MAYSTREET

THE NETWORKING TS FROM SCRATCH

I/O Objects

Robert Leahy - rleahy@rleahy.ca

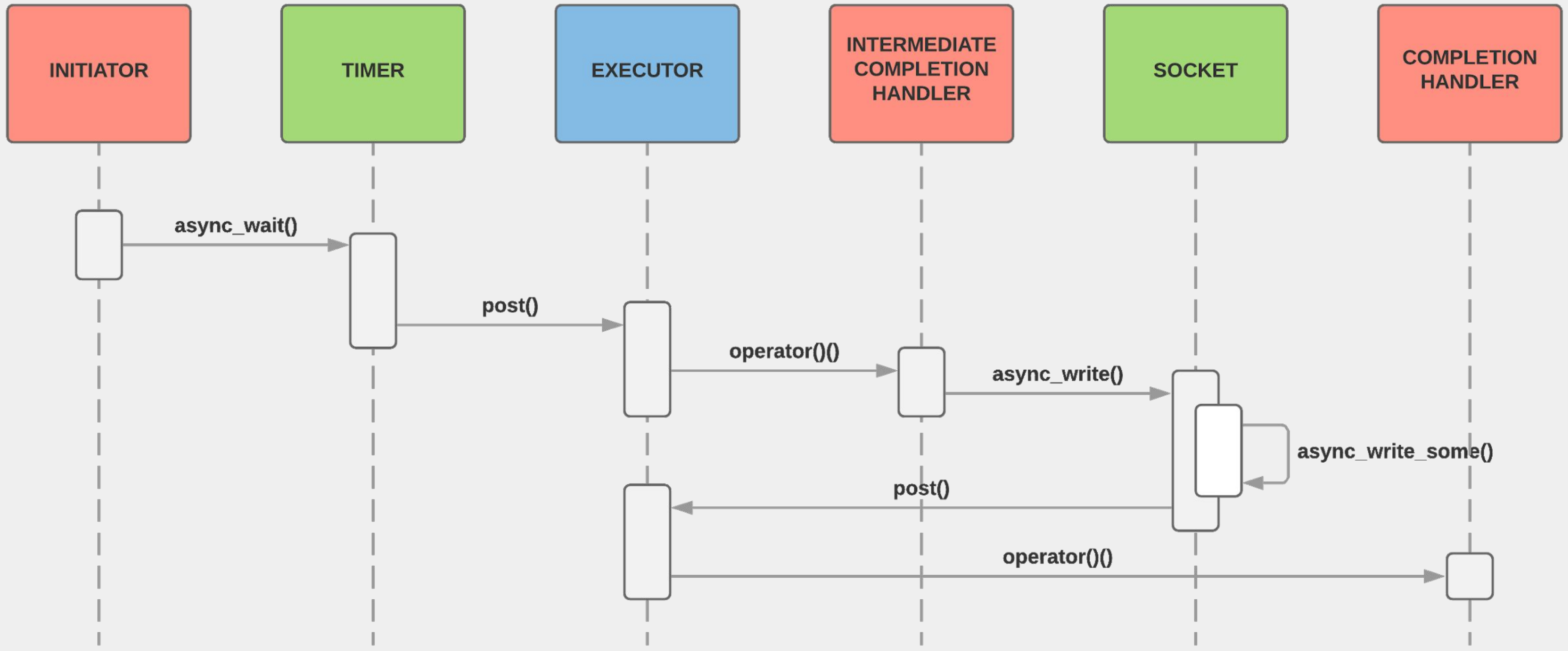
ISO C++ Networking is blocked on Executors (P0443)

Target for Networking is currently C++23 (P0592)

