



RPC App on Bitcoin



@DG Lab - Anditto Heristyo



Purpose

- First look at interacting with the Bitcoin RPC.
- Particularly useful if you're planning to develop your own apps on top of Bitcoin Core.

In this session

- Basic overview of Bitcoin as it relates the corresponding RPC commands.
- Examine a simple modification to the RPC that you can try out & get a glimpse of the RPC source code.
- Introduce an example app that incorporates business logic with a Bitcoin node backend.

Before all of that, though...

- Did you download the Bitcoin source code?
- Did you get it to compile?
- If you are still having problems, it'll be difficult to follow along for this session (or trying it out at home).

If you're having trouble, you can contact any of the trainers for help.

Bitcoin Overview & relation to the RPC

The 3 Networks

3 Networks

1. Mainnet

```
$ ./bitcoin-cli <Command>
```

Caution

Mainnet deals with real money, so be very careful.

3 Networks

2. Testnet

```
$ ./bitcoin-cli -testnet <Command>
```

- Generally same features as Mainnet.
- Mining difficulty is greatly reduced.
- The BTC here has 0 value. (which makes it valuable)
- Breaks sometime.

3 Networks

3. Regtest

```
$ ./bitcoin-cli -regtest <Command>
```

- Only on your local environment.
- You have to mine blocks yourself.

For convenience all examples in this session use `-regtest`

bitcoin.conf

https://en.bitcoin.it/wiki/Running_Bitcoin

Helpful ones when you're messing around:

regtest=1 OR **testnet=1**

rpcuser=user

rpcpassword=password #Dont forget to change

bitcoin.conf

```
printtoconsole=1 #Nice viewable details
```

If you get a CONNECTION REFUSED error:

```
rpcport=8332
```

```
port=8333
```

```
server=1
```

Sample bitcoin.conf used today

```
printtoconsole=1  
server=1  
regtest=1  
rpcuser=user  
rpcpassword=password  
rpcport=8332  
port=8333
```

Let's begin!

First, open the terminal and run `bitcoind`

```
$ cd ~/<Bitcoin Folder>/src
```

```
$ ./bitcoind
```

Or:

```
$ ./bitcoind -daemon
```

First Outputs

- Creation of Wallet (Private Keys)
- Connecting to the network
- Updating block information

Quick Check

In a different window:

```
$ ./bitcoin-cli getblockchaininfo
```

```
{
  "chain": "regtest",
  "blocks": 0,
  "headers": 0,
  "bestblockhash": "0f9188f13cb7b2c71f2a335e3a4fc328bf5beb436012afca590b1a11466e2206",
  "difficulty": 4.656542373906925e-10,
  "mediantime": 1296688602,
  "verificationprogress": 1,
  "chainwork": "0000000000000000000000000000000000000000000000000000000000000002",
  "pruned": false,
  "softforks": [
    {
      "id": "bip34",
      "version": 2,
      "reject": {
```

bitcoin-cli Commands

```
$ ./bitcoin-cli help
```

```
$ ./bitcoin-cli help <Command>
```

https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_calls_list

Warning:

“account” related things in the wallet are deprecated & buggy.

Alternate method

[https://en.bitcoin.it/wiki/API_reference_\(JSON-RPC\)](https://en.bitcoin.it/wiki/API_reference_(JSON-RPC))

For example on OSX:

```
$ open /Applications/Bitcoin-Qt.app
```

```
$ curl --user user --data-binary  
'{"id": "t0", "method": "getblockchaininfo",  
"params": [] }' http://127.0.0.1:8332/  
--verbose
```

Mining

Mining

In Bitcoin regtest network, you have to mine by yourself:

```
$ ./bitcoin-cli generate 1
```

Real Mining on Mainnet

Application Specific Integrated Circuit

Completely different story, which we won't cover here.

You can check the current mining info:

```
$ ./bitcoin-cli getmininginfo
```



Antminer S9

<https://www.bitmain.com/>

Wallet

Wallet

Bitcoin is an open system, and thus the management of your keys is a very important security point.

Important

If you don't control the keys, it's not your money!

Common pattern

A user exchanges USD to BTC on an Exchange, and the private keys for that BTC are kept there.



