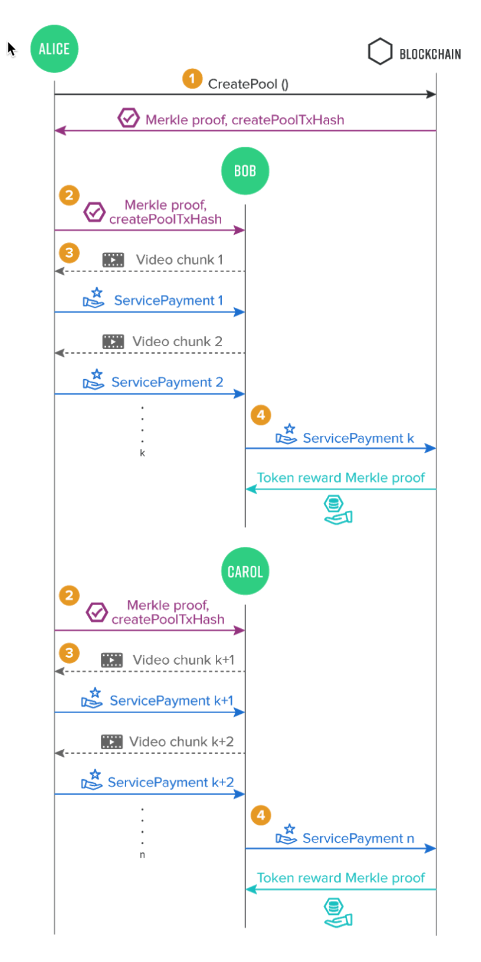


Figure4. *Resource Oriented Micropayment Pool* shows viewer Alice making off-chain transactions to cachers Bob and Carol for video chunks

● **Step 2. Initial handshake between peers:** Whenever Alice wants to retrieve the specified resource from a peer (Bob, Carol, or David, etc.). She sends the Merkle proof of the on-chain *CreatePool( )* transaction to that peer. The recipient peer verifies the Merkle proof to ensure that the pool has sufficient deposit and collateral for the requested resource, and both parties can proceed to the next steps.

● **Step 3. Off-chain micropayments:** Alice signs S*ervicePayment* transactions and sends them to the peers

off-chain in exchange for parts of the specified resource (e.g. a piece of the video file, a live stream segment,

etc.). The *ServicePayment* transaction contains the following data:

*targetAddress,* // Bob or Carol

*transferAmount*, // Agreed upon prior to transactions

*createPoolTxHash,* // Interface doesn’t seem to require passing this. Mis-documented?

*targetSettlementSequence,* // Same as payment\_sequence?

*Sign( SKA, targetAddress|| transferAmount|| createPoolTxHash|| targetSettlementSequence)*

The *targetAddress* is the address of the peer that Alice retrieves the resource from, and the *transferAmount* is

the amount of token payment Alice intends to send(from the ReservePool). The *targetSettlementSequence* is to prevent a replay attack. It is similar to the “nonce” parameter in an Ethereum transaction. If a target(Bob?) publishes a *ServicePayment* transaction to the blockchain (see the next step), its targetSettlementSequence needs to increment by one.

* Is this incrementation tracked by Alice? The blockchain? Does Alice have to make sure that *targetSettlementSequence(s)* are uniquely incrementing for Bob+1, Bob+2, Carol+101, Carol+102?

The recipient (of the Merkle proof) peer (Bob or Carol) needs to verify the off-chain transactions and the signatures(how? Hash?). Upon validation, the peer(Bob or Carol) can send Alice the resource specified by the CreatePool() transaction.

* This is confusing from Figure4. If time is progressing going down, it appears that the resource chunk is sent before the service payment for it. Is this the case? The description suggests the opposite.

Also, we note that the off-chain ServicePayment transactions are sent directly between two peers. Hence there is no scalability bottleneck for this step.

**● Step 4. On-chain settlement:** Any peer (i.e. Bob, Carol, or David, etc) that received the ServicePayment transactions from Alice can publish the signed transactions to the blockchain anytime before the time-lock expires to withdraw the tokens. We call the *ServicePayment* transactions that are published the “on-chain settlement” transactions.

* Is there a way to “batch” a bunch of ServicePayments that have been received by Bob/Carol into a single on-chain ServicePayment(aggregate payment) to minimize the number of on-chain transactions(and fees)?

Note that the recipient peers needs to pay for the gas fee for the on-chain settlement transaction. To pay less transaction fees, they would have the incentive to publish on-chain settlements only when necessary, which is beneficial to the scalability of the network.

We note that no on-chain transaction is needed when Alice switches from one peer to another to retrieve the

resource. In the video streaming context, this means the viewer can switch to any caching node at any time without making an on-chain transaction that could potentially block or delay the video stream delivery. As shown in the figure, in the event that Bob leaves, Alice can switch to Carol after receiving k chunks from Bob, and keep receiving video segments without an on-chain transaction.

Moreover, the total amount of tokens needed to create the micropayment pool is (collateral+ deposit) , which can be as low as twice of the value of the requested resource, no matter how many peers Alice retrieves the resource from. Using computational complexity language, the amount of reserved token reduces from O( n) to O( 1) compared to the unidirectional payment channel approach, where n is the number of peers Alice retrieves the resource from.

Double Spending Detection and Penalty Analysis

To prevent Alice, the creator of the micropayment pool from double spending, we need to 1) be able to detect double spending, and 2) ensure that the net value Alice gains from double spending is strictly negative.

Detecting double spending is relatively straightforward. The validators of the Theta Network check every on-chain transaction. If a remaining deposit in the micropayment pool cannot cover the next consolidated payment transaction signed by both Alice and another peer, the validators will consider that Alice has conducted a double spend.

Next, we need to make Alice worse off if she double spends. This is where the collateral comes in. Earlier, we mentioned that the amount of collateral tokens has to be larger than the deposit. And here is why. In Figure 5 below, Bob, Carol, and David are honest. Alice is malicious. Even worse, she colludes with another malicious peer Edward. Alice exchanges partially signed(intentionally mis-signed?) transactions with Bob, Carol, and David for the specified 16 resource.

Since Alice gains no extra value for the duplicated resource, the maximum value she gets from Bob, Carol, and David is at most the deposit amount. As Alice colludes with Edward, she can send Edward the full(properly signed) deposit amount.

She then asks Edward to commit the settlement transaction before anyone else and return her the deposit later. In other words, Alice gets the resource which is worth at most the deposit amount for free, before the double spending is detected. Later when Bob, Carol, or David commit the settlement transaction, the double spending is detected, and the full collateral amount will be slashed. Hence, the net return for Alice is netAliceΩdepositcollateral Therefore, we can conclude that for this scenario, as long as collateral > deposit, Alice’s net return is negative. Hence, if Alice is rational, she would not have any incentive to double spend.

We can conduct similar analysis for other cases. The details are omitted here, but it can be shown that in all cases

Alice’s net return is always negative if she conducts a double spend.

Another case is that Alice is honest, but some of her peers are malicious. After Alice sends a micropayment to one of those peers, it might not return Alice the resource she wants. In this case, Alice can turn to another peer to get the resource. Since each incremental micropayment can be infinitesimally small in theory, Alice’s loss can be made

arbitrarily small. 17 Figure5. Malicious Actor Detection and Penalty shows malicious actor Alice attempting to make a double spend and the resulting penalty she receives 18