Used standalone Asio (1.18.0) to prepare these slides

Using the P0443-friendly extensions to the Networking TS shipping in “standalone” Asio & Boost.Asio (P0958)

async_wait_then_write

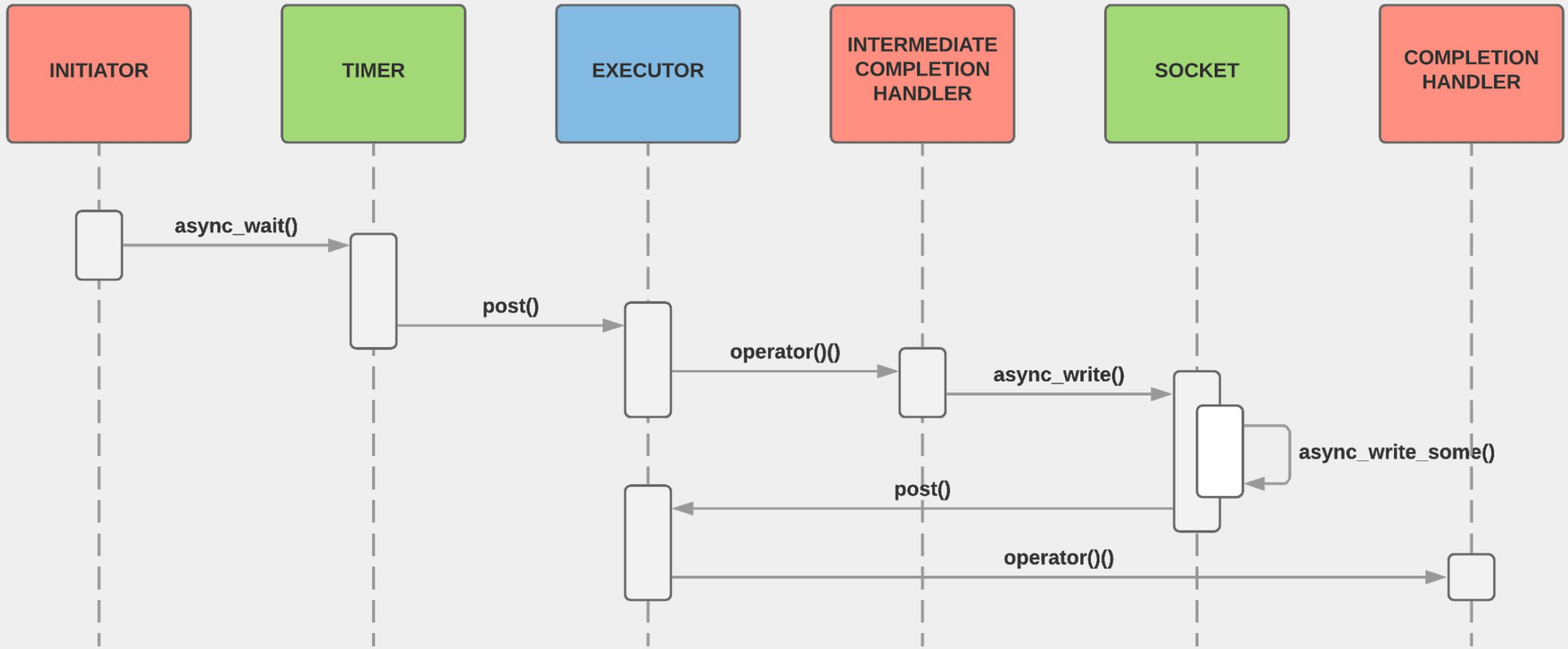


Composing asynchronous operations enables the layered construction of operations whose power increases with each layer

Templates & named type requirements decouple composed asynchronous operations from concrete types

Customization points defer decisions to the consumer rather than the author making them arbitrarily

async_wait_then_write



Asynchronous operations are “atomic” pieces of asynchronous functionality

Composing asynchronous operations presupposes asynchronous operations to compose

Guarantees of composed asynchronous operations depend on those guarantees being honored transitively

"I/O objects" provide a handle to low level asynchronous functionality within the framework of the Networking TS

Examples:

```
std::net::ip::tcp::socket
std::net::steady_timer
```

Possibilities:

- EFVI-based multicast socket
- Database access layer
- Printer spool

```
template<std::execution::executor Executor>
struct basic_async_event {
  using executor_type = Executor;
  explicit basic_async_event(executor_type ex);
  auto get_executor() const noexcept;
  std::size_t notify_one();
  std::size_t notify_all();
  template<typename CompletionToken>
  decltype(auto) async_wait(CompletionToken&& token);
};

using async_event = basic_async_event<std::net::io_context::
  executor_type>;
```

Asynchronous operations must store the final completion handler

Composed asynchronous operations store final completion handler in intermediate completion handler which is then passed to a lower layer for storage

