SwapBill Documentation

Release 0.5

Thomas Young

September 11, 2014

Contents

https://github.com/crispweed/swapbill

Explanation:

CHAPTER 1

Introduction

1.1 Embedded protocol

SwapBill is an 'embedded' cryptocurrency protocol, which means that SwapBill transactions are hosted on an existing blockchain.

The SwapBill protocol defines a subset of host transactions which are also considered valid SwapBill transactions, with additional control information encoded into these special transactions, and with protocol rules for how these transactions then act on a global SwapBill state.

1.2 Security

The main advantage of this setup is that we benefit from the security guarantees of an existing, established blockchain, including proof of work, whilst also having the flexibility to add new transaction types and features that would be difficult to add directly to an existing blockchain.

Notably, this setup eliminates the need to obtain any kind of 'critical mass' of miners during a product launch phase, and makes it possible for SwapBill to be launched and operate robustly without any minimum required uptake.

1.3 Purpose

The primary purpose for SwapBill is to introduce trustless exchange features such as 'atomic' exchange between coins hosted on different blockchains.

1.4 Status

A reference client for the SwapBill protocol is provided, written in Python and using the bitcoin and/or litecoin reference clients as as backends for peer to peer network connectivity and block validation.

The SwapBill client is currently at the preview stage, and for this reason only operates on testnet versions of host blockchains.

In the current release (0.5), the supported host blockchains are bitcoin testnet and litecoin testnet.

As described here, testnet coins are designed to be without value, and the same goes for swapbill generated with the current preview client, so you can try things out at this stage without risking any real value.

1.5 Multiple denominations

A different swapbill denomination is created for each host blockchain on which swapbill is embedded. With the current reference client it is possible to create either ‘bitcoin testnet swapbill’ or ‘litecoin testnet swapbill’.

1.6 Protocol subject to change

This preview client release is provided for community feedback about a protocol in development. The protocol and client interface implemented for the current release, and as described in this document, are not final, and are subject to change.

CHAPTER 2

Requirements

To run the SwapBill reference client you'll need:

- Python version 2.7, 3.2, 3.3 or 3.4
- The third party Python 'ecdsa' and 'requests' modules
- For operations involving 'litecoin testnet swapbill': the litecoin reference client set up and running as an RPC server
- For operations involving 'bitcoin testnet swapbill': the bitcoin reference client set up and running as an RPC server

So you'll need *either* a litecoin *or* bitcoin RPC server set up, but can then work with the corresponding swapbill denomination for whichever server you have set up.

If you have both RPC servers set up, you can also try out the functionality for cross chain exchanges between these two denominations.

The code has been tested on Linux and Windows, but should work on any platform with support for the litecoin reference client and the required Python dependencies.

2.1 Header only clients

While the current version of the SwapBill preview client requires a full litecoind or bitcoind node as a backend, the protocol is designed *not* to require a full blockchain scan, and so a 'header only' client is also possible, and something that will likely be added in the future.

CHAPTER 3

Terminology

For simplicity, throughout this documentation, we may use:

- 'the client' to refer to the SwapBill reference client
- 'bitcoind' to refer to the bitcoin reference client, but also by implication other potential host blockchain reference clients (such as litecoind)
- 'swapbill' to refer to amounts of swapbill (without necessarily also specifying which host this is on)
- 'SwapBill' (capitalised) to refer to the SwapBill protocol, or to the embodiment of this protocol in the SwapBill reference client

For currency amounts we'll commonly refer to 'swapbill' and 'host coin'.

CHAPTER 4

Setting up the host RPC server

The (SwapBill) client currently requires a 'full node' to be set up on the host blockchain, and running as an RPC server. (See *Requirements*.) The client will then call through to this RPC server for blockchain updates, and for signing and sending swapbill transaction.

4.1 Downloadable installers

For bitcoin, you can install 'Bitcoin Core', from here. For litecoin, you can install 'Litecoin-QT', from here.

4.2 Building from source

Or you can build from source, from https://github.com/bitcoin/bitcoin or https://github.com/litecoin-project/litecoin.

When building from source, note that gui support is not actually required. If you just build either bitcoind or litecoind, and not the QT versions, you can avoid bringing in QT dependencies.

4.3 Configuration

You'll need to configure bitcoind to start as a server, and to connect to testnet.

The default location for the bitcoind configuration file is `~/.bitcoin/bitcoin.conf` on Linux, and something like `C:\Users\YourUserName\AppData\Roaming\BitCoin\bitcoin.conf` on Windows.

(For litecoind the locations are similar, but with 'litecoin' in place of 'bitcoin'.)

Create a file in this default location (if not already present), and add lines like the following:

```
server=1
testnet=1
rpcuser=rpcuser
rpcpassword=somesecretpassword
```

(Change the password!)

To start the server you can then either launch bitcoinQT (the graphical client) normally, or run bitcoind from the command line.

A good setup for Linux can be to tell bitcoind to run in the background (e.g. by adding daemon=1 to the conf file), and use 'tail -f' to watch the end of the generated log file ('tail -f ~/.bitcoin/testnet3/debug.log').

You can test the RPC server by making RPC queries from the command line, e.g.:

```
~/git/bitcoin/src $ ./bitcoin-cli getbalance
11914.15504872
```

or:

```
~/git $ litecoin/src/litecoind getblockcount
381925
```

(This RPC interface is very handy for interaction with the reference client generally, and for general troubleshooting.)

4.4 Bootstrapping

When you first start a full node, it can be worthwhile downloading a 'bootstrap' file to speed up initial synchronisation, but we'll just be connecting to the testnet, and bootstrapping is not really necessary for this. Just go ahead and start your node and let it synch to the testnet blockchain!

4.5 Obtaining host currency

You'll need some host currency to work with.

With testnet coin, it's possible to obtain some coin quite quickly, through testnet 'faucets', such as this one (for bitcoin testnet), or here for litecoin testnet.

In the case of the litecoin testnet it's also fairly easy to get testnet coin directly by through the (CPU) mining functionality in litecoind. Use the `'setgenerate true'` RPC command to turn this on. (It seems a lot harder to do this with bitcoin testnet, though.)

4.6 SwapBill client configuration

The SwapBill client looks for your bitcoin or litecoin config file in the default locations, and reads your rpc username and password from there, so no additial configuration should be required before running the SwapBill client.

CHAPTER 5

Running the client

5.1 Obtaining the source code

The project is hosted on https://github.com/crispweed/swapbill, and so you can get the client source code with git, as follows:

```
~/git $ git clone https://github.com/crispweed/swapbill
Cloning into 'swapbill'...
remote: Counting objects: 52, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 52 (delta 9), reused 46 (delta 3)
Unpacking objects: 100% (52/52), done.
```

In case you don't have git, you can also download the source code directly as an archive from https://github.com/crispweed/swapbill/archive/master.zip, extract to a new directory.

5.2 No installer

There's no installation process for the client and you can just run it directly from the downloaded source tree.

You'll need to ensure that the third party python library dependencies are met before running the client, (see *Requirements*) or you'll get an error message telling you to do this.

And then you run the client with (e.g.):

```
~/git $ cd swapbill/
~/git/swapbill $ python Client.py get_balance
```

5.3 Selecting host blockchain

You can use the '–host' command line option to choose the desired host blockchain to run against. This defaults to 'bitcoin', but if you want to run against litecoind (for example if you only have litecoind installed, and not bitcoind) then you need to change client invocation as follows:

~/git/swapbill $ python Client.py –host litecoin get_balance

You can change this selection per client invocation, and work with multiple host blockchains without any problems. (This is required for the cross chain exchange functionality!)

