The Networking TS in Practice: Patterns for Real World Problems

Robert Leahy

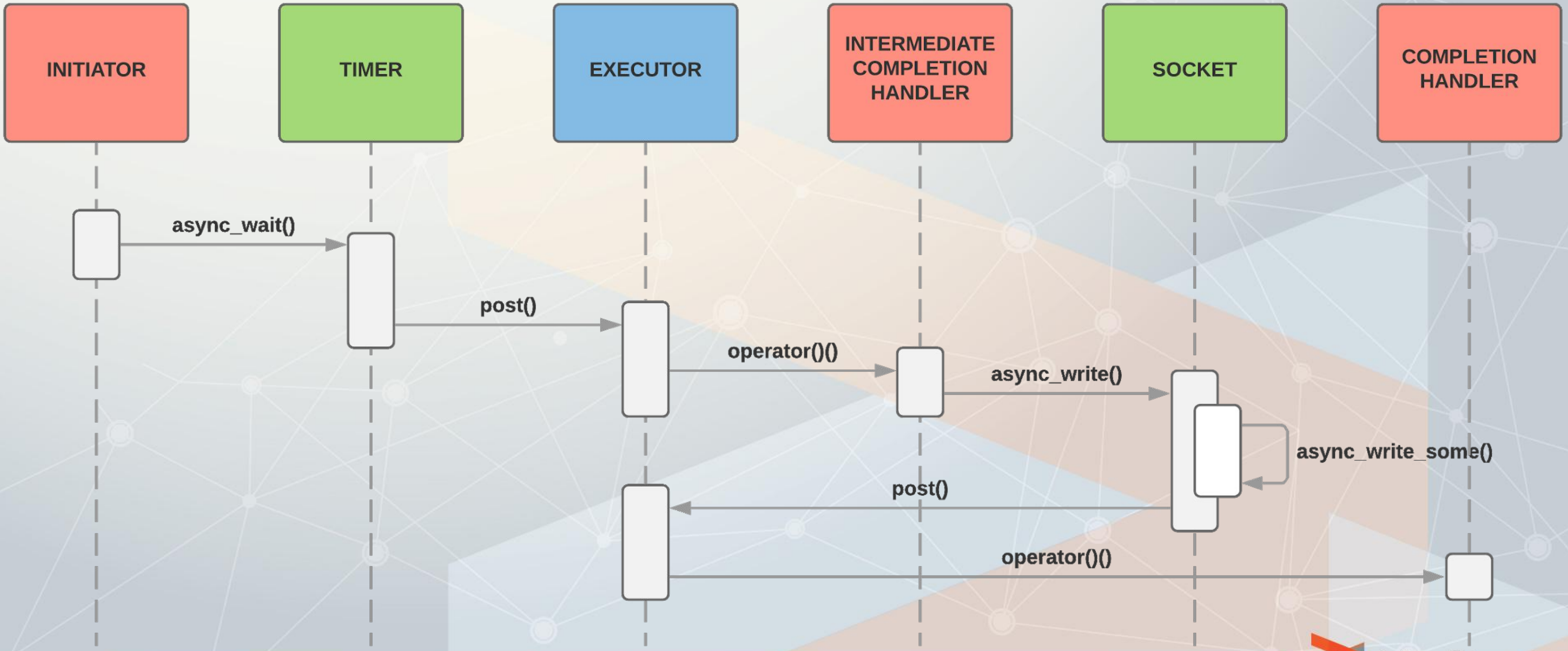
Senior Software Engineer
MayStreet Inc.
rleahy@rleahy.ca

MayStreet

Networking TS, Boost.Asio, & ISO C++

- Had hoped for Networking TS in C++20
- Now hoping for C++23
- Boost.Asio interface compatible
  - `boost::asio` namespace
  - Used to prepare these slides
- GCC 9 experimental implementation
  - `std::experimental::net` namespace
  - Incomplete

