Boost.Blockchain

A new business model for open source

Arthur O'Dwyer
2019-05-08

Monetizing your applications is hard enough.

- SaaS?
- Paid support?
- Paid “pro” features?
- Donations?

Monetizing your ***library*** is even harder.

- Many C++ libraries are “header-only”
- Source code must be available
- Difficult to tell who’s downloading and using your code

There’s got to be a better way!

Enter Boost.Blockchain

- Harnesses the power of upcoming C++26 features
- Constexpr, consteval, Ranges, Networking.TS

Enter Boost.Blockchain

- Harnesses the power of upcoming C++26 features
- Constexpr, consteval, Ranges, Networking.TS
- Compile-time networking
- Get paid in Bitcoin when people compile your code!

It’s simple to use

```
#ifndef YOUR_HEADER
#define YOUR_HEADER

#define BOOST_BLKCHN_ADDRESS \
    1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2
#define BOOST_BLKCHN_FEE \
    (100*boost::blockchain::units::satoshi)

#include <boost/blockchain/monetize.hpp>

// your content goes here

#endif // YOUR_HEADER
```

How does it work?

```
#include <boost/blockchain/miner.hpp>

namespace boost::blockchain::monetize {

    constexpr boost::constasio::io_service ios;

    static inline constexpr bool monetized =
        blockchain::miner(BOOST_BLKCHN_ADDRESS, ios)
            .run(blockchain::monetize::time_to(BOOST_BLKCHN_FEE))
        >> true;

    static_assert(monetized);

}
```

How does it work?

```
#include <boost/blockchain/history.hpp>

namespace boost::blockchain::monetize {

    consteval auto time_to(units::crypto desired_fee) {
```

How does it work?

```
#include <boost/blockchain/history.hpp>

namespace boost::blockchain::monetize {

    consteval auto time_to(units::crypto desired_fee) {
      namespace rv = std::ranges::view;
      namespace ra = std::ranges::action;
```

How does it work?

```
#include <boost/blockchain/history.hpp>

namespace boost::blockchain::monetize {

    consteval auto time_to(units::crypto desired_fee) {
      namespace rv = std::ranges::view;
      namespace ra = std::ranges::action;
      auto hist = history(ios).fetch();
```

How does it work?

```
#include <boost/blockchain/history.hpp>

namespace boost::blockchain::monetize {

    consteval auto time_to(units::crypto desired_fee) {
      namespace rv = std::ranges::view;
      namespace ra = std::ranges::action;
      auto hist = history(ios).fetch();
```

Why not = `co_await constexpr history(ios).fetch()` ?

How does it work?

```
#include <boost/blockchain/history.hpp>

namespace boost::blockchain::monetize {

    consteval auto time_to(units::crypto desired_fee) {
      namespace rv = std::ranges::view;
      namespace ra = std::ranges::action;
      auto hist = history(ios).fetch();
```

Why not `= co_await constexpr history(ios).fetch()` ?

Well, because then `time_to` would have to be a coroutine.

How does it work?

```
#include <boost/blockchain/history.hpp>

namespace boost::blockchain::monetize {

    consteval auto time_to(units::crypto desired_fee) {
      namespace rv = std::ranges::view;
      namespace ra = std::ranges::action;
      auto hist = history(ios).fetch();
      auto r = hist.transactions() | rv::reverse | rv::take(100);
      auto fees = r | rv::transform(&transaction::fee);
```

How does it work?

```cpp
#include <boost/blockchain/history.hpp>

namespace boost::blockchain::monetize {

    consteval auto time_to(units::crypto desired_fee) {
      namespace rv = std::ranges::view;
      namespace ra = std::ranges::action;
      auto hist = history(ios).fetch();
      auto r = hist.transactions() | rv::reverse | rv::take(100);
      auto fees = r | rv::transform(&transaction::fee);
      auto mean = (fees | ra::accumulate(0*units::satoshi))
                  / std::ranges::distance(r);
```

How does it work?

```
#include <boost/blockchain/history.hpp>

namespace boost::blockchain::monetize {

    consteval auto time_to(units::crypto desired_fee) {
      namespace rv = std::ranges::view;
      namespace ra = std::ranges::action;
      auto hist = history(ios).fetch();
      auto r = hist.transactions() | rv::reverse | rv::take(100);
      auto fees = r | rv::transform(&transaction::fee);
      auto mean = (fees | ra::accumulate(0*units::satoshi))
                  / std::ranges::distance(r);
      return (desired_fee / mean) *
             (r | ra::iter_distance(&transaction::timestamp));
    }
```

Concerns

Concerns

- Could Modules defeat our business model?
- “Pay once, build anywhere”?
- I’ll believe it when I see it.

Concerns

- “What if the user just edits `my_header.h` to remove the monetization code?”

Concerns

- “What if the user just edits `my_header.h` to remove the monetization code?”
- You need a better class of users!

Concerns

- Consider encrypting your actual C++ code with a key stored in the blockchain.
- When the compiler provides sufficient proof of work, it is granted access to the key.
- A `consteval` function can then decrypt the source code and continue compilation. (TODO: reflection and metaclasses?)

Future Directions

Future directions

- Running a bitcoin miner in the compiler is time-consuming, especially with distributed builds.

Consider moving the bitcoin miner into the build system instead.

Thank you

Please hodl your questions