The client maintains independent subdirectories within its data directory for each host, with separate wallet files and state cache data.

5.4 RPC errors

If you don't have bitcoind running, or if you don't have the RPC interface set up correctly, you'll see something like:

```
~/git/swapbill $ python Client.py get_balance
Couldn't connect for remote procedure call, will sleep for ten seconds and then try again.
Couldn't connect for remote procedure call, will sleep for ten seconds and then try again.
(...repeated indefinitely)
```

But if you start the RPC server, the client should connect and complete the command from there.

If the RPC interface is working correctly you should see something like this:

```
~/git/swapbill $ python Client.py get_balance
Failed to load from cache, full index generation required (no cache file found)
State update starting from block 305846
Committed state updated to start of block 305886
In memory state updated to end of block 305906
Operation successful
balance : 0
```

5.5 Command line help

You can get some help about the full set of command line arguments for the client with '-h' or '–help':

```
~/git/swapbill $ python Client.py -h
usage: SwapBillClient [-h] [--dataDir DATADIR] [--host {bitcoin,litecoin}]
                      {force_rescan,burn,pay,counter_pay,buy_offer,sell_offer,complete_sell,reveal_se
                      ...

the reference implementation of the SwapBill protocol

positional arguments:
  {force_rescan,burn,pay,counter_pay,buy_offer,sell_offer,complete_sell,reveal_secret_for_pending_pay
                        the action to be taken
    force_rescan        delete cached state, forcing a full rescan on the next
                        query invocation
    burn                destroy host coin to create swapbill
    pay                 make a swapbill payment
    counter_pay         make a swapbill payment that depends on the same
                        secret as another payment
    buy_offer           make an offer to buy host coin with swapbill
    sell_offer          make an offer to sell host coin for swapbill
    complete_sell       complete a exchange with host coin by fulfilling a
                        pending exchange payment
    reveal_secret_for_pending_payment
                        provide the secret public key required for a pending
                        payment to go through
    back_sells          commit swapbill to back exchanges with host coin
    make_seed_output    make a transaction for use as a seed output
    get_receive_address
                        generate a new key pair for the swapbill wallet and
```

```
                        display the corresponding public payment address
    get_balance         get current SwapBill balance
    get_buy_offers      get list of currently active host coin buy offers
    get_sell_offers     get list of currently active host coin sell offers
    get_pending_exchanges
                        get current SwapBill pending exchange payments
    get_sell_backers    get information about funds currently commited to
                        backing host coin sell transactions
    get_pending_payments
                        get information payments currently pending proof of
                        receipt
    get_state_info      get some general state information

optional arguments:
  -h, --help            show this help message and exit
  --dataDir DATADIR     the location of the data directory
  --host {bitcoin,litecoin}
                        host blockchain, can currently be either 'litecoin' or
                        'bitcoin'
```

And then, you can get help about individual commands by passing '-h' (or '–help') right after the command:

```
~/git/swapbill $ python Client.py burn -h
usage: SwapBillClient burn [-h] --amount AMOUNT

optional arguments:
  -h, --help       show this help message and exit
  --amount AMOUNT  amount of host coin to be destroyed, as a decimal fraction
                   (one satoshi is 0.00000001)
```

5.6 Worked examples

The best way to understand what the main commands do is to go through the various examples provided later on in this documentation.

CHAPTER 6

Wallet organisation

6.1 SwapBill and host wallets

When you run the client there are two different wallet locations to be aware of:

- the 'standard' bitcoind wallet built in to bitcoind, and
- a separate, independant wallet only by the SwapBill client.

Through the RPC interface SwapBill effectively has access to your bitcoind wallet, and uses this for this to pay for:

1. transaction fees
2. dust output amounts
3. 'burn' transactions
4. host currency payments as part of an exchange between swapbill and host currency

The first two items listed here are essentially 'backing' funds and should only ever consume very small amounts of host currency.

The last two items can potentially be significant amounts, but SwapBill will only make these kinds of payments as part of quite specific actions.

6.2 SwapBill wallets

SwapBill then stores another set of private keys, separately from bitcoind, to control SwapBill specific outputs. These are essentially outputs that control balances in the SwapBill protocol, and it's necessary for SwapBill to track these keys independantly in order to prevent bitcoind from inadvertently consuming these outputs.

These private keys can be found in 'wallet.txt' files, located in the SwapBill data directory, under host specific subdirectories.

As well as private keys, SwapBill also stores 'secrets', located in 'secretsWallet.txt'. These secrets are host independant, and 'secretsWallet.txt' is therefore located outside of host specific subdirectories.

Don't send your wallet files or reveal the contents to anyone, unless you want them to be able to spend your swapbill, and make sure that these files are backed up securely!

CHAPTER 7

Currency Supply

A small, fixed, 'seed' amount of swapbill is currently created by the protocol, and assigned to outputs determined by the SwapBill developers, and written into the client source code. The purpose of this seed amount is to help fund development of SwapBill protocol and to help provide some initial liquidity for exchanges and backer funds.

After this seed amount, the only way to *create* any additional swapbill is by 'proof of burn' (on the host blockchain).

You can find some discussion of the concept of proof of burn here.

Essentially, if you *destroy* some of the host currency in a specified way, the swapbill protocol will credit you with a corresponding amount in swapbill.

More specifically, exactly one unit of swapbill is created for each unit of host currency that is destroyed.

This proof of burn mechanism is quite important, then, in that it provides a kind of fixed cap for the price of swapbill, in terms of the host coin. Since it's always guaranteed to be possible to create 1 unit of swapbill for 1 unit of host currency, it doesn't make any sense for anyone to ever exchange swapbill at a higher price than this (and, in fact, the exchange mechanisms don't permit any higher rate of exchange).

In most cases you shouldn't need to *actually perform* any burn transactions, since you can expect to get a better price by *exchanging* host coin for existing swapbill, instead,

Worked examples:

CHAPTER 8

Burn transactions

SwapBill allows you to create swapbill by burning host coin. (See *Currency Supply*.)

This is not the recommended way to obtain swapbill, but it's an important part of the SwapBill protocol definition, so let's see how this works!

Assuming you have sufficient host coin, a burn transaction can be submitted as follows:

```
~/git/swapbill $ python Client.py burn --amount 0.5
Loaded cached state data successfully
State update starting from block 279288
Committed state updated to start of block 279288
In memory state updated to end of block 279308
attempting to send Burn, destination output address=mhF3UNUueb4USnHJ5N9x5E67KTV9BgFDHk, amount=500000
Operation successful
transaction id : ac51ac2bfbbb16912ed7423da275f9b6acae3f466eff852595f0fbb5aa6699cf
```

8.1 Host blockchain selection

Once this goes through you will have destroyed 0.5 bitcoin, but in exchange you're credited with a corresponding amount of swapbill.

The above command defaults to the bitcoin host blockchain (and requires bitcoind to be running as an RPC server), and has the effect of burning testnet btc in exchange for bitcoin testnet swapbill, but you can do exactly the same thing with litecoind as a backend by changing the client invocation to::

```
~/git/swapbill $ python Client.py --host litecoin burn --amount 0.5
```

(In which case the client will burn 0.5 testnet ltc in exchange for 0.5 litecoin testnet swapbill!)