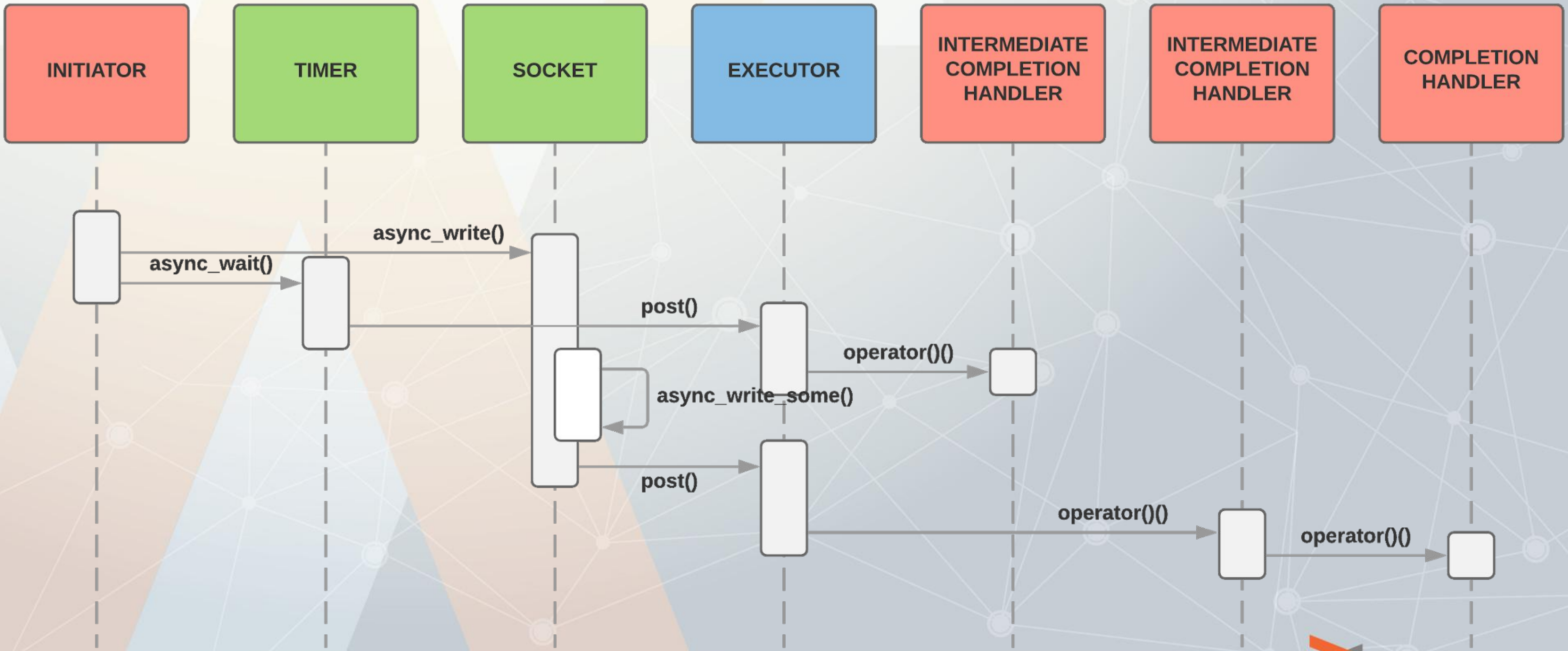
async_wait_then_write



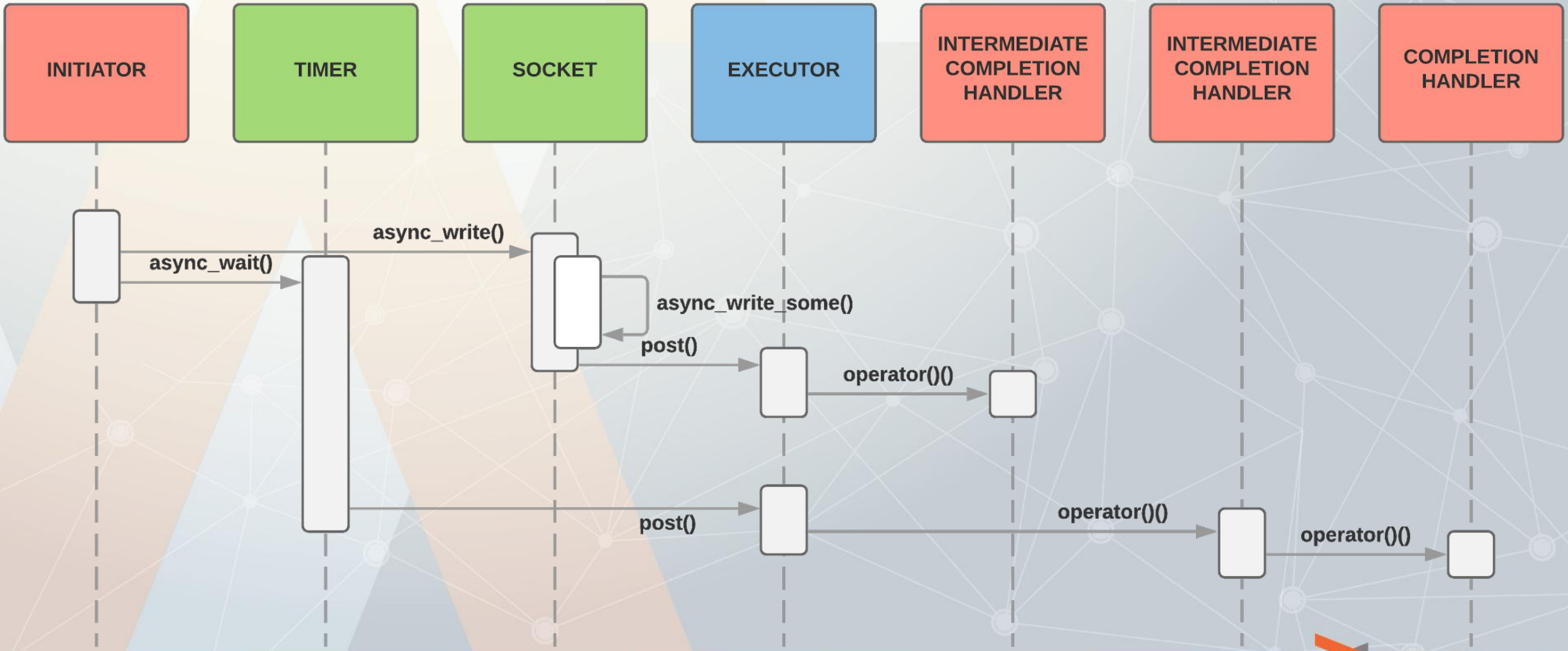
Series Composition

- Each successive operation waits for completion of previous operation
- Pattern can be extended to compose any number of operations **in series**
- Series composition only covers some use cases

async_write_with_timeout



async_write_with_timeout (Alternate)



Shared State

- Intermediate completion handlers must have shared state to:
  - Track completion of intermediate completion handlers
  - Collect results
  - Store final completion handler

```cpp
template<class AsyncWriteStream, class ConstBuffers, class Timer, class Token>
auto async_write_with_timeout(AsyncWriteStream& stream, ConstBuffers cb, Timer& timer,
  typename Timer::duration timeout, Token&& token)
{
  using completion_type = /* ... */;
  using handler_type = /* ... */;
  struct state : private boost::noncopyable { /* ... */ };
  class write_op { /* ... */ };
  class timeout_op { /* ... */ };
  completion_type completion(token);
  auto a = std::net::get_associated_allocator(completion.completion_handler);
  auto ptr = std::allocate_shared<state>(a, stream, timer,
    std::move(completion.completion_handler));
  std::net::async_write(ptr->stream, cb, write_op(ptr));
  timer.expires_after(timeout);
  timer.async_wait(timeout_op(std::move(ptr)));
  return completion.result.get();
}
```

```
struct state : private boost::noncopyable {
  state(AsyncWriteStream& s, Timer& t, handler_type h) noexcept(/* ... */)
    : stream(s), timer(t), handler(std::move(h))
  {}
  AsyncWriteStream& stream;
  Timer& timer;
  handler_type handler;
  std::error_code ec;
  std::size_t bytes_transferred = 0;
  std::size_t outstanding = 2;
};
```

```
class write_op {
  std::shared_ptr<state> state_;
public:
  using allocator_type = std::net::associated_allocator_t<handler_type>;
  auto get_allocator() const noexcept {
    return std::net::get_associated_allocator(state_->handler);
  }
  using executor_type = std::net::associated_executor_t<handler_type,