Various incidents: MtGox, Bitfinex, etc.

Your own bitcoind wallet

- Make sure you're on a computer free from virus, malware, keyloggers, etc.
- Encrypt your HDD.
- Encrypt your Wallet:
 - `$./bitcoin-cli encryptwallet "PASSPHRASE"`
 - `$./bitcoin-cli walletpassphrase "PASSPHRASE" TIMEOUT`

If you're trying out the RPC App later, don't encrypt your Regtest wallet before running it.

Let's try to send some BTC

First check how much BTC you have.

```
$ ./bitcoin-cli getbalance
```

What do you get?

Let's mine some more!

Block subsidy is usable after 100 blocks.

```
$ ./bitcoin-cli generate 100
```

Try checking the balance one more time:

```
$ ./bitcoin-cli getbalance
```

Now let's try to send some BTC

Create another address and send it to yourself.

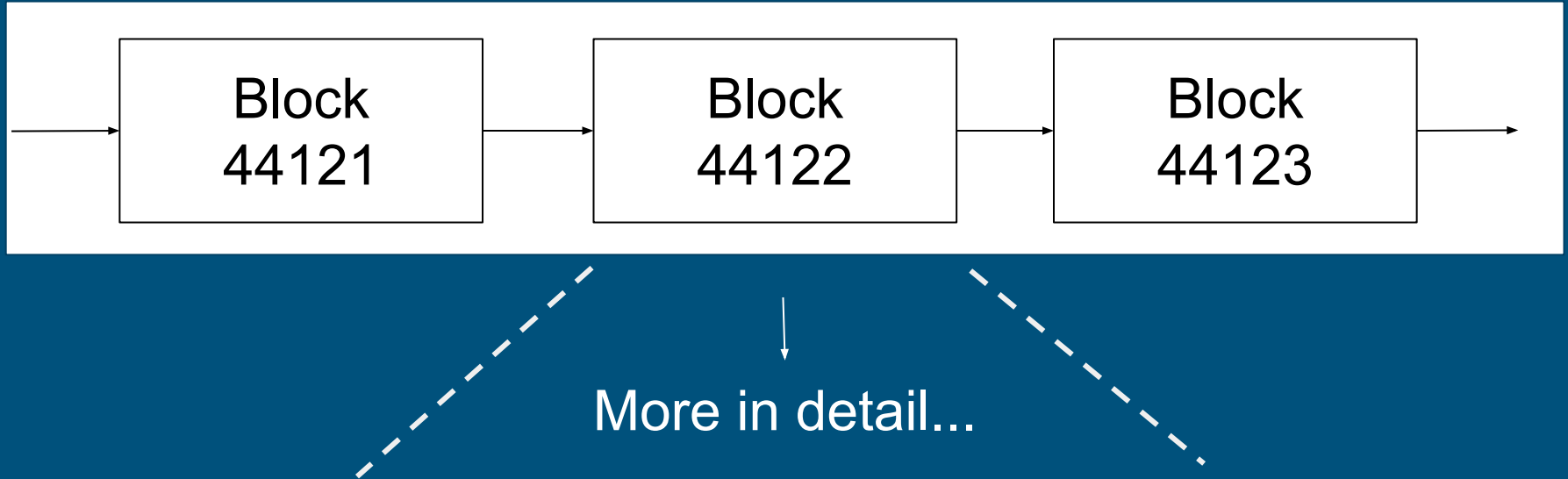
Simplest way to do it:

```
$ ./bitcoin-cli getnewaddress
```