Throughout the examples we'll use the default host setting (bitcoin) for simplicity, but you can adapt the examples to use litecoin as necessary.

8.2 Minimum balance constraint

It's worth noting at this point that the SwapBill protocol includes a constraint on the minimum amount of swapbill associated with any given SwapBill 'account', or output. This is a financial motivation for users to minimise the number of active swapbill outputs to be tracked, and a discouragement for 'spam' outputs. For 'bitcoin swapbill', whe constraint is currently set to exactly 0.001 swapbill, or 100000 swapbill satoshis, and so that's the minimum amount we're allowed to burn. If you try to burn less, the client should refuse to submit the transaction and display a suitable

error message. (For 'litecoin swapbill the minimum balance is higher, and is currently set to exactly 0.1 swapbill, or 10000000 swapbill satoshis.)

8.3 Transaction confirmation

By default, queries such as get_balance only report the amount actually confirmed (with at least one confirmation) by the host blockchain, and so if we try querying this straight away, we won't see any swapbill credited yet for this burn:

```
~/git/swapbill $ python Client.py get_balance
Loaded cached state data successfully
State update starting from block 279288
Committed state updated to start of block 279288
In memory state updated to end of block 279308
Operation successful
balance : 0
```

But we can use the -i option to force the query to include pending transactions (from the bitcoind memory pool), and then we get:

```
~/git/swapbill $ python Client.py get_balance -i
Loaded cached state data successfully
State update starting from block 279288
Committed state updated to start of block 279288
In memory state updated to end of block 279308
in memory pool: Burn
 - 0.5 swapbill output added
Operation successful
balance : 0.5
```

And then, if we wait a bit to allow the transaction to go through, we can see this as a confirmed transaction:

```
~/git/swapbill $ python Client.py get_balance
Loaded cached state data successfully
State update starting from block 279288
Committed state updated to start of block 279289
in memory: Burn
 - 0.5 swapbill output added
In memory state updated to end of block 279309
Operation successful
balance : 0.5
```

Note that it can sometimes take a while for new blocks to be mined on the testnet blockchains (in particular with the litecoin testnet), depending on whether anyone is actually mining this blockchain, and if no one is mining (!) it can then take a while for swapbill transactions to be confirmed.

8.4 Aside: committed and in memory transactions

In the above output we can see different block counts for 'committed' and 'in memory' state, and it's worth taking a moment to explain this.

What's going on here is that the client commits state to disk to avoid spending time resynchronising on each invocation, but with this committed state lagging a fixed number of blocks (currently 20) behind the actual current block chain end.

This mechanism enables the client to handle small blockchain reorganisations robustly, without overcomplicating the client code. If there are blockchain reorganisations of more than 20 blocks this will trigger a full resynch, but blockchain reorganisations of less than 20 blocks can be processed naturally starting from the committed state.

For transaction reporting during synchronisation: * Transactions that are included in the persistent state cached to disk get prefixed by 'committed'. * Transactions that are confirmed in the blockchain but not yet cached to disk get prefixed by 'in memory'. (When you run the client again, you'll normally see these transactions repeated, unless there was a blockchain reorganisation invalidating the transaction.) * Transactions that are not yet confirmed in the blockchain, but present in the bitcoind memory pool get get prefixed with 'in memory pool'.

8.5 Better way to obtain swapbill

As noted above, burning host coin is not the recommended way to get initial swapbill. You can get a better price if you exchange host coin for swapbill, and we'll look at how to do this a bit later on..

CHAPTER 9

Pay transactions

To make a payment in swapbill, we use the 'pay' action.

As with native bitcoin and litecoin payments, the payment recipient must first generate a target address for the payment, and we can do this with the 'get_receive_address' action:

```
~/git/swapbill $ python Client.py get_receive_address
...
Operation successful
receive_address : mzBfgH8vLf9EAo4yvJz1UieXqb9jX8YzUs
```

This is actually just a standard address in the same format as you would use for the host blockchain, but the client manages this address independantly of bitcoind (see *Wallet organisation*), and uses this address to sign SwapBill outputs.

Just as with other altcoin wallets, the client and will detect any SwapBill outputs paying to this address and add the corresponding swapbill amounts to your balance.

To pay some swapbill in to this address:

```
~/git/swapbill $ python Client.py pay --amount 0.1 --toAddress mzBfgH8vLf9EAo4yvJz1UieXqb9jX8YzUs
Loaded cached state data successfully
State update starting from block 279379
Committed state updated to start of block 279379
In memory state updated to end of block 279399
attempting to send Pay, change output address=n319WGe5egQi3WRAxJ3RKAHFSafEvMSGxd, destination output
Operation successful
transaction id : b3fb87e0750e572ad3b7407f4d1ddfa829afd7ef3da7872dd744499ae2b03307
```

9.1 Working with multiple wallets

In this case we're actually just paying ourselves. It's also possible to manage multiple swapbill wallets independantly, by changing the client data directory, and to use this to try out transactions between different wallet 'owners'.

First, let's create a new SwapBill wallet 'owner' (corresponding to an alternative SwapBill data directory):

```
~/git/swapbill $ mkdir alice
~/git/swapbill $ python Client.py --dataDir alice get_receive_address
Failed to load from cache, full index generation required (no cache file found)
...
Operation successful
receive_address : n4M62XjmWCS9qhKCzaPDoKYmicAVVXoyGS
```

Note that you need to create the new data directory before invoking the client. The client won't create this directory for you.

And now we can pay 'alice' from the default wallet:

> ~/git/swapbill $ python Client.py -pay –amount 0.1 –toAddress mm-dAwus4b6chWzvRtNVcL2YfxvWTeWUcq3

In this case, the default wallet owner is debited the swapbill payment amount, and this is credited to 'alice'.

9.2 Waiting for change to clear

If you try the examples posted here directly after the previous 'burn' transaction examples there should be enough funds available for both of these pay transactions.

If you get an error `Operation failed: Insufficient swapbill for transaction.` this may mean that you need to wait for a previous transaction to be confirmed, in order for the change from that transaction to become available for spending once again. (This works in exactly the same way as native bitcoin transactions, and exactly the same issue occurs there.)

You can check the amount actually available to spend at any one instant with `python Client.py get_balance`.

If you add the -i option to this query, so `python Client.py get_balance -i`, this tells you how much *will have available* after all pending transactions (in the bitcoind memory pool) have cleared.

(The -i option just means 'include memory pool'.)

If `get_balance` doesn't show enough funds for a transaction, but `get_balance -i` does, then you just need to wait for your memory pool transactions to go through.

CHAPTER 10

Trading transactions

The swapbill protocol includes extensive support for decentralised exchange between swapbill and the host currency, based on buy and sell offer information posted on the block chain and with protocol rules for matching these buy and sell offers.

Three additional client actions (and associated transaction types) are provided for this exchange mechanism: * buy_offer * sell_offer * complete_sell

10.1 Terminology

The client uses 'buy' to refer to buying host coin (with swapbill), and 'sell' to refer to selling host coin (for swapbill), and we'll use the same convention in this documentation.

10.2 Buy transactions

Buying host coin with swapbill is the most straightforward use case because this requires just one transaction to post a trade offer, with the SwapBill protocol then taking over handling of the trade completely from there.

10.3 Sell transactions

There are then two different kinds of host coin sell offer.

'Backed' sell offers are the recommended method, when there are are backers available. In this case the backer has already committed a swapbill amount to cover the trade, so you just need to make one sell offer transaction, and the backer then takes care of exchange completion payments.