Since there is no lower layer we can no longer foist the responsibility for storage onto the next layer

Since completion handler could be of any invocable type must use type erased wrapper

`std::function` isn't viable as `CompletionHandler` only requires `Cpp17MoveConstructible`

Need something like `any_invocable`

```
template<typename T>
class pending;

template<typename R, typename... Args>
class pending<R(Args...)> {
  //  1. Allocates storage for a std::decay_t<T> using associated
  //     allocator
  //  2. Forwards sole argument to construct std::decay_t<T> in allocated
  //     storage
  template<typename T> requires(!std::is_same_v<std::decay_t<T>,
    pending>)
  explicit pending(T&& t);
  //  1. Moves the target out of the managed storage
  //  2. Deallocates the managed storage
  //  3. Invokes the target
  R operator()(Args... args);
};
```

```
template<std::execution::executor Executor>
class basic_async_event {
  Executor ex_;
  std::vector<pending<void()>> pendings_;
public:
  explicit basic_async_event(Executor ex) : ex_(std::move(ex)) {}
  auto get_executor() const noexcept { return ex_; }
  std::size_t notify_one() {
    if (pendings_.empty()) return 0;
    auto pending = std::move(pendings_.front());
    pendings_.erase(pendings_.begin());
    pending();
    return 1;
  }
  std::size_t notify_all() { /* ... */ }
```

```
  template<typename CompletionHandler>
  void async_wait(CompletionHandler&& h) {
    auto ex = std::net::get_associated_executor(h, ex_);
    pendings_.emplace_back([h = std::forward<CompletionHandler>(h),
      ex = std::move(ex)]() mutable
    {
      auto alloc = std::net::get_associated_allocator(h);
      ex.dispatch(std::move(h), alloc);
    });
  }
};
```

```
template<class CompletionToken, std::net::completion_signature Signature,
  class Initiation, class... Args>
DEDUCED async_initiate(Initiation&& initiation, CompletionToken& token,
  Args&&... args);
```

Eliminates **`std::net::async_result`** boilerplate

Returns correct type & value for completion token

Initiation receives completion handler & trailing arguments

Supports lazy/deferred initiation strategies

```cpp
template<typename CompletionToken>
decltype(auto) async_wait(CompletionToken&& token) {
  return std::net::async_initiate<CompletionToken, void()>([&](auto h)
  {
    auto ex = std::net::get_associated_executor(h, ex_);
    pendings_.emplace_back([h = std::forward<CompletionHandler>(h),
      ex = std::move(ex)]() mutable
    {
      auto alloc = std::net::get_associated_allocator(h);
      ex.dispatch(std::move(h), alloc);
    });
  }, token);
}
```

`std::execution::execute` is a customization point object which submits work to an `Executor`

`dispatch`, `defer`, & `post` requested execution with certain properties

P0443 models this by establishing properties on `Executor`

Equivalent of `post` requires `blocking.never` and prefers `continuation.fork` on `Executor`

```
template<typename CompletionToken>
decltype(auto) async_wait(CompletionToken&& token) {
  return std::net::async_initiate<CompletionToken, void()>([&](auto h)
  {
    auto ex = std::net::get_associated_executor(h, ex_);
    pendings_.emplace_back([h = std::move(h), ex = std::move(ex)]()
      mutable
    {
      auto alloc = std::net::get_associated_allocator(h);
      auto alloc_ex = std::prefer(std::move(ex), std::execution::
        allocator(alloc));
      std::execution::execute(alloc_ex, std::move(h));
    });
  }, token);
}
```

`pending` owns the target (like `std::function`)

I/O object contains a container of `pending`

Transitively I/O object owns `pending`

Type erasure means target could be any type

Target could own the I/O object creating a cycle

Cycles could be deemed to be initiating function contract violation (Restinio takes this approach)

Completion handlers owning a state (perhaps including one or more I/O objects) is a useful pattern

