MAYSTREET

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca

Deploying the Networking TS

```
Opening DataConn socket on 0.0.0.0:11653...
Started (send SIGINT or SIGTERM to exit)
2021-10-06T04:01:26.490694105Z
  Accepted connection 10.244.0.119:42972 => 0.0.0.0:11653
2021-10-06T04:01:26.490785365Z
  10.244.0.119:42972 => 0.0.0.0:11653 request: {"version":"v1","query":"p
2021-10-06T04:01:26.721512358Z
  10.244.0.119:42972 => 0.0.0.0:11653 accept: {"format":"application/jso
2021-10-06T04:01:26.870505319Z
  10.244.0.119:42972 => 0.0.0.0:11653 complete (1184 bytes)
2021-10-06T04:01:26.871665987Z
  10.244.0.119:42972 => 0.0.0.0:11653 disconnected due to failure reading
```

Scenario

System interacts with outside world via
- WebSockets
- HTTP/REST

Front-end selects back-end to handle query

Front-end and back-end communicate with separate protocol

Back-end implemented with Asio

Queries serviced by proprietary database back-end (written in C++)

DataConn

TCP protocol without login or encryption
    Intended solely for use on private network

Client (front-end) sends request

Server (back-end) accepts or rejects

Response streamed until final message or until client requests cancel
    Server acknowledges cancel

After response connection is ready for another request

Server sends heartbeats during inactivity

DataConn Message Structure

Offset	Field
0	Length (bytes) (little endian)
1	
2	
3	
4	Message type
...	Body (optional)

DataConn Messages (Client to Server)

Type	Name	Payload?
R	Request	✓
C	Cancel	

DataConn Messages (Server to Client)

Type	Name	Payload?	Terminal?
A	Acknowledge Request	✓	
J	Reject Request	✓	✓
I	Intermediate Chunk	✓	
F	Final Chunk	✓	✓
E	Error	✓	✓
K	Acknowledge Cancel		✓
H	Heartbeat		

Reading DataConn

```
enum class message_type { unknown, request, cancel, acknowledge_request,
  reject_request, intermediate_chunk, final_chunk, error, acknowledge_cancel,
  heartbeat
};

template<typename AsyncReadStream, typename DynamicBufferV2,
  typename CompletionToken>
decltype(auto) async_read(AsyncReadStream& stream, DynamicBufferV2 buffer,
  CompletionToken&& token);
```

Dynamic Buffer Requirements

```
struct /* ... */ {
  using const_buffers_type = /* ... */;
  using mutable_buffers_type = /* ... */;
  std::size_t size() const;
  std::size_t max_size() const;
  std::size_t capacity() const;
  const_buffers_type data(std::size_t pos, std::size_t n) const;
  mutable_buffers_type data(std::size_t pos, std::size_t n);
  void grow(std::size_t n);
  void shrink(std::size_t n);
  void consume(std::size_t n);
};
```

Limiting Incoming Message Size

First four bytes of a message give length of message

Malicious client could send any value, including extremely large value
    Maximum of 4 294 967 295 (4 GiB)

Naïve implementation could be induced to allocate extremely large buffer

Denial of service attack if `std::bad_alloc` is thrown or OOM killer activates

Must be able to limit amount of memory application is willing to allocate

`max_size` is promising but `grow` throws `std::length_error` if it's exceeded

```cpp
enum class async_read_error {
  // ...
  max_size_too_short_for_header,
  max_size_too_short_for_payload
};

std::error_code make_error_code(async_read_error) noexcept;

template<typename DynamicBufferV2>
std::error_code async_read_grow(DynamicBufferV2& buffer, std::size_t n,
  async_read_error err)
{
  const auto max = buffer.max_size();
  if ((n > max) || ((max - n) < buffer.size())) {
    return make_error_code(err);
  }
  buffer.grow(n);
  return {};
}
```

What's the Completion Signature?

```
void(std::error_code, std::size_t, message_type,
  typename DynamicBufferV2::const_buffers_type);
```

`ConstBufferSequence` (final parameter) contains payload of message

Could use `consume` to leave only payload in `DynamicBufferV2` but this would...