When no backers are available, you can also make unbacked sell offers, but are then responsible for subsequent exchange completion payments yourself.

When trade offers go through, swapbill amounts are associated with the offers and paid out again when offers are matched, according to the SwapBill protocol rules.

In the case of buy offers this is the swapbill amount being offered in exchange for host coin.

In the case of sell offers this is a deposit amount, in swapbill, currently set to 1/16 of the trade amount, which is held by the protocol and paid back on condition of successful completion. If the seller completes the trade correctly then this deposit is refunded, but if the seller fails to make a completion payment after offers have been matched then the deposit is credited to the matched buyer (in compensation for their funds being locked up during the trade).

Exchange rates are always fractional values between 0.0 and 1.0 (greater than 0.0 and less than 1.0), and specify the number of host coins per swapbill (so an exchange rate of 0.5 indicates that one host coin exchanges for 2 swapbill).

A couple of other details to note with regards to trading: * the sell offer transactions also require an amount equal to the protocol minimum balance constraint to be 'seeded' into the sell offer (but, unlike the deposit, this seed amount will be returned to the seller whether or not trades are successfully completed) * trade offers are subject to minimum exchange amounts for both the swapbill and host coin equivalent parts of the exchange * trade offers may be partially matched, and host coin sell offers can then potentially require more than completion transaction * matches between small trade offers are only permitted where the offers can be matched without violating the minimum exchange amounts and minimum offer amounts for any remainder

The trading mechanism provided by SwapBill is necessarily fairly complex, and a specification of the *exact* operation of this mechanism is beyond the scope of this document, but we'll show a concrete example of trading worked through in the client to show *how to use* the mechanism.

CHAPTER 11

Backed sell transactions

A backed sell transaction enables you to sell host currency for swapbill (to obtain some initial swapbill, for example), with just one single sell offer transaction.

Backed sell transactions are actually the recommended way to obtain some initial swapbill, in preference to burn transactions.

As with burn transactions, no swapbill balance is required in order to make a backed sell offer, but this does depend on some backing amount having already been committed by a third party, and some commission is payable to the backer for these transactions.

(Because commission is paid to backer, with the rate of commission subject to market forces, *if* there is swapbill available for exchange then there *should* be backers available, otherwise this is an indication that swapbill supply is insufficient, and so creating more swapbill by burning is appropriate!)

11.1 Checking backers

You can use the 'get_sell_backers' action to check if there are backers available, and to find out information about the backers such as rate of commission being charged, as follows:

```
~/git/swapbill $ python Client.py get_sell_backers
...
Operation successful
host coin sell backer index : 0
    I am backer : False
    transactions covered : 1000
    backing amount : 190
    blocks until expiry : 29865
    maximum exchange swapbill : 0.17788235
    backing amount per transaction : 0.19
    expires on block : 309437
    commission : 0.005
```

The key values to look at are 'commission' and 'transactions covered'.

The commission value here indicates that 0.5% commission is payable to the backer on the amount of host coin offered for sale.

The 'transactions covered' value is calculated based on 'backing amount' and 'backing amount per transaction', and this tells us how many transactions (at a minimum) can be guaranteed by the current backing amount, and therefore how safe it is to use this backer for sell transactions.

The backed trade mechanism works by backers committing funds to guarantee trades for a certain number of transactions in advance.

If we submit a sell transaction using this backer, and then 1000 other valid trade transactions to the same backer (all using the maximum allowed backing amount) all come through on the blockchain in between this backer query and our sell transaction, then it's possible that our transaction does not get backed, and we lose the host coin amount paid in to our sell transaction.

As long as our sell transaction comes through to the blockchain with *less than* 1000 other transactions to the same backer in between, however, SwapBill uses the funds committed by the backer to guarantee our exchange.

Lets go ahead and exchange some host coin for swapbill, through this backer.

11.2 Listing buy offers

The next step is to check the buy offers currently posted to the blockchain, to get an idea of the current exchange rate:

```
~/git/swapbill $ python Client.py get_buy_offers
...
Operation successful
exchange rate : 0.92
    mine : True
    swapbill offered : 1.1
    host coin equivalent : 1.012
exchange rate : 0.95
    mine : True
    swapbill offered : 2
    host coin equivalent : 1.9
```

The best offer comes first, with 1.1 swapbill offered at an exchange rate of 0.92 host coin per swapbill.

Let's assume we're ok with making an exchange at this rate.

11.3 Maximum exchange with rate

As we saw above, each backer has a maximum backing amount per individual transaction.

This is important in order to prevent a single transaction using all of the backing amount, and to provide some solid guarantees about the number of transactions that can be backed. But it also means that there is a maximum amount we can exchange through the backer in any one transaction.

The get_sell_backers query above gave us a 'maximum exchange swapbill' value, which specified the maximum amount which can be exchanged, specified in swapbill, but we're going to need to specify an amount in host coin in our sell transaction.

We could get out a calculator, and work this out, but it's easier to let the get_sell_backers query do this for us:

```
~/git/swapbill $ python Client.py get_sell_backers --withExchangeRate 0.92
...
Operation successful
host coin sell backer index : 0
    backing amount : 190
    maximum exchange swapbill : 0.17788235
    transactions covered : 1000
    expires on block : 309437
    maximum exchange host coin : 0.16365176
```

```
    commission : 0.005
    backing amount per transaction : 0.19
    I am backer : False
    blocks until expiry : 29863
```

So the maximum amount if host coin we can exchange through this backer in a single transaction, at an exchange rate of 0.92 host coin per swapbill, is 0.16365176.

11.4 Backed sell transaction

After checking that we have enough funds available in our host coin wallet:

```
~/git $ bitcoin/src/bitcoin-cli getbalance
2.98189956
```

We can go ahead and submit our backed sell transaction as follows:

```
~/git/swapbill $ python Client.py sell_offer --hostCoinOffered 0.16365176 --exchangeRate 0.92 --backe
...
attempting to send BackedSellOffer, sellerReceive output address=n4WPsVHA3pdAjmDpfZy6dxZ6pigDgEBws7,
Operation successful
transaction id : 4704f8b40446c123bb2a715abaa3f100f99cd499886a529216904053361ba175
```

Note that this transaction doesn't need *any* initial swapbill balance. It is funded purely in host coin.

Assuming there are no other competing sell offers, this offer should go through directly (in the next block) and be matched with the corresponding buy offer:

```
~/git/swapbill $ python Client.py get_balance
Loaded cached state data successfully
State update starting from block 279555
Committed state updated to start of block 279556
in memory: BackedSellOffer
 - 0.17788234 swapbill output added
In memory state updated to end of block 279576
Operation successful
balance : 0.17788234
```

And we can see that the buy offer has been updated accordingly:

```
~/git/swapbill $ python Client.py get_buy_offers
...
Operation successful
exchange rate : 0.92
    host coin equivalent : 0.84834825
    mine : False
    swapbill offered : 0.92211766
exchange rate : 0.95
    host coin equivalent : 1.9
    mine : False
    swapbill offered : 2
```

(The top buy offer there is a remainder left over after this offer was partially matched by our sell.)