    decltype(stream.get_executor())>;
  auto get_executor() const noexcept {
    return std::net::get_associated_executor(state_->handler,
      state_->stream.get_executor());
  }
  explicit write_op(std::shared_ptr<state> s) noexcept : state_(std::move(s)) {}
  write_op(write_op&&) = default;
  write_op& operator=(write_op&&) = delete;
  write_op(const write_op&) = delete;
  write_op& operator=(const write_op&) = delete;
  // Next slide...
};
```

```
class write_op {
  // Previous slide...
  void operator()(std::error_code ec, std::size_t bytes_transferred) {
    if (--state_->outstanding) {
      state_->ec = ec;
      state_->bytes_transferred = bytes_transferred;
      state_.reset();
    } else {
      auto h = std::move(state_->handler);
      ec = state_->ec;
      state_.reset();
      h(ec, bytes_transferred);
    }
  }
};
```

```
class timeout_op {
  std::shared_ptr<state> state_;
public:
  using allocator_type = std::net::associated_allocator_t<handler_type>;
  auto get_allocator() const noexcept {
    return std::net::get_associated_allocator(state_->handler);
  }
  using executor_type = std::net::associated_executor_t<handler_type,
    decltype(timer.get_executor())>;
  auto get_executor() const noexcept {
    return std::net::get_associated_executor(state_->handler,
      state_->timer.get_executor());
  }
  explicit timeout_op(std::shared_ptr<state> s) noexcept : state_(std::move(s)) {}
  timeout_op(timeout_op&&) = default;
  timeout_op& operator=(timeout_op&&) = delete;
  timeout_op(const timeout_op&) = delete;
  timeout_op& operator=(const timeout_op&) = delete;
  // Next slide...
};
```

```cpp
class timeout_op {
  // Previous slide...
  void operator()(std::error_code) {
    if (--state_->outstanding) {
      state_->ec = make_error_code(std::errc::timed_out);
      state_.reset();
    } else {
      auto h = std::move(state_->handler);
      auto ec = state_->ec;
      auto bytes_transferred = state_->bytes_transferred;
      state_.reset();
      h(ec, bytes_transferred);
    }
  }
};
```