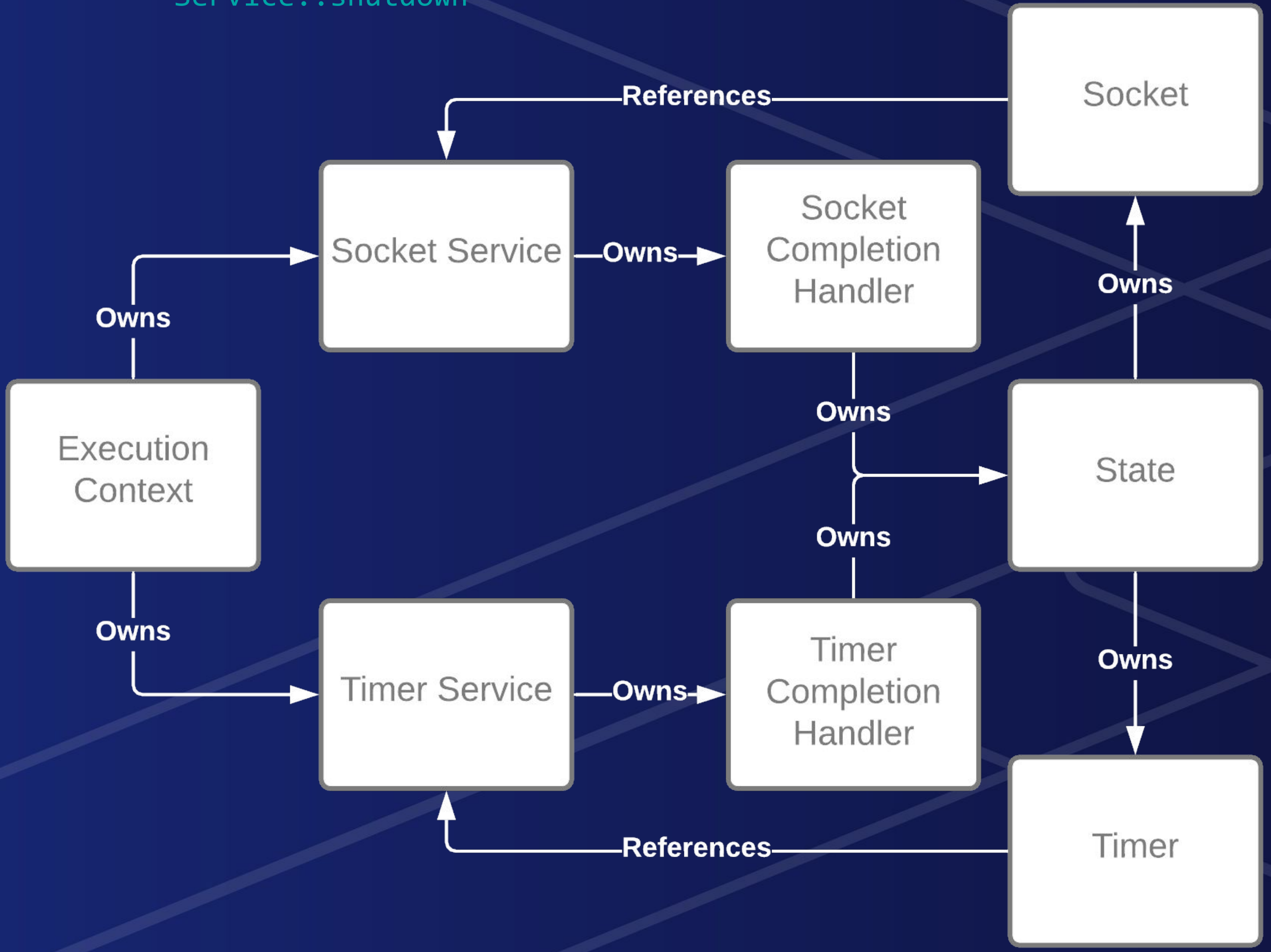
`Service` objects own completion handlers thereby breaking the potential ownership cycle

`Service` objects are owned by an `ExecutionContext`

`ExecutionContext` destroys all services (and therefore completion handlers) as its lifetime ends

```
struct my_service : std::net::execution_context::service {
  using key_type = my_service;
  explicit my_service(std::net::execution_context& ctx);
private:
  virtual void shutdown() noexcept override;
};
```