11.5 Commission

Let's check the amount debited from our host coin wallet (after allowing change to clear):

```
~/git $ bitcoin/src/bitcoin-cli getbalance
2.81442955
```

So, this cost us 0.16747001 host coin. This corresponds to:

- the amount of host coin we offered in our sell transaction (0.16365176)
- plus backer commission of 0.005 * 0.16365176 = 0.000818259
- plus 0.003 in transaction fees

By default, backers commission is added to the amount specified for hostCoinOffered in the sell_offer command. If we want to specify an amount to be paid *including backer commission* then we can do this by setting the –includesCommission option.

11.6 Pending exchange

If we check with the get_pending_exchanges command, just after our sell transaction goes through, we can see that a pending exchange has been created, corresponding to this offer:

```
~/git/swapbill $ python Client.py get_pending_exchanges
...
Operation successful
pending exchange index : 0
    I am seller (and need to complete) : False
    I am buyer (and waiting for payment) : False
    backer id : 0
    blocks until expiry : 13
    confirmations : 3
    swap bill paid by buyer : 0.17788234
    expires on block : 279591
    deposit paid by seller : 0.01111765
    outstanding host coin payment amount : 0.16365176
```

This shows that the backer needs to complete the exchange with the person who made the buy offer.

Normally, the backer will go ahead and complete this exchange after a certain number of blocks have been confirmed. But this is something that we don't have to worry about, at all, in this case, as the backed exchange mechanism insulates us completely from exchange completion details. (If the backer fails to complete the exchange with the buyer, *the backer* will lose their deposit.)

It's also possible to make an exchange *without a backer*. In this case no backer commission is payable, but you have to take care of making exchange completion transactions yourself. (This is something we'll look at a bit later on.)

11.7 Waiting for a match

In this case our sell offer was matched immediately, with an existing buy offer.

In other situations there may not be a matching buy offer (if we chose a lower exchange rate, for example), competing sell offers may come through and match a buy offer before our offer.

And it's possible you offer to partially match with a partial remainder offer outstanding.

In these case's you'll need to wait for your offer to match, or for each part of your offer to match, before being credited with the corresponding swapbill.

CHAPTER 12

Buy transactions

Ok, so we've looked at how to get hold of some swapbill, either through a backed exchange, or by burning host coin.

SwapBill is intended to serve a fairly specific purpose, however, (for facilitating decentralised cross currency exchange, specifically), and when you've finished using your swapbill you will most likely want to exchange this back for host currency.

You can do this with the buy offer transaction (buying host coin with swapbill). The process for this is similar to backed sell offers, but even more straightforward, because there's no need to select a backer in this case.

12.1 Checking sell offers

Starting with a buyer, who has 1.5 swapbill they want to exchange for host coin:

```
~/git/swapbill $ python Client.py get_balance
...
Operation successful
balance : 1.5
```

Let's check the current list of sell offers:

```
~/git/swapbill $ python Client.py get_sell_offers
...
Operation successful
exchange rate : 0.91
    host coin offered : 0.9
    deposit : 0.06181319
    swapbill equivalent : 0.98901099
    mine : False
exchange rate : 0.88
    host coin offered : 1
    deposit : 0.07102273
    swapbill equivalent : 1.13636364
    mine : False
```

12.2 Matching the top offer

The best rate here is 0.91 host coin per swapbill. Let's assume we're ok with exchanging at anything down to 0.9 host coin per swapbill. So, we'll try and match that top offer first:

```
~/git/swapbill $ python Client.py buy_offer --swapBillOffered 0.98901099 --blocksUntilExpiry 1 --exch
...
attempting to send BuyOffer, hostCoinBuy output address=mijmaJQvuLdpbXNqx5MRz6qnTUTZALK2Qy, exchangeR
Operation successful
transaction id : 34be36f0bdb7f165838bb1210f0eaf0aa8a91416a6f4c38e0b3431088ebbdf5f
```

This is similar to the sell offer we posted in the previous example, but there a couple of other points to note here.

First of all, note that we've specified a value for blocksUntilExpiry. This is because we just want to match the top existing offer, but there is a possibility that someone else matches that offer first. So we've set our transaction to expire in the next block, if it doesn't match, so that we can then go on and use the swapbill in another offer without having to wait too long.

A second point to note is that the decimal fractions displayed by queries such as get_sell_offers, and passed in for transaction parameters are actually *exact* values, and not subject to approximation errors during parsing or display. So we can copy the exact text printed for the top sell offer and expect our offer to match this exactly. (This is a subtle point, but nevertheless quite an important implementation detail!)

No other offers come in before our buy offer, then, and this matches the top offer.

The amount offered is debited from our balance:

```
~/git/swapbill $ python Client.py get_balance
...
Operation successful
balance : 0.51098901
```

And we can see that the top sell offer has been removed:

```
~/git/swapbill $ python Client.py get_sell_offers
...
in memory: BuyOffer
 - 1.5 swapbill output consumed
 - 0.51098901 swapbill output added
In memory state updated to end of block 279587
Operation successful
exchange rate : 0.88
    deposit : 0.07102273
    swapbill equivalent : 1.13636364
    host coin offered : 1
    mine : False
```

12.3 Posting a speculative offer

We decide to post the remaining funds again, in another buy offer:

```
~/git/swapbill $ python Client.py --buy_offer --swapBillOffered 0.51098901 --exchangeRate 0.9
...
attempting to send BuyOffer, hostCoinBuy output address=mo8ACE96HGVUfrthq4u1g4nZCZTB94jEuS, exchangeR
Operation successful
transaction id : 98998855e17ffec7a63e9342981d4fad8f5acc97f4ae51dd27915479ac20863e
```

Again, note that we can post the exact current balance value, and the client will then include the whole balance in the offer. Remember that the SwapBill protocol includes a minimum balance constraint, so we're not permitted to submit transactions that would leave a very small amount of change. (If so, the client will report an error, and refuse to submit the transaction. Try spending very slightly less than your current balance, to see this in practice.)

In this case we didn't specify blocksUntilExpiry. For buy transactions this currently defaults to 8. For backed sell transactions there is no expiry and for unbacked sells the default is just 2 blocks (because unbacked sells need to be followed up with exchange completion for each matching buyer).

This offer doesn't match any existing sell offer, and is therefore added to the existing order book as an outstanding offer:

```
~/git/swapbill $ python Client.py get_buy_offers
...
Operation successful
exchange rate : 0.9
    swapbill offered : 0.51098901
    host coin equivalent : 0.45989011
    mine : True
exchange rate : 0.92
    swapbill offered : 0.92211766
    host coin equivalent : 0.84834825
    mine : False
exchange rate : 0.95
    swapbill offered : 2
    host coin equivalent : 1.9
    mine : False
```

If no-one posts a matching offer before the end of the expiry period, the swapbill amount offered will be returned to our active balance. But, as it is, a couple of sell offers come along in the next few blocks, and match the outstanding offer remainder.

We can see the SellOffer transactions come up in the sync output, and we can also see that the buy offer has been matched and is no longer present:

```
~/git/swapbill $ python Client.py get_buy_offers
Loaded cached state data successfully
State update starting from block 279570
...
in memory: SellOffer
 - trade offer updated
in memory: SellOffer
 - trade offer updated
In memory state updated to end of block 279591
Operation successful
exchange rate : 0.92
    swapbill offered : 0.92211766
    host coin equivalent : 0.84834825
    mine : False
exchange rate : 0.95
    swapbill offered : 2
    host coin equivalent : 1.9
    mine : False
```

It turns out that our second offer was actually matched by two smaller sell offers. And so at this point, we now have three trade offer matches outstanding, waiting for final host coin payments from the seller to complete.