- ...require bytes to be shifted
- ...discard bytes read from the network

Writing DataConn

```
using async_write_state = std::array<byte, 5>;

//  Completion signature:
//  void(std::error_code, std::size_t);
template<typename AsyncWriteStream, typename ConstBufferSequence,
  typename CompletionToken>
decltype(auto) async_write(AsyncWriteStream& stream, message_type type,
  async_write_state& state, ConstBufferSequence payload,
  CompletionToken&& token);
```

Building a Toolbox

Overarching goal is to build a server application

Non-trivial software engineering projects rapidly become an ecosystem

Start with low level pieces and build incrementally higher layers

Small building blocks will be used by server but can also be used elsewhere

Approach allowed for an interactive client/debugger to be written easily

Heartbeats

```
enum class async_heartbeat_reason { wait, write };

struct async_heartbeat_statistics {
  std::size_t bytes_transferred;
  std::size_t messages;
  std::size_t waits;
  async_heartbeat_reason reason;
};

//  Completion signature:
//  void(std::error_code, async_heartbeat_statistics);
template<typename AsyncWriteStream, typename WaitableTimer, typename Events,
  typename CompletionToken>
decltype(auto) async_heartbeat(AsyncWriteStream& stream,
  async_write_state& state, WaitableTimer& timer, Events events,
  CompletionToken&& token);
```

Events

Complex asynchronous operations contain many parts

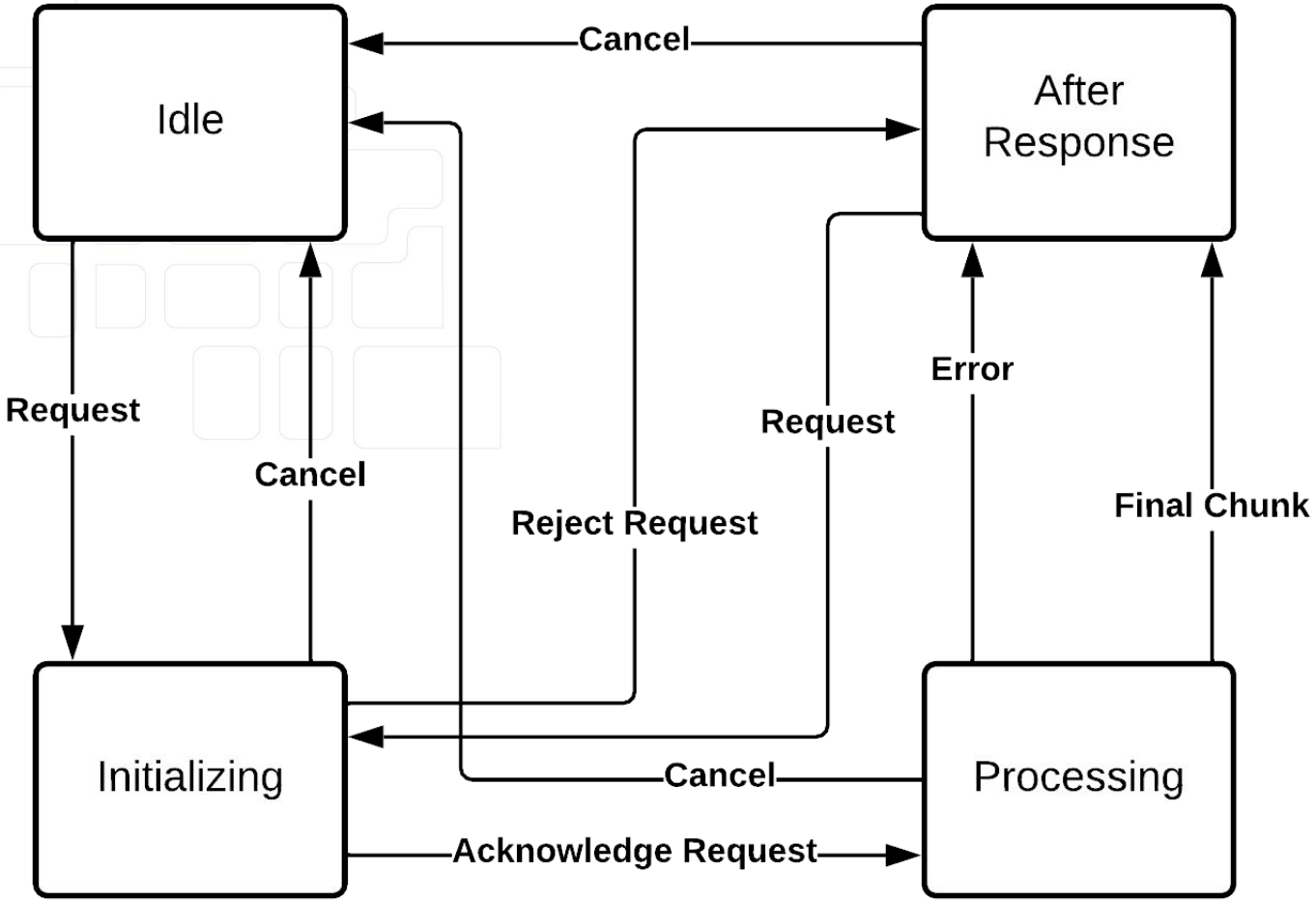
Encapsulating increasing complexity creates a shared framework

