Started May, 2011

libbitcoin

*ohloh.net analysis:*

64,115 lines

*Estimated Effort:*

16 person-years

*Estimated Cost:*

$864,843

/libbitcoin.dyne.org

github.com/spesmilo/libbitcoin 991 commits
libbitcoin is an asynchronous library.

It operates with components called 'services'.

Services take a threadpool as their first argument.

```python
threadpool disk_pool(4); // 4 threads spawned
leveldb_blockchain chain(disk_pool);
```

The dependencies for a service follow the threadpool.

```python
threadpool memop_pool(1); // 1 thread spawned.
transaction_pool txp(memop_pool, chain);
```

Currying is fundamental to libbitcoin.

Currying takes a function and changes its signature.

This is how we make libbitcoin asynchronous, safe, and modular.

Old approach:

```python
class Foo_Widget:
    def on_click(self, event):
        ...
        def ...
```
Old approach:

class IRCBot:

class IrcBot:

    def on_connect(self, event) -> None:
        # join a channel
        ...

    def on_join(self, self, ...) -> None:
        pass

    def on_receive_message(self, self, channel, user, message) -> None:
        # do something
        send(reply)

Problems:

INFLEXIBLE DESIGN

- You must use their class layouts.
- Their flavour of OOP.
- Keep track of temporary variables between method calls (somehow).
- Synchronise shared values.

URBSN CENTER

HAEGUS

HAEGO
What is Currying?

Currying transforms function signatures.

```c
void f(a, b, c, d)
g = f(120, -2, -1, "hello")
g = bind(f, 120, -2, -1, "hello")
```

In C++

```c++
#include <functional>
using std::placeholders::_1;
using std::placeholders::_2;

void f(int a, Object b, int c, string d);
g = std::bind(f, 120, _2, _1, "hello");
g(4, foo);
```

- Connect functions of different signatures to each other.
- Keep least temporary variables as bounded arguments (in bind).
```cpp
#include <bitcoin/bitcoin.hpp>
using namespace bc;
bool stepped = false;

void my_function()
{
    std::cout << "Hello World!!!" << std::endl;
}

int main()
{
    threadpool pool(8); // 8 threads
    pool.dispatch(my_function);
    while (!stepped)
    {
        sleep(0.5);
    }
    pool.stop(); // stop for pool shutdown
    pool.join(); // join running threads and wait for them to finish.
    return 0;
}
#include <bitcoin/bitcoin.hpp>
using namespace bc;

class stupid_example
{
  public:
    stupid_example (threadpool& pool) : strand_ (pool) {} //

    void foo_add (int v) //
    {
      strand_.queue ( //
        [this, v] //
          { foo_ += v; }); //

    void foo_increment () //
    {
      strand_.queue ( //
        [this] //
          { ++foo_; }); //

    private:
      async_strand strand_; //

      int foo_ = 0; //
};
int main()
{
    threadpool pool(2);
    stupid_example example(pool); // But it's an example nonetheless!
    // Returns immediately
    example.foo_add(10);
    // Returns immediately
    example.foo_incr();

    std::cout << "Press enter to shut down." << std::endl;
    std::cin.get();
    pool.stop();
    pool.join();
    return 0;
}
libbitcoin operations take a handler (first argument)

```cpp
def do_something( arguments..., handler):
    void handler
    do something
    then call this
```

```cpp
void handler( const std::error_code& ec, arguments...)
{
  
  error code as first argument,
  handlers differ depending
  on the different operations