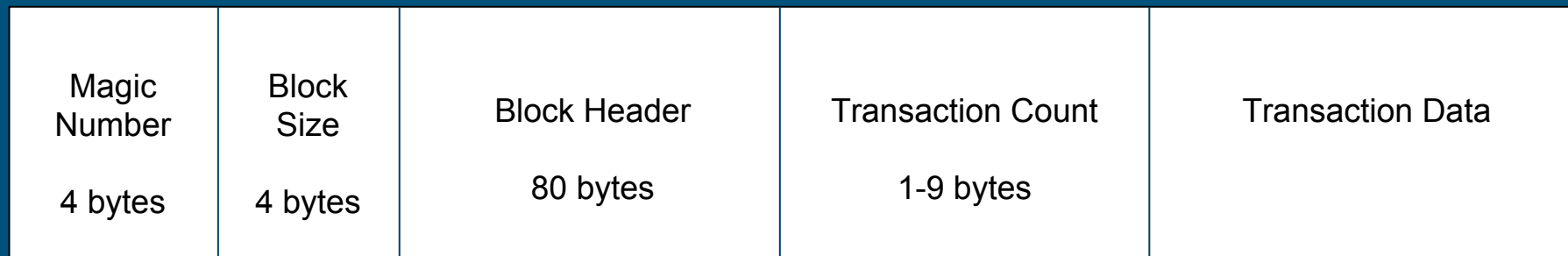
```
$ ./bitcoin-cli sendtoaddress <Address>  
<Amount>
```

Looking inside the blockchain

Blockchain



Inside a Block



The diagram illustrates the internal structure of a Bitcoin block. It is represented as a horizontal rectangle divided into five equal-width sections. Each section contains two lines of text: the top line is the field name, and the bottom line is the size in bytes. From left to right, the sections are: 'Magic Number' (4 bytes), 'Block Size' (4 bytes), 'Block Header' (80 bytes), 'Transaction Count' (1-9 bytes), and 'Transaction Data'. A solid blue line is positioned above the 'Magic Number' section. A dashed white line starts above the 'Block Size' section and extends towards the top right. Another dashed white line starts above the 'Transaction Data' section and extends towards the top right. A dashed white line starts below the 'Block Header' section and extends towards the bottom left. A solid white arrow points downwards from the 'Block Header' section to the text 'More in detail...'.

Magic Number	Block Size	Block Header	Transaction Count	Transaction Data
4 bytes	4 bytes	80 bytes	1-9 bytes	

More in detail...

```
$ ./bitcoin-cli getbestblockhash
```

```
$ ./bitcoin-cli getblock <OUTPUT>
```

Block Header

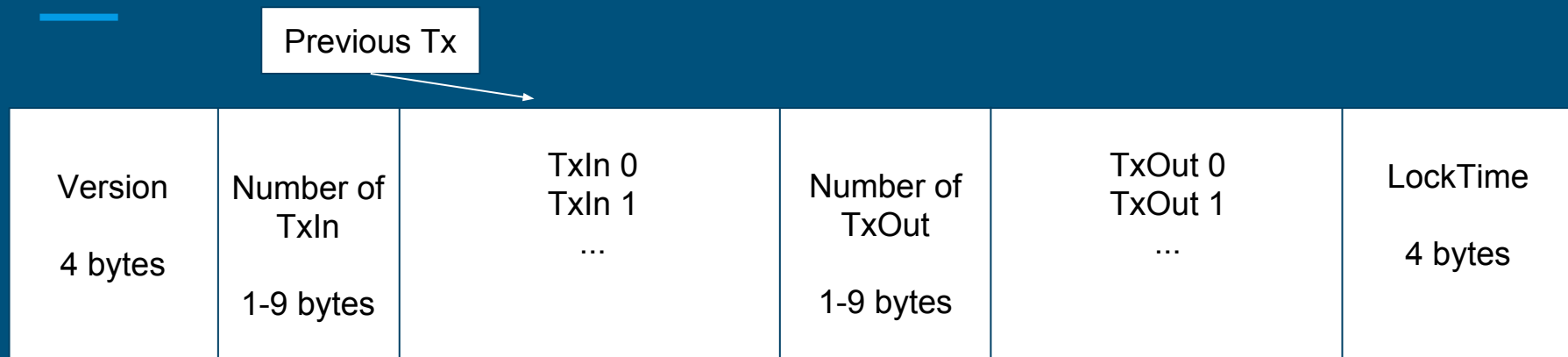


	hashPrevBlock	hashMerkleRoot		Bits	Nonce
Version	256-bit Hash	MerkleRoot of Transaction data	Time stamp	Difficulty	32-bit number
4 bytes	32 bytes	32 bytes	4 bytes	4 bytes	4 bytes

```
$ ./bitcoin-cli getbestblockhash
```

```
$ ./bitcoin-cli getblockheader <OUTPUT>
```