Service::shutdown



```cpp
template<typename Service>
typename Service::key_type& use_service(execution_context& e);

template <typename Service, typename... Args>
Service& make_service(execution_context& e, Args&&... args);

template<typename Service>
bool has_service(execution_context& e);
```

```
template<typename T>
struct service : std::net::execution_context::service {
  using key_type = service;
  explicit service(std::net::execution_context& ctx);
  //  Destroys all managed objects and ignores
  //  further calls to destroy()
  virtual void shutdown();
  //  Creates a managed object by forwarding the
  //  given arguments
  template<typename... Args>
  T* create(Args&&... args);
  //  Destroys the pointee unless shutdown() has
  //  been called
  void destroy(T* obj) noexcept;
};
```

```
template<std::execution::executor Executor>
class basic_async_event {
  using pendings_type = std::vector<pending<void()>>;
  using service_type = service<pendings_type>;
  Executor ex_;
  service_type& service_;
  pendings_type* pendings_;
public:
  explicit basic_async_event(Executor ex) : ex_(std::move(ex)),
    service_(std::net::use_service<service_type>(
      std::query(ex_, std::execution::context))),
    pendings_(service_.create()) {}
  ~basic_async_event() noexcept {
    service_.destroy(pendings_);
  }
  // ...
};
```

`ExecutionContext` provides an environment in which nullary-invocable objects are invoked

If there are pending objects `ExecutionContext` still has work to do

When there are no pending objects stopping may be desirable (e.g. to allow application to end)

`std::net::io_context::run` exhibits this behavior: Returns once there is no pending work

An asynchronous operation in progress clearly constitutes pending work

Asynchronous operation initiated by `basic_async_event::async_wait` does not immediately submit work to any `ExecutionContext`

`ExecutionContext` may therefore run out of work and stop before asynchronous operation completes `outstanding_work` property allows `Executor` to indicate that there is pending work

When `outstanding_work.tracked` is established lifetime of `Executor` corresponds to lifetime of outstanding work against the underlying `ExecutionContext`

In case of `std::net::io_context::run` will not return until lifetime of all such `Executor` objects ends

Completion handlers invoked via **`Executor`** obtained through **`associated_executor`** customization point

Completion handler may not have an **`Executor`** association

"I/O executor" obtained by calling nullary **`get_executor`** on I/O object

Provided as "default candidate object" when deferring to **`associated_executor`**

Two distinct **`Executor`** objects potentially involved in every asynchronous operation

Tracking outstanding work on associated **`Executor`** ensures underlying **`ExecutionContext`** is available to invoke completion handler

`ExecutionContext` underlying I/O executor may run out of pending work and no longer be available

`ExecutionContext` underlying I/O executor may manage operating system resource

On Linux `Executor` associated with socket may provide access to `epoll` file descriptor

These resources must remain available and active

There may be no need to directly submit work to I/O executor however still important to notify it of pending work

```
template<typename CompletionToken>
decltype(auto) async_wait(CompletionToken&& token) {
  return std::net::async_initiate<CompletionToken, void()>([ex = ex_,
    pendings = pendings_](auto h)
  {
    auto io_ex = std::prefer(ex, std::execution::outstanding_work
      .tracked);
    auto assoc_ex = std::net::get_associated_executor(h, ex);
    auto ex = std::prefer(std::move(assoc_ex), std::execution::
      outstanding_work.tracked);
    pendings->emplace_back([h = std::move(h), io_ex = std::move(io_ex),
      ex = std::move(ex)]() mutable { /* ... */ });
  }, token);
}
```

```
// ...
pendings->emplace_back([h = std::move(h), io_ex = std::move(io_ex), ex =
  std::move(ex)]() mutable
{
  auto alloc = std::net::get_associated_allocator(h);
  auto alloc_ex = std::prefer(std::move(ex), std::execution::allocator(
    alloc));
  std::execution::execute(alloc_ex, [h = std::move(h), io_ex =
    std::move(io_ex)]() mutable
  {
    auto local_ex = std::move(io_ex);
    std::move(h)();
  });
});
// ...
```

Asynchronous operations must establish guarantees of Networking TS from scratch

Completion handlers must be stored in a `Service` to avoid ownership cycles

Outstanding work must be tracked to ensure underlying `ExecutionContext` still available upon completion

MAYSTREET

Questions?