Cancelation

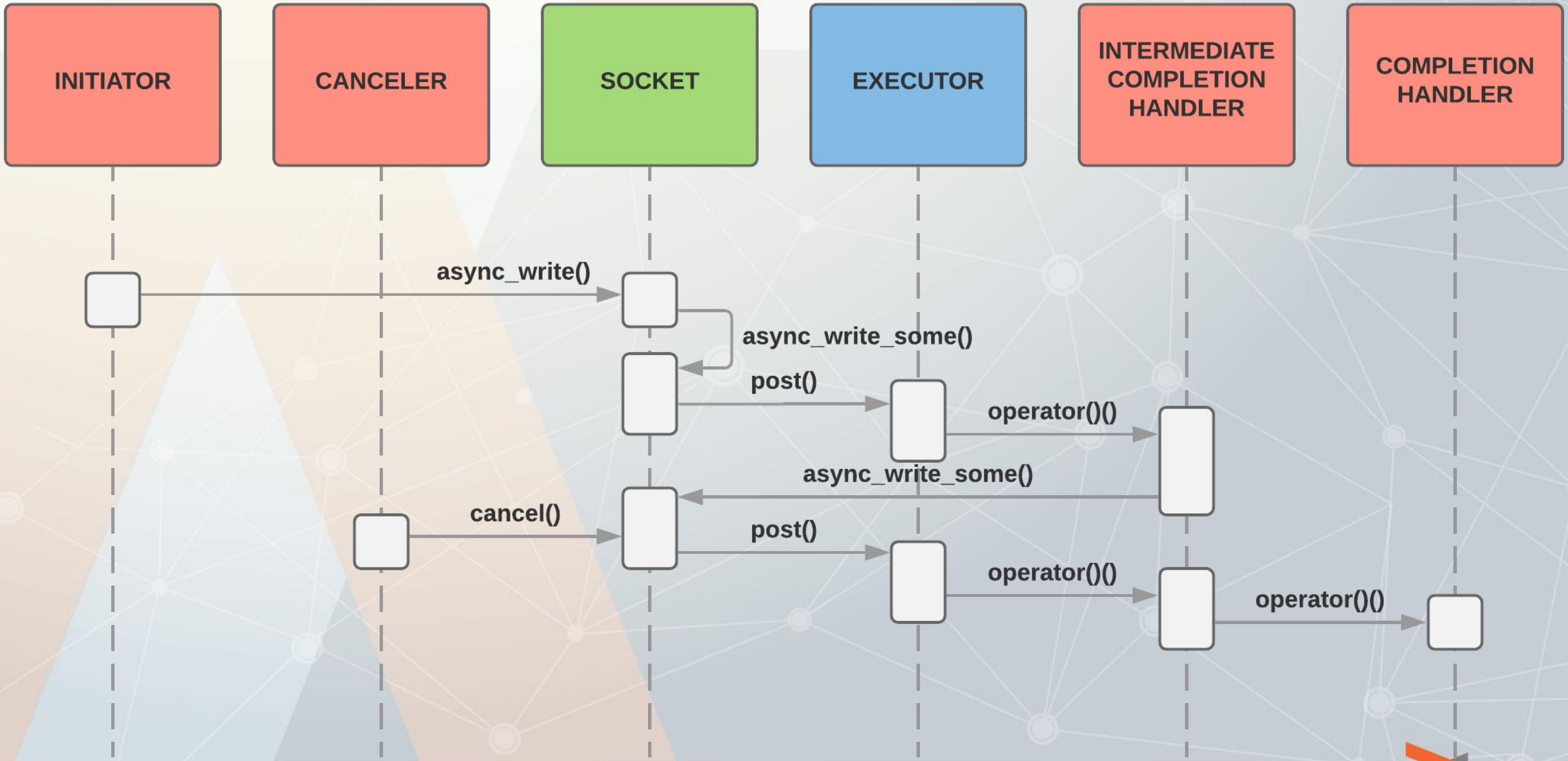
`size_t cancel();`

Effects: Causes any outstanding asynchronous wait operations to complete. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