Shared framework is useless if it can’t be adapted to specific use cases

Events object allows us to inject the action to take in specific use cases

Heartbeat Events

```
struct /* ... */ {
  std::chrono::duration<...> delay();
  void complete_write();
};
```


Idle
After Response
Initializing
Processing
Cancel
Request
Cancel
Reject Request
Request
Error
Final Chunk
Cancel
Acknowledge Request

After Response

Latency can cause client and server state to briefly desynchronize

Consider following sequence of events

1. Server completes query processing
2. Client sends cancel
3. Client receives final chunk
4. Server receives cancel

Similar motivation to `CLOSE_WAIT` and `TIME_WAIT` in TCP

```
enum class async_control_reason { read, write, run, init, protocol };

struct async_control_statistics {
  std::size_t bytes_read;
  std::size_t bytes_written;
  std::size_t messages_read;
  std::size_t messages_written;
  std::size_t accepted;
  std::size_t rejected;
  std::size_t cancelled;
  async_control_reason reason;
};

//  Completion signature:
//  void(std::error_code, async_control_statistics);
template<typename AsyncStream, typename DynamicBufferV2, typename Events,
  typename CompletionToken>
decltype(auto) async_control(AsyncStream& stream, DynamicBufferV2 buffer,
  Events events, CompletionToken&& token);
```

Control Events

```
struct /* ... */ {
  void complete_write();
  bool cancel();
  template<typename ConstBufferSequence>
  auto init(ConstBufferSequence cb);
  // ...
};
```

What Does `init` Return?

```
template<typename ConstBufferSequence>
struct async_control_accept {
  async_control_accept() = default;
  explicit async_control_accept(ConstBufferSequence payload) noexcept(...);
  ConstBufferSequence payload;
};

template<typename ConstBufferSequence>
struct async_control_reject {
  async_control_reject() = default;
  explicit async_control_reject(ConstBufferSequence payload) noexcept(...);
  ConstBufferSequence payload;
};
```

What Does `init` Return?

```
template<typename AcceptConstBufferSequence,
  typename RejectConstBufferSequence = AcceptConstBufferSequence>
using async_control_init_result = std::variant<
  async_control_accept<AcceptConstBufferSequence>,
  async_control_reject<RejectConstBufferSequence>,
  std::error_code>;
```

Control Events (Cont.)

```
struct /* ... */ {
  // ...
  //  Completion signature:
  //  void(async_write_state&);
  template<typename CompletionToken>
  decltype(auto) async_start_write(CompletionToken&& token);
  //  Completion signature:
  //  void(std::error_code, ...);
  template<typename CompletionToken>
  decltype(auto) async_run(CompletionToken&& token);
  //  Completion signature:
  //  void(std::error_code);
  template<typename CompletionToken>
  decltype(auto) async_wait(CompletionToken&& token);
};
```

Injecting Asynchronous Operations

If an asynchronous operation completes immediately (that is, within the thread of execution calling the initiating function, and before the initiating function returns), the completion handler shall be submitted for execution (in a manner that shall not block forward progress of the caller pending completion thereof).

