Document number: P2463R0
Date: 2021-09-27
Project: Programming Language C++
Audience: LEWG, SG1
Reply-to: Christopher Kohlhoff <chris@kohlhoff.com>

The Asio asynchronous model

Slides for P2444r0

```
socket.async_read_some(buffer,
    [](error_code e, size_t)
    {
      // ...
    }
  );
```

The same asynchronous operation used with a lambda…

```
socket.async_read_some(buffer,
    [](error_code e, size_t)
    {
      // ...
    }
  );
```

… with a future …

```
future<size_t> f =
  socket.async_read_some(
      buffer, use_future
    );

// ...

size_t n = f.get();
```

… in a coroutine …

```
awaitable<void> foo()
{
  size_t n =
    co_await socket.async_read_some(
        buffer, use_awaitable
      );

  // ...
}
```

… or in a fiber.

```
void foo()
{
  size_t n = socket.async_read_some(
      buffer, fibers::yield
    );

  // ...
}
```

```
void echo(tcp::socket s)
{
  try
  {
    char data[1024];
    for (;;)
    {
      std::size_t n = s.read_some(buffer(data));
      write(s, buffer(data, n));
    }
  }
  catch (const std::exception& e)
  {
  }
}

void listen(tcp::acceptor a)
{
  for (;;)
  {
    std::thread(echo, a.accept()).detach();
  }
}

int main()
{
  asio::io_context ctx;
  listen({ctx, {tcp::v4(), 55555}});
}
```

- Compositions can use the language to manage control flow (i.e. `for`, `if`, `while`, etc.).

- Compositions may be refactored to use child functions that run on the same thread (i.e. are simply called) without altering functionality.

- If a synchronous operation requires a temporary resource (such as memory, a file descriptor, or a thread), this resource is released before returning from the function.

- When a synchronous operation is generic (i.e. a template) the return type is deterministically derived from the function and its arguments.

- The lifetime of arguments to be passed to a synchronous operation is clear, including the ability to safely pass temporaries.

```
void echo(tcp::socket s)
{
  try
  {
    char data[1024];
    for (;;)
    {
      std::size_t n = s.read_some(buffer(data));
      write(s, buffer(data, n));
    }
  }
  catch (const std::exception& e)
  {
  }
}
```

```
void echo_once(tcp::socket& socket)
{
  char data[128];
  std::size_t n = socket.read_some(buffer(data));
  write(socket, buffer(data, n));
}

void echo(tcp::socket socket)
{
  try
  {
    for (;;)
    {
      echo_once(socket);
    }
  }
  catch (const std::exception& e)
  {
  }
}
```

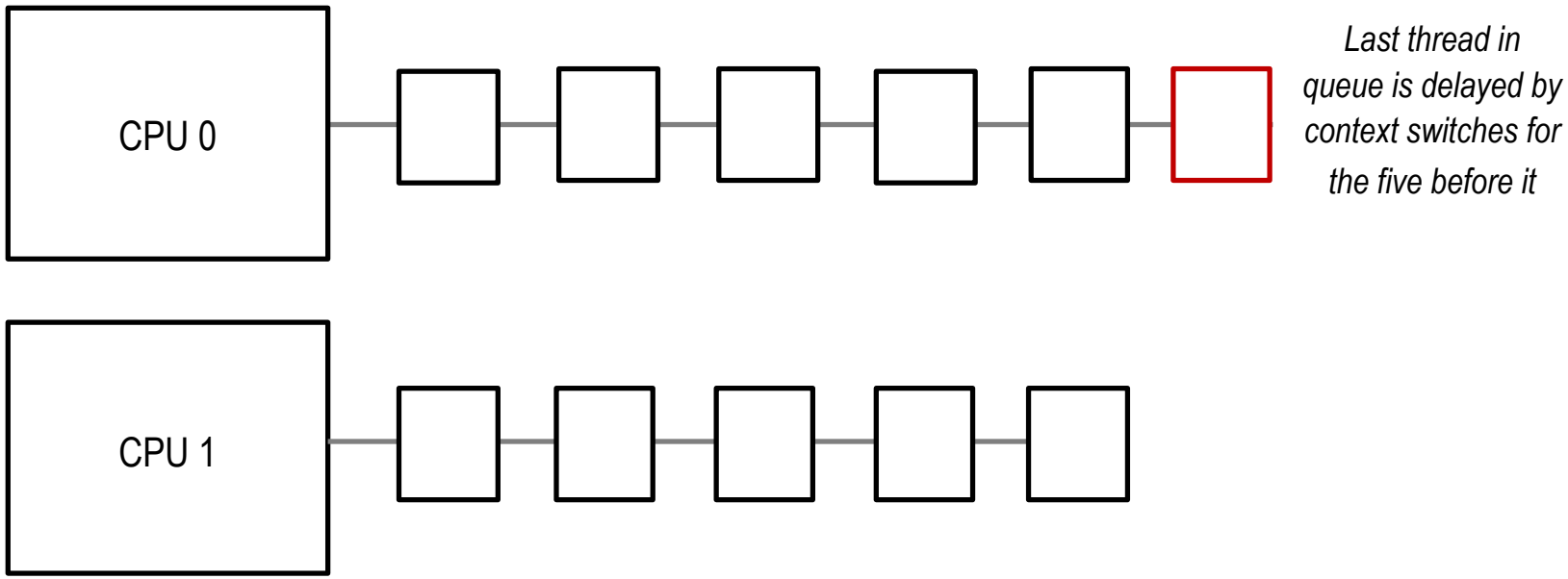
- Compositions can use the language to manage control flow (i.e. `for`, `if`, `while`, etc.).