Inside a Transaction



```
$ ./bitcoin-cli gettransaction <TXID>
```

```
$ ./bitcoin-cli getrawtransaction <TXID>
```

```
$ ./bitcoin-cli getrawtransaction <TXID> 1
```


Address

Bitcoin Address

Generally on Mainnet:

- P2PKH (Pay to Public Key Hash)
- P2SH (Pay to Script Hash)

Bitcoin Address

```
$ ./bitcoin-cli getnewaddress
```

```
$ ./bitcoin-cli sendtoaddress <Address> <Amount>
```

Check money coming in to that address:

```
$ ./bitcoin-cli getreceivedbyaddress <Address>
```

```
$ ./bitcoin-cli getreceivedbyaddress <Address>  
<min. Confirmations>
```

Bitcoin Address

NEW! Segwit supported addresses

- P2SH-P2WPKH (Pay to Witness Public Key Hash)
- P2SH-P2WSH (Pay to Witness Script Hash)

Bitcoin Address

```
$ ./bitcoin-cli getnewaddress
```

```
$ ./bitcoin-cli addwitnessaddress <Address>
```

What happened? On regtest, Segwit is activated after 432 blocks.

```
$ ./bitcoin-cli generate 432
```

Important Point - Address Re-use

An Address should only be used once!

Run `getnewaddress` every time, as the previous example.

Problems: anonymity, security, etc.

Connecting Nodes

Running Multiple Nodes

Just run `./bitcoind` with a different `-datadir` containing a different `bitcoin.conf` with different parameters, particularly the `rpcport` and `port`.

```
$ ./bitcoind -datadir=/<SOME_DIR>/
```


Sample bitcoin.conf for Node #2

```
server=1  
regtest=1  
rpcuser=user  
rpcpassword=password  
rpcport=10000  
port=10001
```

Adding the node

Connecting them is simply running addnode from the 2nd node:

```
$ ./bitcoin-cli -datadir=/<SOME_DIR>/ addnode  
"127.0.0.1:8333" "onetry"
```

You can check it works:

```
$ ./bitcoin-cli -datadir=../ getconnectioncount  
1
```

The RPC Source Code

Bitcoin RPC - Overview

• bitcoin-cli.cpp	Command Line Interface application
• rpc/blockchain.(cpp h)	Block-related RPC commands
• rpc/client.(cpp h)	Utilities, helper
• rpc/mining.(cpp h)	Mining-related RPC commands
• rpc/misc.cpp	Miscellaneous RPC commands
• rpc/net.cpp	Net related RPC commands
• rpc/protocol.(cpp h)	Auto/JSON request/reply etc.
• rpc/rawtransaction.cpp	Tx-related RPC commands
• rpc/register.h	Registering RPC commands
• rpc/server.(cpp h)	RPC server functions
• httprpc.(cpp h)	HTTP RPC server

Bitcoin RPC - Other classes

- amount.h CAmount(satoshi)
- uint256.(cpp | h) Hash, etc. class
- arith_uint256.(cpp | h) uint256 with math features

Before touching any code

It's best to create your own branch and work on that.

```
$ git checkout -b Name-Purpose
```

For example:

```
$ git checkout -b anditto-rpc
```

Debugging

Debug: restart bitcoind

Whenever the code is changed, you need to restart bitcoind.

```
$ ./bitcoind -printtoconsole  
[...]
```

^C (Ctrl + C) stops bitcoind. Then run `make & ./bitcoind -printtoconsole` **once more.**

Hints for debugging

Mac & Linux can use a debugger.

Mac user

```
$ lldb bitcoind (bitcoin-cli, ...)
```

Linux user

```
$ gdb bitcoind (...)
```

Commands are slightly different, so check accordingly.

Debug: lldb / gdb

lldb	gdb	Result
b <file>:<line>	break <file>:<line>	Stop at <line> in <file>
b <function>	break <function>	Stops when entering <function>
bt	bt	Displays the stack frame
up, down	up, down	Move within the stack frame
p <var>	p <var>	print out <var>

Comparison of commands: <http://lldb.lvm.org/lldb-gdb.html>

Debug: `--enable-debug`

Sometimes Lldb & gdb doesn't show the contents of the variable.