  std::error_code ec = bc::error::bad_stream;
  if (ec == bc::error::bad_stream)
  {  // handle bad_stream errors.
    else if (ec)
    {  // handle all other errors.
      else
        // main body
    }
}
```
my usual handler looks like:

void keo_something_happened (std::error_code ec, ...)
{
    if (ec)
    {
        std::cerr << "app: Something failed to happen:"
                   << ec.message() << std::endl;
        return;
    }

    // do stuff...
WHIRLWIND TOUR

Services:

blockchain (pluggable backends -
default is leveldb.
deprecated versions: bdb, postgresql)

transaction-pool
transaction-indexer <- look up transactions by
                      address,
                      node remains in sync with pool.

network, acceptor, channel <-
                            connect and accept connections.

protocol <- p2p network. manages connections, seeding, broadcasting...
hosts <- list of hosts

handshake <- initial connection handshake (exchange version messages)
poller <- poll network for new blocks.
getx-responder
utilities and types:
- payment_address < encoding and decoding of Bitcoin addresses
- script_type < bitcoin script
- transaction_type, block_type, ...
- base58, ripemd, sha256, mnneys, ...
- Serialization using iterators, buffers preallocated:
  data_chunk: rawtx(satoshi:raw_size(tx));
  auto end_iter = satoshi:save(tx, rawtx:begin_c());
  BITCOIN_ASSUME(end_iter == rawtx:end());
  encode_hex, decode_hex(str), satoshi_to_btc(satx);
- magic numbers in method <bitcoin/constats.hpp>
- <bitcoin/block.hpp>, <bitcoin/transaction.hpp>
  hash_digest, hash_block_header(block_header)
  hash_digest, hash_transaction(tx)
  block_type, genesis: block();
- elliptic_curve_key:
  new privkey, get/set privkey
  get/set pubkey
  sign/verify
- deterministic_wallet < electron
  new seed, get/set seed
  get/set mpk, gmpk, generate pubkey
  generate privkey (secret)

Validation of blocks and unconfirmed transactions
is in <bitcoin/validate.hpp>
Focus:

- Scalability
- Intuitive
- Extendable
- Never block
- We ♥ UNIX design

Design:

#1 Simplicity (of implementation)
#2 Correctness (good design)
#3 Consistent (but not if we sacrifice #1 or #2)
#4 Completeness (be practical though)

FRAMEWORK BAD
TOOLKIT GOOD

http://libbitcoin.dyne.org/doc
libbitcoin/examples/fullnode.cpp
obelisk

github.com/spesmilo/obelisk
Obelisk

blockchain server infrastructure:

clients use libobelisk:

```cpp
#include <bitcoin/bitcoin.hpp>
#include <obelisk/obelisk.hpp>

threadpool pool(1);
Obelisk::fullnode_interface fullnode(pool, "tcp://localhost:9091");
fullnode.address().fetch_history(
  address, history_fetched_handler);
```
libbitcoin fullnode example (300 lines of code)

Apache Thrift

"framework for scalable cross-language services development"
~ its website

blockchain daemon with network interface. (github.com/genjix/query)

Apache Thrift:
- Made by Facebook.
- We don't like frameworks.
- Scalable? Don't lie.
"Simplest Way to Connect Pieces"
~ zeromq.org

➔ Click 'Learn'
➔ 'the guide'
Chapter 1 - Fixing the world

"We can leave the political philosophy
for another book."
➔ softwareandsilicon.com

The ZeroMQ library author, Pieter Hintjens,
is a genius.

hintjens.com/blog:17
ZeroMQ =
A Few Basic Building Blocks

REQ → REP

PUB

SUB → SUB → SUB

VENTILATOR

PUSH

PULL

tasks

PUSH

PUSH

PULL

results

WORKERS

SINK

DEALER

DEALER

DEALER

DEALER

ROUTER

ROUTER

load balancer

DEALER

DEALER

DEALER

clients

workers
BLOCKCHAIN WORKERS

BITCOIN NODES.
MORE POWER TO DEVELOPERS...

Sx:
AND ADMINs!

Give people the building blocks and they will make stuff.

Sx possibilities:
- offline transactions.
- multisignature.
- QR codes.
- deterministic wallets.
- embed file hashes in blockchain.
- commands for querying Obelisk blockchain, working with transactions (show, validate, broadcast)
- many possibilities.
- exploratory prototype: ncurses terminal wallet < ncurses.
$ wget http://sx.dyne.org/install-sx.sh
$ sudo bash install-sx.sh

OR

$ bash install-sx.sh INSTALLPREFIX/

http://sx.dyne.org

$ sx help

✓ list of commands

$ sx help COMMAND

sx COMMAND [ARGS]...