- Compositions may be refactored to use child functions that run on the same thread (i.e. are simply called) without altering functionality.

- If a synchronous operation requires a temporary resource (such as memory, a file descriptor, or a thread), this resource is released before returning from the function.

- When a synchronous operation is generic (i.e. a template) the return type is deterministically derived from the function and its arguments.

- The lifetime of arguments to be passed to a synchronous operation is clear, including the ability to safely pass temporaries.


CPU 0
CPU 1
Last thread in queue is delayed by context switches for the five before it

```
class session : public std::enable_shared_from_this<session>
{
public:
  session(tcp::socket socket) : socket_(std::move(socket)) {}

  void do_read()
  {
    socket_.async_read_some(asio::buffer(data_, max_length),
        [self = shared_from_this()](std::error_code ec, std::size_t length)
        {
          if (!ec)
          {
            self->do_write(length);
          }
        });
  }

  void do_write(std::size_t length)
  {
    asio::async_write(socket_, asio::buffer(data_, length),
        [self = shared_from_this()](std::error_code ec, std::size_t /*length*/)
        {
          if (!ec)
          {
            self->do_read();
          }
        });
  }

  tcp::socket socket_;
  enum { max_length = 1024 };
  char data_[max_length];
};
```

```
void echo(tcp::socket s)
{
  try
  {
    char data[1024];
    for (;;)
    {
      std::size_t n = s.read_some(buffer(data));
      write(s, buffer(data, n));
    }
  }
  catch (const std::exception& e)
  {
  }
}

void listen(tcp::acceptor a)
{
  for (;;)
  {
    std::thread(echo, a.accept()).detach();
  }
}

int main()
{
  asio::io_context ctx;
  listen({ctx, {tcp::v4(), 55555}});
}
```

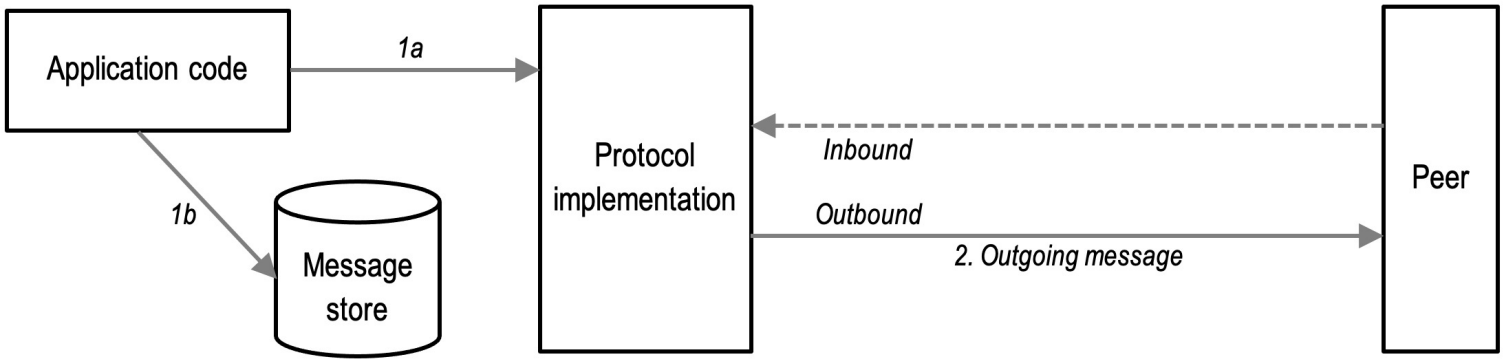
```
awaitable<void> echo(tcp::socket s)
{
  try
  {
    char data[1024];
    for (;;)
    {
      std::size_t n = co_await s.async_read_some(buffer(data), use_awaitable);
      co_await async_write(s, buffer(data, n), use_awaitable);
    }
  }
  catch (const std::exception& e)
  {
  }
}

awaitable<void> listen(tcp::acceptor a)
{
  for (;;)
  {
    co_spawn(a.get_executor(), echo(co_await a.async_accept(use_awaitable)), detached);
  }
}

int main()
{
  asio::io_context ctx;
  co_spawn(ctx, listen({ctx, {tcp::v4(), 55555}}), detached);
  ctx.run();
}
```

```
void echo_once(tcp::socket& socket)
{
  char data[128];
  std::size_t n = socket.read_some(buffer(data));
  write(socket, buffer(data, n));
}

void echo(tcp::socket socket)
{
  try
  {
    for (;;)
    {
      echo_once(socket);
    }
  }
  catch (const std::exception& e)
  {
  }
}
```