Usually running this fixes it:

```
$ ./configure --enable-debug
```

```
$ make clean
```

```
$ make
```

RPC

As you recall, list of usable commands:

```
$ ./bitcoin-cli help
```

Now let's insert a new command!

A Simple Example

Let's add a command

When we run this command:

```
$ ./bitcoin-cli print "sample value"
```

We want this output:

```
sample value
```

Let's add a command

```
$ ./bitcoin-cli print "hello"
```

```
hello
```

File: src/rpc/misc.cpp

Actually...

RPC already has a hidden 'echo' command, but let's make one with a simpler output:

- Just 1 argument
- Only the argument, not a JSON output

Hint

Check out `/rpc/misc.cpp` and see how the function 'echo' is constructed.

The RPC App

Basic Premise

From the previous session:

- Different paradigm compared to regular business apps.
- Some people are going to try to steal from you.
- Even in cases with no bad actors, mistakes are costly & irreversible.

Purpose

- To get you thinking of several different scenarios where things can go wrong.
- Something you can bring home and try out.
- Based on Node.js but should be simple enough to understand and recreate in other languages.

What you will need

- The code: <https://github.com/dgarage/bitcoinrpcapp>
- You can follow along the README for installing.
- Requires: Node.js, Mongodb, Mocha (test framework)

Environment Settings

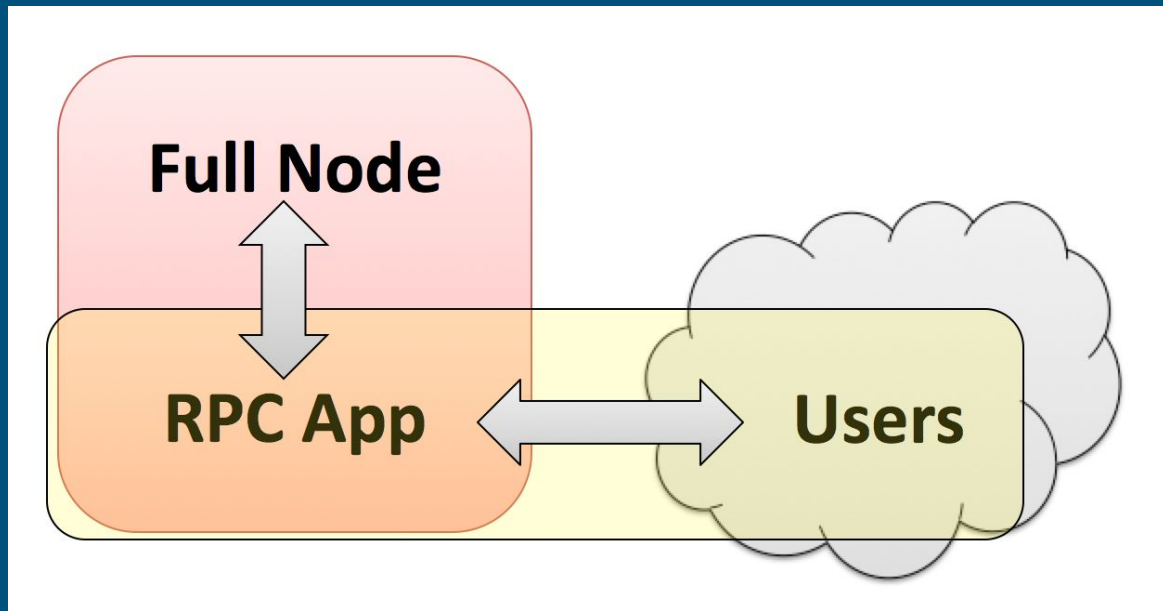
RPC Access

RPC access = **Unlimited Access** to your wallet!

Giving Access = **Handing over your Wallet!**

It's not as simple as: "Let me build an app, publish on the App Store, and have it connect through RPC.."