—NetTS §13.2.7.12 (async.reqmts.async.completion)

Requirement often unnecessary for injected initiating functions

Enforcing would leave performance on the table

Can waive this requirement for injected asynchronous operations

```
enum class query_driver_reason { error, cancel, complete, fail,
  abort };

struct query_driver_statistics {
  std::size_t bytes_transferred;
  std::size_t payload_bytes_transferred;
  std::size_t messages;
  std::size_t runs;
  query_driver_reason reason;
};

struct query_driver_settings {
  std::chrono::nanoseconds poll_interval;
};
```

```
template<typename AsyncWriteStream, typename WaitableTimer,
  typename Query>
struct query_driver {
  using stream_type = sticky_cancel_async_stream<
    AsyncWriteStream>;
  query_driver(AsyncWriteStream stream, Query q,
    query_driver_settings settings);
  Query& query() noexcept;
  const Query& query() const noexcept;
  void abort();
  typename WaitableTimer::time_point last_write() const noexcept;
  bool start_write() noexcept;
  void complete_write();
```

```
  bool cancel();
  template<typename CompletionToken>
  decltype(auto) async_start_write(CompletionToken&& token);
  template<typename CompletionToken>
  decltype(auto) async_run(CompletionToken&& token);
  template<typename CompletionToken>
  decltype(auto) async_wait(CompletionToken&& token);
};
```

Query

```
enum class query_status { wait, payload, error, complete };

struct /* ... */ {
  template<typename ConstBufferSequence>
  auto init(ConstBufferSequence cb);
  query_status run();
  void end() noexcept;
  auto payload() const;
  void toggle();
};
```