We can see this with the get_pending_exchanges query:

```
~/git/swapbill $ python Client.py get_pending_exchanges
...
Operation successful
pending exchange index : 1
    swap bill paid by buyer : 0.98901099
    I am buyer (and waiting for payment) : True
```

```
    deposit paid by seller : 0.06181319
    I am seller (and need to complete) : False
    expires on block : 279602
    blocks until expiry : 12
    confirmations : 4
    outstanding host coin payment amount : 0.9
pending exchange index : 2
    swap bill paid by buyer : 0.33333333
    I am buyer (and waiting for payment) : True
    deposit paid by seller : 0.02083334
    I am seller (and need to complete) : False
    expires on block : 279605
    blocks until expiry : 15
    confirmations : 1
    outstanding host coin payment amount : 0.3
pending exchange index : 3
    swap bill paid by buyer : 0.17765568
    I am buyer (and waiting for payment) : True
    deposit paid by seller : 0.01110348
    I am seller (and need to complete) : False
    expires on block : 279606
    blocks until expiry : 15
    confirmations : 1
    outstanding host coin payment amount : 0.15989011
```

As a host coin buyer, we don't have to take any action here. Either the seller pays the required host coin amount and completes the exchange, or we are refunded our swapbill plus some deposit paid by the seller as part of their sell offer.

A bit later we can see that one of the exchanges has been completed:

```
~/git/swapbill $ python Client.py get_pending_exchanges
...
Operation successful
pending exchange index : 1
    blocks until expiry : 11
    confirmations : 5
    I am seller (and need to complete) : True
    I am buyer (and waiting for payment) : False
    expires on block : 279602
    deposit paid by seller : 0.06181319
    swap bill paid by buyer : 0.98901099
    outstanding host coin payment amount : 0.9
pending exchange index : 2
    blocks until expiry : 14
    confirmations : 2
    I am seller (and need to complete) : True
    I am buyer (and waiting for payment) : False
    expires on block : 279605
    deposit paid by seller : 0.02083334
    swap bill paid by buyer : 0.33333333
    outstanding host coin payment amount : 0.3
```

You can check your host coin balance separately, with `bitcoin/src/bitcoin-cli getbalance`, to confirm that you've received the host coin amount for this exchange. (The SwapBill client verifies that the payment transaction is received, but does not track your host coin balance.)

CHAPTER 13

Unbacked sell transactions

In addition to the backed litecoin sell offers, it's also possible to make *unbacked* sell offers.

The key differences between the two types of sell offer are that:

- some initial swapbill is required in order to make an unbacked sell offer
- in the case of an unbacked sell offer, it is up to the seller to make final completion payments for each trade offer match
- a deposit is payed in to unbacked sell offers, and will be lost if the final completion payment is not made (e.g. if your internet goes down, or something like this!)
- backed sells only require one transaction from the seller, and there is no risk to the seller after that transaction has gone through
- backed sells are based on a lump sum committed to the SwapBill protocol by the backer, however, and then only guarantee offers up to a maximum number of transactions, and there is a theoretical possibility for lots of transactions to come through an consume all of the backer amount
- some commission is payable to the backer for each backed sell offer
- unbacked sells have an expiry period, which can be set when you make the offer
- backed sells never expire

Roughly speaking, backed ltc sells are good for smaller transactions, and are the best way to obtain swapbill initially, but for larger transactions, and if you can be confident about being able to submit completion transactions (e.g. if you have a backup internet connection!) then unbacked sells can be preferrable.

To make an unbacked sell offer we start with a sell_offer action, as before, but in this case we *don't* specify a value for backerID (and so don't need to check for backers and backer details).

Our seller starts with some swapbill:

```
~/git/swapbill $ python Client.py get_balance
...
Operation successful
balance : 3.00531212
```

13.1 Checking buy offers

As before, we check the current set of buy offers:

```
~/git/swapbill $ python Client.py get_buy_offers
...
Operation successful
exchange rate : 0.92
    swapbill offered : 0.92211766
    host coin equivalent : 0.84834825
    mine : False
exchange rate : 0.95
    swapbill offered : 2
    host coin equivalent : 1.9
    mine : False
```

13.2 Sell offer

Let's try and match the top offer:

```
~/git/swapbill $ python Client.py sell_offer --hostCoinOffered 0.84834825 --exchangeRate 0.92
...
attempting to send SellOffer, hostCoinSell output address=moyQMZZDDfS4jGARJojCFY62Lcc8CMuYYY, exchang
Operation successful
transaction id : bd4906dc3f85bbc670290e804ed59b0275a9aec7f25aef6940cc56976400a226
```

This goes through successfully, and we can see that the buy offer has been matched:

```
~/git/swapbill $ python Client.py get_buy_offers
...
in memory: SellOffer
 - 3.00531212 swapbill output consumed
 - 2.94767976 swapbill output added
In memory state updated to end of block 279593
Operation successful
exchange rate : 0.95
    mine : False
    host coin equivalent : 1.9
    swapbill offered : 2
```

A deposit proportional to the amount of swapbill we are looking to exchange has been taken from our current balance, but also a seed amount equivalent to the minimum balance protocol constraint (currently set to 0.001 for bitcoin swapbill):

```
~/git/swapbill $ python Client.py get_balance
...
Operation successful
balance : 2.96535406
```

Now it is up to us to complete. We can see the pending exchange with get_pending_exchanges:

```
~/git/swapbill $ python Client.py get_pending_exchanges
...
Operation successful
pending exchange index : 4
    deposit paid by seller : 0.05763236
    expires on block : 279608
    swap bill paid by buyer : 0.92211766
    outstanding host coin payment amount : 0.84834825
    I am seller (and need to complete) : True
    I am buyer (and waiting for payment) : False
```

```
    confirmations : 1
    blocks until expiry : 15
```

We should wait for a few more blocks to go through before completing the exchange, in case of blockchain reorganisation.

(We can ignore the issue of blockchain reorganisation to a large extent for a lot of other 'single transaction' actions, but this is something we need to be more careful about specifically in the case completion transactions. In the case of backed sells this is something that backers normally have to worry about for you!)

Once we're happy with the number of comfirmations for our pending exchange, the actual completion transaction is then very straightforward:

```
~/git/swapbill $ python Client.py complete_sell --pendingExchangeID 4
...
attempting to send ExchangeCompletion, destinationAddress=n25vgZ5ahLxmM7YujMmRnFGVPUTZA6aooL, destina
Operation successful
transaction id : ca7a712cb8746122aa55f2b49a298099a4b4f1927375cf67e85b62486b2b1421
```

Once this transaction has gone through we're refunded the deposit, and the seed amount, and credited the swapbill amount corresponding to our exchange:

```
~/git/swapbill $ python Client.py get_balance
in memory: ExchangeCompletion
 - trade offer updated
Operation successful
balance : 3.94510408
```

CHAPTER 14

Cross chain exchanges

So far we've looked exclusively at operations on one single blockchain, in the examples so far. But SwapBill can be embedded in more than one blockchain, and includes support for trustless exchange of coin *between different host blockchains*.

14.1 Working with more than one host

The examples so far all use the default value for SwapBill's host parameter, which is 'bitcoin', but all the transactions we've shown up to here can also be applied on other blockchains.

The hosts currently supported by the client are 'bitcoin' and 'litecoin'.