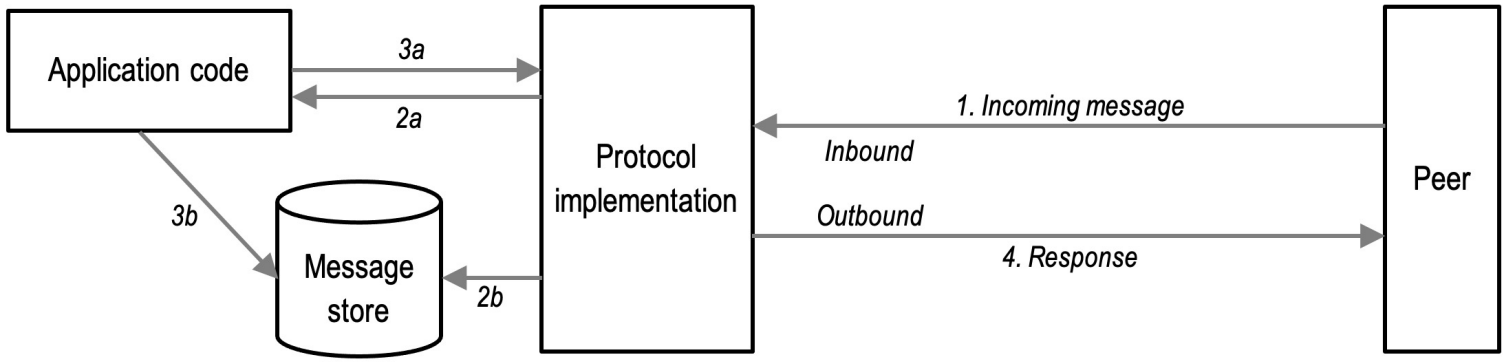
```
awaitable<void> echo_once(tcp::socket& socket)
{
  char data[128];
  std::size_t n = co_await socket.async_read_some(buffer(data), use_awaitable);
  co_await async_write(socket, buffer(data, n), use_awaitable);
}

awaitable<void> echo(tcp::socket socket)
{
  try
  {
    for (;;)
    {
      co_await echo_once(socket);
    }
  }
  catch (const std::exception& e)
  {
  }
}
```

Typical complexity: FIX protocol implementation

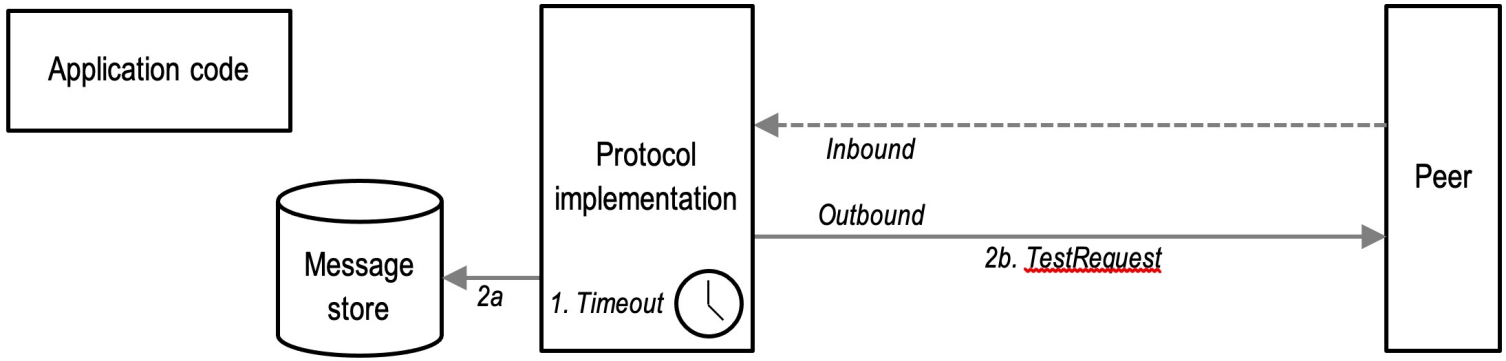
Application code generates an unsolicited message