Returns: The number of operations that were canceled.

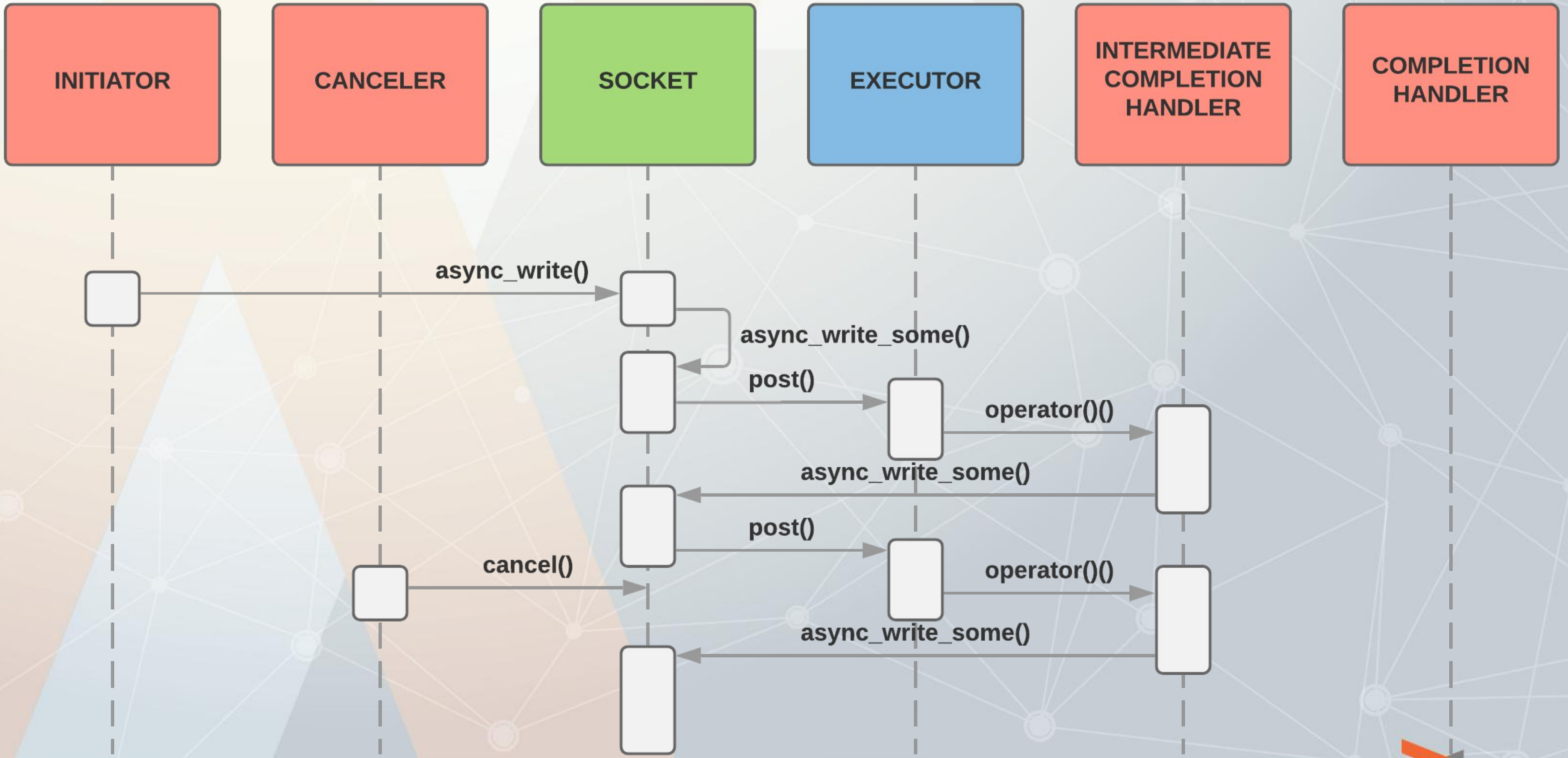
Remarks: Does not block the calling thread pending completion of the canceled operations.