The following simple get_balance command will connect to a bitcoin RPC server, and submit transactions on the bitcoin blockchain:

```
~/git/swapbill $ python Client.py get_balance
```

By specifying an explicit '–host' setting in the following get_balance command, however, we can tell SwapBill to connect to a *litecoin* RPC server, and submit transactions on the *litecoin* blockchain:

```
~/git/swapbill $ python Client.py --host litecoin get_balance
```

If we have both bitcoin and litecoin RPC servers running, we can work with both blockchains at the same time, with the host set independantly for each individual client command.

SwapBill wallets are managed independantly for each host (in subdirectories in the SwapBill data directory).

14.2 Cross chain exchange support

The support provided by SwapBill for cross chain exchange is less extensive, currently, than support for exchanges between swapbill and host coin on the same blockchain.

When exchanging between swapbill and host coin on the same blockchain, buy and sell offer books and offer management is included in the SwapBill protocol, the backer mechanism enables us to provide single transaction exchanges in both directions, and deposits are paid in case of failure to complete exchanges.

In the case of cross chain exchanges, the two parties wanting to exchange need to use some external mechanism (outside of the SwapBill protocol) to find each other and agree on a suitable rate of exchange.

And then, these two parties each depend on the other party to actually go forward and complete the exchange. The exchange mechanism is 'atomic' in that both parties are protected from losing their coin, but it's possible for either

of the parties to 'bail out' of the exchange without any loss of deposit, and in this case the other party may suffer the inconvenience of their part of the exchange being locked up for a specified period of time (depending on the parameters chosen for the exchange) before these funds are finally refunded.

14.3 Pay on reveal secret

The cross chain exchange mechanism is based on a fairly straightforward fundamental mechanism, a special version of the 'pay' transaction which only actually pays the destination adress on condition that a specified secret is revealed.

You can submit this special case of pay transaction as follows:

```
~/git/swapbill $ python Client.py pay --amount 0.1 --toAddress mibcs5rAjrw4Xmb6MSZKQgxURVVrja1Qjt --k
...
attempting to send PayOnRevealSecret, change output address=mj3fuoJ25dETSUmjWv7i3fBj3rVK1URGRA, desti
Operation successful
transaction id : 713e7047fb2c1fe080c244fa7ca4da74eb9fa06ad576d0bf2cb62dd6137f19bf
```

This won't pay the destination address straight away, but will instead create a pending payment. The current set of pending payments can then be queried with the get_pending_payments command:

```
~/git/swapbill $ python Client.py get_pending_payments
...
Operation successful
pending payment index : 1
    paid by me : True
    I hold secret : True
    blocks until expiry : 100
    amount : 0.1
    paid to me : True
    confirmations : 1
    expires on block : 279703
```

In the 'pay' command SwapBill created a secret for you, and keeps track of this in a separate 'secrets wallet'.

If you have the secret for a pending payment, you can use the reveal_secret_for_pending_payment command to reveal this secret and force the payment to complete:

```
~/git/swapbill $ python Client.py reveal_secret_for_pending_payment --pendingPaymentID 1
...
attempting to send RevealPendingPaymentSecret, pendingPayIndex=1, publicKeySecret=b'7Cg\xdc\x16\x16\x
Operation successful
transaction id : e9ff101d0046edf4a65b4bb0916d8d6b430e265856a68208c0aa5a69216807e5
```

14.4 Payment on someone else's secret

It's also possible to make a payment dependant on *someone else's* secret being revealed, with the counter_pay command:

```
~/git/swapbill $ python Client.py counter_pay --help
usage: SwapBillClient counter_pay [-h] --amount AMOUNT --toAddress TOADDRESS
                                  [--blocksUntilExpiry BLOCKSUNTILEXPIRY]
                                  --pendingPaymentHost {bitcoin,litecoin}
                                  --pendingPaymentID PENDINGPAYMENTID

optional arguments:
```

```
  -h, --help            show this help message and exit
  --amount AMOUNT       amount of swapbill to be paid, as a decimal fraction
                        (one satoshi is 0.00000001)
  --toAddress TOADDRESS
                        pay to this address
  --blocksUntilExpiry BLOCKSUNTILEXPIRY
                        if the transaction takes longer than this to go
                        through then the transaction expires (in which case no
                        payment is made and the full amount is returned as
                        change)
  --pendingPaymentHost {bitcoin,litecoin}
                        host blockchain for target payment, can currently be
                        either 'litecoin' or 'bitcoin'
  --pendingPaymentID PENDINGPAYMENTID
                        the id of the pending payment, on the specified
                        blockchain
```

This does essentially the same as the pay transaction shown above (with '–onRevealSecret'), but with the difference being that, in this case, instead of generating a secret, the counter_pay command makes the payment dependant on *the same secret* as another pending payment.

The '–pendingPaymentHost' and '–pendingPaymentID' are used to specify which pending payment the secret should be taken from. Importantly, the host blockchain for the pending payment that is referenced in this way can be specified independantly of the payment being submitted.

14.5 Secrets watch list

When you make a submit a counter_pay action, SwapBill also adds the secret to a watch list. If that secret is revealed, subsequently, during block chain synchronisation, SwapBill will then add this secret to your secrets wallet.

14.6 Putting it together

Let's put all the above together, then, and see how this can be used for cross chain exchange.

We'll simulate two parties for the exchange by setting up separate SwapBill data directories for each party:

```
~/git/swapbill $ mkdir a
~/git/swapbill $ python Client.py --dataDir a get_balance
Failed to load from cache, full index generation required (no cache file found)
State update starting from block 278805
Committed state updated to start of block 279587
In memory state updated to end of block 279607
Operation successful
balance : 0
~/git/swapbill $ mkdir b
~/git/swapbill $ python Client.py --dataDir b get_balance
Failed to load from cache, full index generation required (no cache file found)
State update starting from block 278805
Committed state updated to start of block 279587
In memory state updated to end of block 279607
Operation successful
balance : 0
```

14.7 Initial balances

The two parties for the exchange will need 'bitcoin swapbill' and 'litecoin swapbill' to exchange. (To exchange 'native' bitcoin and litecoin, these should first be converted into 'bitcoin swapbill' and 'litecoin swapbill' with the on-chain exchange mechanisms described in the previous examples.)

For this example we'll give **a** a balance of 3.5 bitcoin swapbill, and **b** a balance of 350 litecoin swapbill, which they then want to exchange.

For **a**:

```
~/git/swapbill $ python Client.py --dataDir a get_receive_address
...
Operation successful
receive_address : mhjZL4K111nP6UPxait6jFpQfEAdoKVwVi
~/git/swapbill $ python Client.py pay --toAddress mhjZL4K111nP6UPxait6jFpQfEAdoKVwVi --amount 3.5
...
attempting to send Pay, change output address=n4UDtohgBFWwtEyJnSSQuyA5ZEcGdf5Tq5, destination output
Operation successful
transaction id : 90b85732f46a85c4c51bdf917903ed747dfa0ba7bb01250acbcb708217529385
~/git/swapbill $ python Client.py --dataDir a get_balance -i
...
Operation successful
balance : 3.5
```

And for **b**:

```
~/git/swapbill $ python Client.py --dataDir b --host litecoin get_receive_address
...
Operation successful
receive_address : mnuKWDoH4wME5YdRQy6xM2y42QsHkCK4Fi
~/git/swapbill $ python Client.py --host litecoin pay --toAddress mnuKWDoH4wME5YdRQy6xM2y42QsHkCK4Fi
...
attempting to send Pay, change output address=n2qKD4wDo1Cp7f2pd1NXMhv9rhwyaxVREq, destination output
Operation successful
transaction id : c0578cfcf768043b6711d7c2730a1c86f35021870c4e76d60c04f1043448e704
~/git/swapbill $ python Client.py --dataDir b --host litecoin get_balance -i
...
Operation successful
balance : 350
```

For this to work, note that we need *both* litecoind *and* bitcoind running and set up as RPC servers. (See *Setting up the host RPC server*.)

