New contingent service payment protocol:

1. Informally the fairness is satisfied because if the client does not deposit anything, then the server provides no proof. Also, if the client claims a false proof, the claimed is checked by an honest arbiter, so if the client is detected and its coins is transfer to the server. On the other hand, if the server provides no proof, then the client gets its deposit and the server’s deposit back after a certain time. Also, if the server provides an invalid proof, the client can detect it and raises a dispute. In this case, the arbiter is invoked who checks the proof correctness and sends the deposits to the client.

In our protocol, if the parties behave rationally, then there will be no need to invoke the arbiter.

------------- **FairSwap** -------------------

FairSwap: it has been designed to support efficiently exchange of digital goods with coins in a fair manner. It is not suitable for services (see below).

In the protocol, similar to zkCSP, the server first sends proofs to the client who checks the proofs and then pays deposits. However, if this is used directly for exchange of services for coins, then (a) as soon as the client gets the proof it is convinced that the server knows the proof and it does not pay deposit. Also, like in zkCSP, the client can avoid participating in the fair exchange, while it has been using the service. Therefore, it can waste the server resources. Also, similar to zkCSP, it leaks information (e.g. file size, deposit amount, proof status etc)

Let’s assume the protocol is modified such that the client needs to deposit its coins before it start using the service and it must provide an input to the contract when the server provides encrypted values to it. In this case, the client can still cheat, by claiming the encrypted values or the leave nodes do not match the Merkle tree root the server stored in the contract. To resolve this issue, we have to modify the protocol again, so the server also sends the signature of the encrypted values to the client. In this case, when the client wants to claim the server misbehaves, it also has to send all the encrypted value and its signature to the contract, which first check all signatures and then reconstructs the tree on top of the encrypted values. However, this significantly increases the client-side communication (at least by a factor of two) and contract side computation by at least n public key operations.

Note the protocol explicitly allows parties to abort without loosing their deposit, before the key is revealed to the contract, i.e. at any point in the first phase/round where the sender/server sends the encrypted values to the client who verifiers them (see page 9 left-side column).

**To summarise, the protocol does not directly support fair exchange of services.**

Ignore:

This problem would not be addressed even if we require the client to deposit its coins when it starts using the service. Because of the problem (a) above, i.e. after it is convinced by the server knows the proof, it simply ignores it, and do not send anything to the contract, therefore the server is not paid. Also, the protocol explicitly allows parties to abort without loosing their deposit, before the key is revealed to the contract, i.e. at any point in the first phase/round where the sender/server sends the encrypted values to the client who verifiers them (see page 9 left-side column). Moreover, one may require the client to deposit first (when it start using the service) and force the client to provide an input (in round 2) where the client activates the contract (after it checks the merkle tree proof sent by the server). However, the client can claim that the server proofs are invalid. Recall that the proofs contains n+m encrypted values such that the root of a Merkle tree constructed on these values are stored by the server on the contract. In other words, the client claims the root and leaves do not match, so it has provided an input. The Fairswap protocol provides no smart contract-based verification mechanism in this case. A simple solution is that the server first signes the encrypted values (the leaves) and sends them to the client. Now for the client sends all encrypted values to the server who again build a tree on top of them and compares the resulting root with what the sender sent to the contract. However, this is very inefficient, as the client has to send message whose size is even bigger than the original file size to the contract, which ultimately defeats the purpose of the entire protocol.

OptiSwap:

This protocol is not suitable for a service as well.

Note this protocol similar to Fairswap does not preserve the privacy. Similar to Fairswap, the client can avoid sending deposit to the contract after it receives the encrypted values from the seller in 2nd round. Moreover, similar to Fairswap, it would not suffice to just plug a verifiable service secure only against the server to OptiSwap, as the same issues would occur if it is done.