A safer setup for an RPC App



Correct setup for an RPC App

- ① Setup a safe Full Node.
- ② Create an RPC App that connects to that Node, and place it on the same server.
- ③ Build into the RPC App something that connects outside of the server.
- ④ Users connect through an app that talks to ③.

Correct setup for an RPC App

- ① Setup a safe Full Node.
- ② Create an RPC App that connects to that Node, and place it on the same server.
- ③ Build into the RPC App something that connects outside of the server.
- ④ Users connect through an app that talks to ③.

Restrictions on RPC usage

- By default, access outside of `localhost` is blocked.
- Setting authentication for RPC has 2 different ways:
 - ① Inside the settings file (`bitcoin.conf`)
setup the directory: `-datadir=X`
 - ② Set directly: `-rpcuser=Y -rpcpassword=Z`

RPC App

For this instance, inside the bitcoin.conf file:

```
rpcuser=user  
rpcpassword=password
```

Warning: For this demo we use 「password」, but in other situations please **use a safe password**.

RPC App

Once config is saved, you need to restart bitcoind.

① On the window running `./bitcoind` press **^C** (Ctrl+C).

If you can't find it, `killall -9 bitcoind` also shutdown all bitcoind.

② Run once more: `./bitcoind`

RPC App

A CLI-only simple system.

Features:

- Creating an invoice.
- Confirm payments.
- Protect against Double Spending, re-orgs, etc.

RPC App

- node.js
- Uses a bitcoin-RPC wrapper (bcrpc)
- Uses a bitcoin test suite (bitcointest)

RPC App

Install the necessary libraries & try running it:

```
$ cd <Your_bitcoin_folder>/<RPC_App>
```

```
$ npm install
```

```
$ ./rpc-cli
```

Available commands:

```
[...]
```


Running the tests

Run the test suite:

```
$ npm test
> rpc-cli@1.0.0 test ../bc2/conference/rpc-cli
> mocha tests.js
  db
    ✓ can be connected to
  bitcoind
    ✓ can be found
  bitcointest
    ✓ configures
```

...

The RPC App

Invoice

The company (through the app) issues an invoice, with the following information:

- invoice number
- bitcoin address (people pay into here)
- amount (amount you want paid)
- content (reason for payment, item description, etc.)

Invoice

More concretely:

- ① Get a new address with `bitcoin.getNewAddress`
- ② Insert the `addr`, `amount`, `content`, into the invoice in db.

Recording Payment

When someone sends bitcoin to the address (addr) attached to the invoice, that Tx is recorded.

This is saved as a payment in the DB.

```
db.payment.insert({txid, addr, invoice, amount})
```

If that txid exists, updates the existing record.

Recording Payment

Any changes to Payments (create/update), the status of Invoice is updated (unpaid, partially paid, etc.)

- `if sum(payment.amount) == 0 → unpaid`
- `if sum(payment.amount) == invoice.amount → paid`
- `if sum(payment.amount) < invoice.amount → partial`
- `if sum(payment.amount) > invoice.amount → overpaid`
- ...

RPC App

Simply put:

- ① Append bitcoin address (addr) to invoice
- ② When you find the Tx with the address of the recipient, you record it as a payment.
- ③ When `sum(payment.amount) = invoice.amount` it's deemed as paid.

RPC App

There are still several problems with the current interpretation.

Take a couple seconds to think about these problems.
(With the previous session in mind.)

Double Spending

One problem is Double Spending. One scenario:

- ① Create a Tx, with output made to the invoice addr.
- ② Copy the Tx, but with output to own wallet.

First sent ① to the target, immediately when they confirm it, send ② to other nodes. In the RPC App the status is changed to Paid on ① & ignores ②.

Double Spending

A different way Double Spend.

- ① Create Tx, out=addr, amount=invoice.amount/2
- ② Copy the Tx, change the hex

Send ①, then send ② when confirmed.

In the app it will register as paid `invoice.amount/2` twice.

RPC App - Problems

There are other problems, but basically the judgement for 'confirmed' and 'unconfirmed' status during development is important. For example we can set:

Pending = not yet verified = (for example) less than 5 blocks

Confirmed = verified = (for example) 6 blocks and above

→ paid, pending_paid, partial, pending_partial, ...