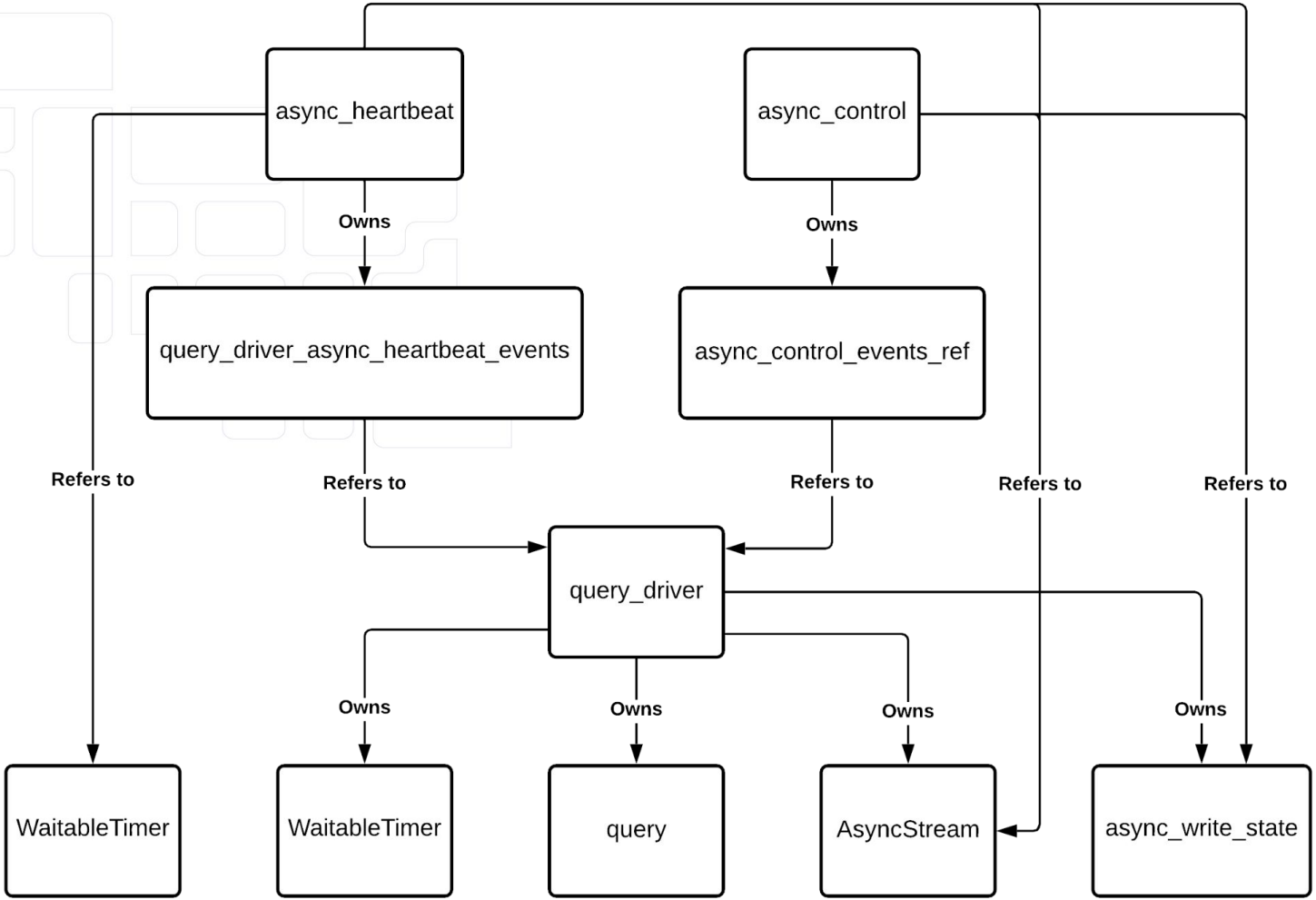
Double Buffering

Generated chunks must remain valid as they are sent

Once send completes another chunk may be written

Rather than lazily refill the buffer maintain a “front” and “back” buffer

- Send from one buffer while filling the other in the background
- `toggle` switches the front and back buffer


async_heartbeat
async_control
Owns
Owns
query_driver_async_heartbeat_events
async_control_events_ref
Refers to
Refers to
Refers to
Refers to
Refers to
query_driver
Owns
Owns
Owns
Owns
WaitableTimer
WaitableTimer
query
AsyncStream
async_write_state

Accepting Incoming Connections

```
//  Completion signature:
//  void(std::error_code, std::size_t);
template<typename Acceptor, typename Events,
  typename CompletionToken>
decltype(auto) async_accept(Acceptor& acc,
  typename Acceptor::endpoint_type& ep, Events events,
  CompletionToken&& token);

struct /* ... */ {
  decltype(auto) next_context();
  template<typename Endpoint, typename AsyncStream>
  void accepted(Endpoint ep, AsyncStream stream);
};
```

Maintaining Worker Threads

```
struct thread_pool {
  explicit thread_pool(unsigned threads);
  template <typename Function>
  std::size_t run(Function f);
  void stop() noexcept;
  using iterator = /* ... */;
  iterator begin() noexcept;
  iterator end() noexcept;
  using const_iterator = /* ... */;
  const_iterator begin() const noexcept;
  const_iterator end() const noexcept;
};
```

```
Opening DataConn socket on 0.0.0.0:11653...
Started (send SIGINT or SIGTERM to exit)
2021-10-06T04:01:26.490694105Z
  Accepted connection 10.244.0.119:42972 => 0.0.0.0:11653
2021-10-06T04:01:26.490785365Z
  10.244.0.119:42972 => 0.0.0.0:11653 request: {"version":"v1","query":"p
2021-10-06T04:01:26.721512358Z
  10.244.0.119:42972 => 0.0.0.0:11653 accept: {"format":"application/jso
2021-10-06T04:01:26.870505319Z
  10.244.0.119:42972 => 0.0.0.0:11653 complete (1184 bytes)
2021-10-06T04:01:26.871665987Z
  10.244.0.119:42972 => 0.0.0.0:11653 disconnected due to failure reading
```