We used payments from the default swapbill wallet, in each case, but you could also use burn transactions or exchanges to create these initial balances.

14.8 Procedure for exchange

The basic procedure for the exchange will be as follows:

- **a** creates a litecoin swapbill receive address and sends this to **b**
- **b** creates a bitcoin swapbill receive address and sends this to **a**
- **a** submits a pay on reveal secret transaction (to **b**'s receive address), with quite a long time until expiry
- **b** checks the details for this payment, and makes sure this is confirmed, and then, if everything is ok, submits a counter_pay transaction (to **a**'s receive address), with a much shorter time until expiry

- both payments are now dependant on the same secret, which is currently known only to **a**
- **a** can now submit a reveal secret transaction, enabling the counter_pay to go through
- **b** then obtains the secret (during a subsequent syncronisation), and can submit a reveal secret transaction to enable the first payment to go through

Let's see how this works out, though, in concrete SwapBill client invocations..

14.9 Generate receive addresses

The parties generate receive addresses (and send these to each other):

```
~/git/swapbill $ python Client.py --dataDir a --host litecoin get_receive_address
...
Operation successful
receive_address : n3QncHC6iAGPo5G4bFNZ9BeAfcb5KWAaEq
~/git/swapbill $ python Client.py --dataDir b get_receive_address
...
Operation successful
receive_address : mu9uDM3r571d198iaqznBfZFEtHfxCEpDA
```

14.10 First payment, a to b

a submits a first pay on reveal secret transaction:

```
~/git/swapbill $ python Client.py --dataDir a pay --amount 3.5 --toAddress mu9uDM3r571d198iaqznBfZFEt
Loaded cached state data successfully
State update starting from block 279597
Committed state updated to start of block 279597
in memory: Pay
 - 3.5 swapbill output added
In memory state updated to end of block 279617
attempting to send PayOnRevealSecret, change output address=n1YrUfwjZkKPjgpVViuBzaTtTahjWBwtwE, desti
Operation successful
transaction id : db7913fda875b3dbf94f1a09e272f0d2279175be27b3133a3f97f4b45ce63828
```

Note that we specified 18 for '–blocksUntilExpiry' here, which, based on approximately 10 minutes per block for bitcoin, works out at approximately 3 hours.

A different number could be used here, but:

- This needs to allow plenty of time for **b** to set up a counter payment, with shorter expiry period, and checks for a certain number of confirmations.
- If **b** doesn't play ball, and doesn't move forward with their part of the exchange, **a**'s swapbill will be locked up until the end of this expiry period.

b checks this payment:

```
~/git/swapbill $ python Client.py --dataDir b get_pending_payments
...
Operation successful
pending payment index : 2
    paid to me : True
    confirmations : 1
    paid by me : False
```

```
    expires on block : 279636
    blocks until expiry : 17
    amount : 3.5
```

but then waits until the payment has a reasonable number of confirmations before making a counter payment.

14.11 Counter payment, b to a

Once **b** is satisfied that there are enough confirmations for the first payment (and confident that this first payment won't be backed out by a blockchain reorganisation), **b** proceeds with the counterpayment, as follows:

```
~/git/swapbill $ python Client.py --dataDir b --host litecoin counter_pay --amount 350 --toAddress n
...
attempting to send PayOnRevealSecret, change output address=mzy6QnsqiQjFAritKoqKUGvRyeJuy7bo1J, desti
Operation successful
transaction id : cbcbfdbf6d1d60750bfe4f2eed07cdf7abfcfadf9d87e765e9a5de9f151a1df3
```

The value of 36 for '–blocksUntilExpiry' here, based on approximately 2.5 minutes per block for litecoin, works out at approximately 1.5 hours.

This expiry delay needs to be: * long enough to give **a** enough time to accept the counter payment (including waiting for enough confirmations), but * short enough that, even if **a** waits until this is nearly expired before accepting, **b** still has plenty of time to accept the first payment.

a checks this counter payment:

```
~/git/swapbill $ python Client.py --dataDir a --host litecoin get_pending_payments
...
Operation successful
pending payment index : 3
    blocks until expiry : 36
    paid to me : True
    confirmations : 1
    amount : 350
    paid by me : False
    I hold secret : True
    expires on block : 383216
```

Note that both 'paid to me' and 'I hold secret' show True, which shows that **a** can now go ahead and accept this counter payment.

14.12 Counter payment accepted by a

Once **a** is satisfied that there are enough confirmations for the counterpayment, **a** goes ahead and reveals the secret, to accept this:

```
~/git/swapbill $ python Client.py --dataDir a --host litecoin reveal_secret_for_pending_payment --per
...
attempting to send RevealPendingPaymentSecret, pendingPayIndex=3, publicKeySecret=b'\xa5D\xa5q\xa5\x7
Operation successful
transaction id : 38fa2ab46a26466d2b31e0921e4e76858655e4736565d4c98dc9925099cce955
```

The counterpayment goes through, and **a** receives the litecoin part of the exchange:

```
~/git/swapbill $ python Client.py --dataDir a --host litecoin get_balance
...
Operation successful
balance : 350
```

But, for this, **a** was forced to reveal their secret.

14.13 First payment completed by b

When **b** next synchronises, this secret is detected and added to their secrets wallet:

```
~/git/swapbill $ python Client.py --dataDir b --host litecoin get_balance
Loaded cached state data successfully
State update starting from block 383159
Committed state updated to start of block 383160
...
in memory: storing revealed secret with hash 4b14f0fc03d9aca2509688e3da04678bc418aad1
In memory state updated to end of block 383181
Operation successful
balance : 0
```

And, if we now check the status for the first payment, on the bitcoin blockchain:

```
~/git/swapbill $ python Client.py --dataDir b get_pending_payments
...
Operation successful
pending payment index : 2
    paid by me : False
    paid to me : True
    I hold secret : True
    blocks until expiry : 15
    confirmations : 3
    amount : 3.5
    expires on block : 279636
```

We can see that **b** is now able to complete this payment, as follows:

```
~/git/swapbill $ python Client.py --dataDir b reveal_secret_for_pending_payment --pendingPaymentID 2
...
attempting to send RevealPendingPaymentSecret, pendingPayIndex=2, publicKeySecret=b'\xa5D\xa5q\xa5\x
Operation successful
transaction id : d2c9d58948e23a21d655d89c1de7d2688a97e6360409a0e839f91ba97054d55c
```

And **b** receives the bitcoin part of the exchange:

```
~/git/swapbill $ python Client.py --dataDir b get_balance
...
in memory: RevealPendingPaymentSecret
In memory state updated to end of block 279622
Operation successful
balance : 3.5
```

Other information:

CHAPTER 15

Links and Resources

The SwapBill documentation can be found here: http://swapbill.readthedocs.org

The original bitcointalk announcement thread can be found here

The SwapBill source code lives here: https://github.com/crispweed/swapbill

Twitter: @swapbill Official Subreddit: http://www.reddit.com/r/swapbill/