In Conclusion

Have fun, play around with the code, but stay safe :)



@ DG Lab - Anditto Heristyo

Twitter: @anditto_h

Appendix

Sample solution for adding 'print' function

Sample solution (1 / 4)

- ① Put a new function at an appropriate place. (like after the 'echo' function).

```
UniValue print(const JSONRPCRequest& request)
{
}
```


Sample solution (2 / 4)

② Put in `print` in the `commands[]` inside the file.

```
{  "util",    "print",    &print,    {"text"}  },
```

With this, compile one more time, and run `bitcoind`. Try the command:

```
./bitcoin-cli print "sample value"
```

Sample solution (3 / 4)

③ First, insert the following in place from step ①:

```
if (request.fHelp || request.params.size() != 1)
    throw std::runtime_error(
        "print \"message\"\\n"
    );
```

Sample solution (4 / 4)

④ Lastly, insert code for console output:

```
if (request.fHelp || request.params.size() != 1)
    throw std::runtime_error(
        "print \"message\"\\n"
    );
return request.params[0];
```

Compile one more time. Did it work?

RPC App

Task 1

Task 1

Where : `payments.js`, line 25 (`createInvoice`)

Content : **creating** `model.invoice`

Details : Create the missing functions (more precisely, get a bitcoin address, and put it into the invoice DB)

Task 1

After finishing Task 1, the rpc-cli can create an invoice.

```
$ ./rpc-cli create 5 "Socks"
```

```
$ ./rpc-cli list
```

invoice:	amount:	status:	conf:	pending:
587ca8535a1e4536248f5df9	5	???	0 BTC	0 BTC

But as of now, `rpc-cli info 587ca...` is not available.

Task 1 Sample Solution

Task 1 (1/2)

First, get a new Bitcoin address.

```
bitcoin.getNewAddress( (err, response) => {  
    if (err) return cb(err);  
    // ...  
});
```

We pass along errors to cb.

Task 1 (1/2)

2nd way to get a new Bitcoin address.

```
const addr = bitcoin.getNewAddressS().result;
```

This is also fine but,

```
try ... catch (e) { cb(e); }
```

is necessary.

Task 1 (2/2)

After getting an address, call **model.invoice.create**, and pass the result to cb.

```
bitcoin.getNewAddress((err, response) => {  
    if (err) return cb(err);  
    const addr = response.result;  
    model.invoice.create(satoshi, content, addr,  
cb);  
});
```

Task 2

Task 2

Where : `payments.js`, line 100 (`updateInvoice`)

Content : handling payment

Details : when verifying payment,

**Calculate total/final/disabledAmount
(check the variables with those names)**

Hint 1

When a re-org happens, the following update to the payment is necessary:

```
model.payment.setStatus(payment._id, 'reorg', (err) => {  
  model.history.create(invoice._id, payment._id, {  
    action: 'reorg',  
    ...  
  })  
})
```

Hint 2

Whether or not a reorg has occurred can be observed by checking whether or not a Tx is in a block.

Thus, if `transaction.blockhash` returns null, that Tx is not inside the block!

Task 2 Sample Solution

Task 2

What we want to do here:

- ① Verify a transaction attached to a payment
- ② Add up the totalAmount
- ③ Decide if it's verified or not, and add the finalAmount
- ④ Check the lowest confirmations (number of blocks)

Task 2

Is the Tx OK?

- Retrieve the Tx from the Node

① You get an Error → Can't find the Tx → **There's a problem**

② You can't find the blockhash → Tx isn't in the block → Possibly a re-org happened → possible Double Spend → **There's a problem**

Task 2

① is already in the code, so you don't need to do anything.

If ② happens, we change the `model.payment` status to 'reorg' and send to next payment (`asyncCallback`).

We don't want to leave that in the records, so using **`model.history.create`** we put in an action called **reorg**.