`cancel` (Outstanding Operation)



MayStreet

`cancel` (No Outstanding Operations)



```cpp
template<typename AsyncWriteStream>
class sticky_cancel_async_write_stream {
  AsyncWriteStream& stream_;
  bool canceled_ = false;
public:
  explicit sticky_cancel_async_write_stream(AsyncWriteStream& stream) noexcept
    : stream_(stream)
  {}
  auto get_executor() const noexcept {
    return stream_.get_executor();
  }
  auto cancel() {
    canceled_ = true;
    return stream_.cancel();
  }
  // Next slide...
};
```

```cpp
template<typename AsyncWriteStream>
class sticky_cancel_async_write_stream {
  // Previous slide...
  template<typename ConstBufferSequence, typename Token>
  auto async_write_some(ConstBufferSequence cb, Token&& token) {
    if (!canceled_) {
      return stream_.async_write_some(std::move(cb), std::forward<Token>(token));
    }
    /* ... */ completion(token);
    auto ex = std::net::get_associated_executor(completion.completion_handler,
      get_executor());
    auto a = std::net::get_associated_allocator(completion.completion_handler);
    auto f = [handler = std::move(completion.completion_handler)]() mutable {
      handler(make_error_code(std::errc::operation_canceled), 0);
    };
    ex.post(std::move(f), a);
    return completion.result.get();
  }
};
```

```
struct state : private boost::noncopyable {
  state(AsyncWriteStream& s, Timer& t, handler_type h) noexcept(/* ... */)
    : stream(s), timer(t), handler(std::move(h))
  {}
  AsyncWriteStream& stream;
  sticky_cancel_async_write_stream<AsyncWriteStream> stream;
  Timer& timer;
  handler_type handler;
  std::error_code ec;
  std::size_t bytes_transferred = 0;
  std::size_t outstanding = 2;
};
```


MayStreet

```cpp
class write_op {
  // ...
  void operator()(std::error_code ec, std::size_t bytes_transferred) {
    state_->timer.cancel();
    if (--state_->outstanding) {
      state_->ec = ec;
      state_->bytes_transferred = bytes_transferred;
      state_.reset();
    } else {
      auto h = std::move(state_->handler);
      ec = state_->ec;
      state_.reset();
      h(ec, bytes_transferred);
    }
  }
};
```

```
class timeout_op {
  // ...
  void operator()(std::error_code) {
    state_->stream.cancel();
    if (--state_->outstanding) {
      state_->ec = make_error_code(std::errc::timed_out);
      state_.reset();
    } else {
      auto h = std::move(state_->handler);
      auto ec = state_->ec;
      auto bytes_transferred = state_->bytes_transferred;
      state_.reset();
      h(ec, bytes_transferred);
    }
  }
};
```

Parallelism

- Natural happens-before relationship when composing in series
- Not so with parallel composition
- `std::net::strand`
  - Has runtime cost
  - Guarantee may already be met
  - Initiations and completions can still race
- Put onus on user to maintain strand
  - Boost.Beast does this with `boost::beast::basic_stream`