Putting it All Together

```
template<typename Query, typename Configuration, typename Events>
void run_server(Configuration& config, Events events);

struct /* ... */ {
  template<typename Configuration>
  auto with_pool(Configuration& config, thread_pool& pool);
  void thread_start(asio::io_context& ctx);
};
```

```
template<typename Query>
struct connection_state {
  using query_type = verbose_query<Query, connection_description>;
  using query_driver_type = /* ... */;
  template<typename Configuration, typename WithPool>
  connection_state(asio::ip::tcp::socket s,
    asio::ip::tcp::endpoint local, asio::ip::tcp::endpoint remote,
    Configuration& config, WithPool& with_pool);
  query_driver_type driver;
  sticky_cancel_waitable_timer<timer> heartbeat_timer;
  std::vector<byte> buffer;
  struct done_type {
    std::error_code ec;
    std::variant<async_control_reason, async_heartbeat_reason>
      reason;
  };
  std::optional<done_type> result;
```

```
template<typename Pointee, typename Reason>
void done(std::shared_ptr<Pointee>&& ptr, std::error_code ec,
  Reason reason)
{
  result = result.value_or(done_type{ec, reason});
  driver.abort();
  heartbeat_timer.cancel();
  if (ptr.use_count() != 1) {
    ptr.reset();
    return;
  }
  driver.query().write_timestamp();
  std::cout << "  " << driver.query().description() <<
    " disconnected due to failure ";
```

```
    struct {
      void operator()(async_control_reason reason) const {
        switch (reason) {
        // ...
        }
      }
      void operator()(async_heartbeat_reason reason) const {
        switch (reason) {
        // ...
        }
      }
    } visitor;
    std::visit(visitor, result->reason);
    std::cout << ": " << result->ec.message() << std::endl;
  }
};
```

```
struct accept_state {
  accept_state(asio::io_context& ctx,
    asio::ip::tcp::endpoint local) : acceptor(ctx), local(local)
  {
    acceptor.open(local.protocol());
    acceptor.set_option(run_server::acceptor::reuse_address(
      true));
    acceptor.bind(local);
    acceptor.listen();
  }
  asio::ip::tcp::acceptor acceptor;
  asio::ip::tcp::endpoint local;
  asio::ip::tcp::endpoint remote;
};
```

```
template<typename Query, typename Configuration, typename Events>
void run_server(Configuration& config, Events events) {
  thread_pool pool(config.threads());
  auto with_pool = events.with_pool(config, pool);
  std::list<accept_state> accepts;
  for (auto&& local : config.endpoints()) {
    std::cout << "Opening DataConn socket on " << local << "..." <<
      std::endl;
    auto&& state = accepts.emplace_back(*pool.begin(), local);
    auto on_accept = /* ... */;
    async_accept(state.acceptor, state.remote,
      async_accept_round_robin_events(pool.begin(), pool.end(),
      std::move(on_accept)), [&state](auto ec, auto)
    {
      std::cerr << "Error (" << ec.message() << ") accepting on " <<
        state.local << std::endl;
      throw std::system_error(ec);
    });
  }
```

```
auto on_accept = [&, local](asio::ip::tcp::endpoint remote,
  asio::ip::tcp::socket stream)
{
  auto ex = stream.get_executor();
  asio::execution::execute(std::move(ex), [&, local, remote, stream =
    std::move(stream)]() mutable
  {
    auto ptr = std::make_shared<connection_state<Query,
      Configuration>>(std::move(stream), local, remote, config, with_pool);
    ptr->driver.query().write_timestamp();
    std::cout << "  Accepted connection " << remote << " => " << local <<
      std::endl;
    async_control(ptr->driver.stream(), dynamic_buffer(ptr->buffer,
      config.max_buffer()), async_control_events_ref(ptr->driver),
      [ptr](auto ec, auto stats) mutable
    {
      ptr->done(std::move(ptr), ec, stats.reason);
    });
```

```
    auto&& state = *ptr;
    async_heartbeat(state.driver.stream(), state.driver.write_state(),
      state.heartbeat_timer, query_driver_async_heartbeat_events(
      state.driver, config.heartbeat_interval()),
      [ptr = std::move(ptr)] (auto ec, auto stats) mutable
    {
      ptr->done(std::move(ptr), ec, stats.reason);
    });
  });
};
```

```
  asio::signal_set signals(*pool.begin(), SIGINT, SIGTERM);
  signals.async_wait([&](auto, auto) {
    std::cout << "Stopping..." << std::endl;
    pool.stop();
  });
  std::cout << "Started (send SIGINT or SIGTERM to exit)" << std::endl;
  pool.run([&](auto&& ctx) {
    events.thread_start(ctx);
  });
}
```

Questions?

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca

MAYSTREET