Task 2 (1/4)

```
if (!blockhash) {  
    //reorg happened  
    disabledAmount += amountSatoshi;  
    if (payment.status === 'reorg') {  
        //nothing is done since we already know  
        return asyncCallback();  
    }  
    // ...
```

Task 2 (2/4)

In the case re-org happened (continued):

```
model.payment.setStatus(payment._id, 'reorg', (err) => {  
  model.history.create(invoiceId, payment._id, {  
    action: 'reorg',  
    invoiceId: invoiceId,  
    paymentId: payment._id,  
  }, 'payment went through a reorg', asyncCallback);  
});  
return;  
}
```

Task 2 (3/4)

When the payment is OK, is put into the calculations.

```
    const isFinal = confirmations >=
config.requiredConfirmations;
    if (minConfirms > confirmations) minConfirms =
confirmations;
    if (maxConfirms < confirmations) maxConfirms =
confirmations;
```

Task 2 (4/4)

```
totalAmount += amountSatoshi;  
if (isFinal) finalAmount += amountSatoshi;  
  
model.payment.setStatus(  
    payment._id,  
    isFinal ? 'confirmed' : 'pending',  
    asyncCallback);
```

Task 3

Task 3

Where : `payments.js`, line 140 (`updateInvoice`)

Content : updating invoice

Details : verify and calculate the payment info, and update the invoice.

Hint

Check for over- (or under-) paying, using

```
Math.abs (paid_amount - invoiced_amount) <  
config.thresholdSatoshi
```

Task 3

If you complete Task 3, all the tests should be passing!

Task 3 Sample Solution

Task 3

Here we want to do 1 thing:

Determine the status of the invoice.

Here we have 7 status.

Task 3

The 7 Status

Case	Verified	Unverified
Not yet paid	unpaid	---
Paid, but insufficient	partial	pending_partial
Paid, but overpaid	overpaid	pending_overpaid
Paid	paid	pending_paid

Task 3

- To determine payment confirmation status, we need to set a minimum confirmation number.
- It's easy if we have a flag indicating whether a payment with exactly the correct amount has occurred or not.
- Also good to separate confirmed and unconfirmed payments.

Task 3 (1/3)

Several variables that we need on the top:

```
const confirmations = Math.min(minConfirms, maxConfirms);  
const pendingAmount = totalAmount - finalAmount;  
const threshold = config.thresholdSatoshi;  
const finalRem = invoice.amount - finalAmount;  
const totalRem = invoice.amount - totalAmount;  
const finalMatch = Math.abs(finalRem) < threshold;  
const totalMatch = Math.abs(totalRem) < threshold;
```


Task 3 (2/3)

On the bottom:

Put a helper variable

```
let status;
```

Task 3 (3/3)

The 7 status (example):

```
if (finalMatch && totalMatch)
else if (totalAmount === 0)
else if (finalRem < -threshold)
else if (totalRem < -threshold)
else if (totalMatch)
else if (totalAmount === finalAmount)
else

status = 'paid';
status = 'unpaid';
status = 'overpaid';
status = 'pending_overpaid';
status = 'pending_paid';
status = 'partial';
status = 'pending_partial';

model.invoice.updateStatus(invoiceId, status, cbwrap);
```

Beyond Tasks 1~3

About bitcointest

bitcointest creates and destroys nodes.

Usually it's fine, but sometimes some process remains.

When the bitcointest is acting strange, sometimes this works

```
$ killall -9 bitcoind
```

About bitcointest

... The previous commands kills all bitcoind process.
Stopping individual bitcointest process use:

```
$ ps -A | grep bitcoind
```

which produce a list, where you can see the PID and pass it to `kill -9` (for example here, **11453**).

```
11453 ... ../src//bitcoind -regtest  
-datadir=/tmp/bitcointest//22011 -rpcuser=user...
```

Other hints

`npm test` goes through the file `tests.js`, so you can check out the source code and find out what it's expecting.

Running this only executes Task 1: **TASK=1** `npm test`

Which is also the same for Task 2 & 3.

Other hints

V=1 `npm test` gives you more information in the outputs.

DEBUG=bc2 `npm test` shows `debug()` for the RPC App.

Both can be used at the same time:

```
$ V=1 DEBUG=bc2 npm test
```

When you want to reset the DB (**erases everything!**):

```
$ mongo --eval "db.dropDatabase() "
```