Managing Parallelism

- Single-threaded I/O objects and assumption of strand make code
  - Easy to write
  - Easy to reason about
  - Efficient by default
- CPUs have many cores so applications ought to leverage parallelism
- Single `std::net::io_context` with many threads using explicit strands
  - Error prone
  - Adds overhead
- Many `std::net::io_context` each running in a single thread
  - Mostly correct by default
  - No overhead
  - Need to distribute work

Round Robin

- Consider TCP server
  - Continually accepts incoming connections
  - Easy to parallelize management of different connections
- Using many `std::net::io_context` objects leverages parallelism implicit in problem
- `std::net::basic_socket_acceptor` allows accepted socket to be associated with arbitrary `std::net::io_context`

```
template<class CompletionToken>
DEDUCED async_accept(io_context&, CompletionToken&&);
```


MayStreet

```cpp
template<class Acceptor, class Iterator, class CompletionHandler>
struct async_accept_op {
  Acceptor& acc_;
  Iterator begin_, curr_, end_;
  CompletionHandler h_;
  using allocator_type = /* ... */;
  auto get_allocator() const noexcept { /* ... */ }
  using executor_type = /* ... */;
  auto get_executor() const noexcept { /* ... */ }
  template<typename Stream>
  void operator()(std::error_code ec, Stream s) {
    ++curr_;
    if (curr_ == end_) {
      curr_ = begin_;
    }
    h_(ec, std::move(s), curr_);
  }
};
```

```cpp
template<typename Acceptor, typename Iterator, typename Token>
auto async_accept(Acceptor& acc, Iterator begin, Iterator curr, Iterator end,
  Token&& token)
{
  using completion_type = /* ... */;
  completion_type completion(token);
  using op_type = async_accept_op<Acceptor, Iterator,
    typename completion_type::completion_handler_type>;
  op_type op{acc, begin, curr, end, std::move(completion.completion_handler)};
  acc.async_accept(curr->context(), std::move(op));
  return completion.result.get();
}
```


MayStreet

Operations that Never Succeed

- In networking processing often proceeds until exceptional condition
  - Connection termination
  - Server/client shutdown
- Until now all asynchronous operations complete after performing well-defined amount of work and report success or failure
- Asynchronous operation which never succeeds and proceeds until exceptional condition better models real world
- Repeatedly accept incoming connections and round robin `std::net::io_context` objects until exceptional condition

```cpp
template<typename Acceptor, typename Iterator, typename AfterAccept, typename Handler>
struct async_accept_loop_op {
  Acceptor& acc_;
  Iterator begin_, end_;
  AfterAccept after_;
  Handler h_;
  using allocator_type = /* ... */;
  auto get_allocator() const noexcept { /* ... */ }
  using executor_type = /* ... */;
  auto get_executor() const noexcept { /* ... */ }
  void initiate(Iterator curr) {
    auto&& acc = acc_;
    auto begin = begin_;
    auto end = end_;
    ::async_accept(acc, begin, curr, end, std::move(*this));
  }
  // Next slide...
};
```

```cpp
template<typename Acceptor, typename Iterator, typename AfterAccept, typename Handler>
struct async_accept_loop_op {
  // Previous slide...
  template<typename Stream>
  void operator()(std::error_code ec, Stream s, Iterator curr) {
    if (ec) {
      h_(ec);
      return;
    }
    after_(std::move(s));
    initiate(curr);
  }
};
```

```cpp
template<typename Acceptor, typename Iterator, typename AfterAccept, typename Token>
auto async_accept(Acceptor& acc, Iterator begin, Iterator end,
  AfterAccept after_accept, Token&& token)
{
  using completion_type = /* ... */;
  completion_type completion(token);
  using op_type = async_accept_loop_op<Acceptor, Iterator, AfterAccept,
    typename completion_type::completion_handler_type>;
  op_type op{acc, begin, end, std::move(after_accept),
    std::move(completion.completion_handler)};
  op.initiate(begin);
  return completion.result.get();
}
```

Summary

- Real world problems seem to resist the framework and patterns of the Networking TS but with care they can be reconciled
- Composition of operations in parallel requires special attention to `Executor` choice to avoid data races
- Multiple `std::net::io_context` objects make parallelism more manageable
- Not all operations need to have success completion condition

MayStreet

Questions

Full examples (including tests)

https://github.com/RobertLeahy/CppCon2019Talk