It is built upon PoRID protocol and is in the same security model as RC-S-P is. RC-PoR-P inherits the features of PoRID and RC-PoR-P; however, unlike RC-PoR-P, it does not use any zero-knowledge proofs and there is no third-party arbiter involved. Even though RC-PoR-P has some overlaps with RC-S-P, it has many differences too. Therefore, we provide the protocol’s overview and its detailed description below.

At a high level the protocol works as follows. The client and server use SAP to provably agree on two private statements, one statement includes payment details, and another one specifies a secret key, k, and a pad’s length. They also agree on public parameters such as (a) the private time bubble’s length, that is the total number of billing cycles: z, plus a waiting period, H, and (b) a smart contract that specifies z and the total amount of masked coins each party should deposit. They deploy the contract. Each party deposits its masked coins in the contract within a fixed time. If any party does not deposit enough coins on time, then the parties have a chance to withdraw their coins and terminate the contract after a certain time. To start using/providing the service, the client invokes PoRID.setup() to encode the file and generate a metadata and set of public parameters. It sends an encryption of the metadata and public parameters to the smart contract and sends the encoded file to the server who decrypts them and using the encoded file checks their correctness by calling PoRID.server(). If the server decides not to serve, it sends to the contract 0 within a fixed time; in this case, the parties can withdraw their deposit and terminate the contract. Otherwise, the server sends 1 to the contract.

At the end of each billing cycle, the client generates an encrypted query, by calling PoRID.genQuery() and encrypting its output using the key, k. It sends the result to the contract. In the same cycle, the server retrieves the query, and decrypts it. Then, it locally checks its correctness, by calling PoRID.checkQuery(). If the query is rejected, the server locally stores the index of that billing cycle. Then, it generates a dummy PoR proofs, pads them and sends the result to the contract. However, if the server accepts the query, it generates PoR proofs by calling PoRID.prove(). Then, it encrypts the proofs, pads them and sends the result to the contract. After that, the client removes the pads, decrypts the proofs and locally verifies them, by calling PoRID.verify(). If the verification is passed, then the client knows the file is retrievable with a high probability. But, if the proof is rejected, then it locally stores the index of that billing cycle and waits until the private time bubble passes and dispute resolution time arrives.

During the dispute resolution period, in case the server rejects the query or the client rejects the PoR proofs, that party sends to the contract (a) the indices of the billing cycles in which its counter party provided invalid values, and (b) the statement that contains the decryption key and padding detail. The contract checks the validity of the statement first. If it accepts, then it removes the pads and decrypts the values. Then, it checks the party’s claim by calling PoRID.checkQuery () and PoRID.identify () if the server or client calls the contract respectively. The contract also keeps track of the number of times each party provided invalid query or PoR proof.

In the next phase, to distribute the coins, either client or server sends to the contract: (a) “pay” message, (b) the agreed statement that specifies the payment details, and (c) the statement’s proof. The contract verifies the statement and if approved it distributes the coins according to the statement’s detail, and the number of times each party misbehaved.