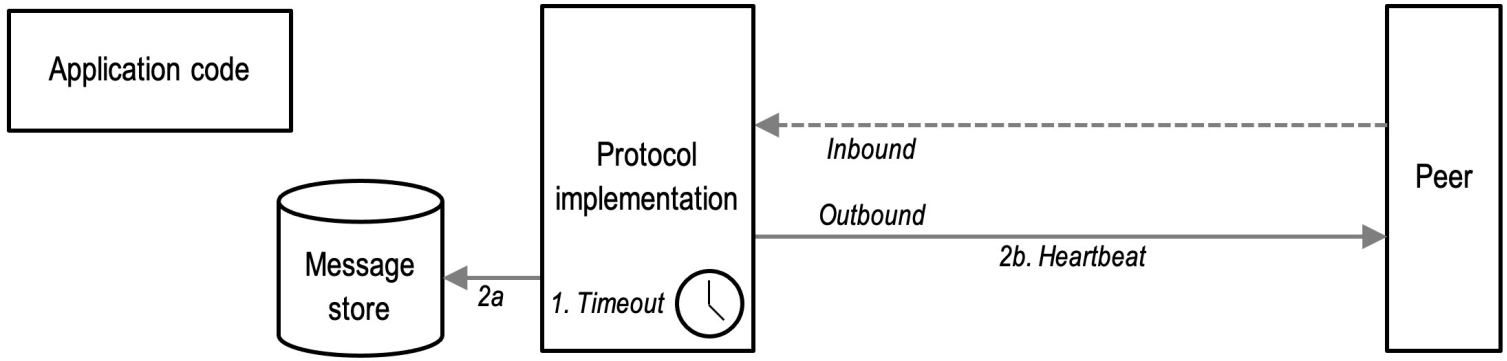
Application code responds to message from peer



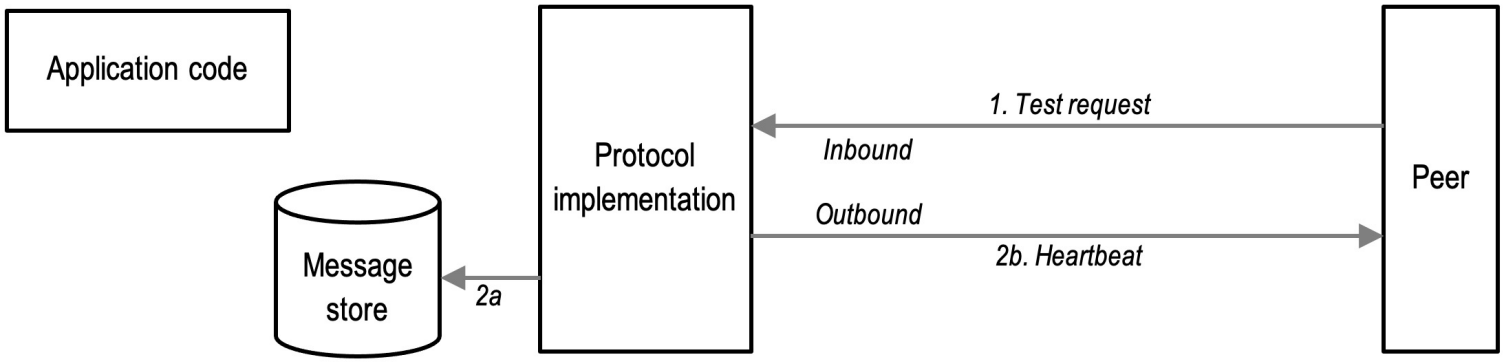
No message received from peer for a period, generate a TestRequest



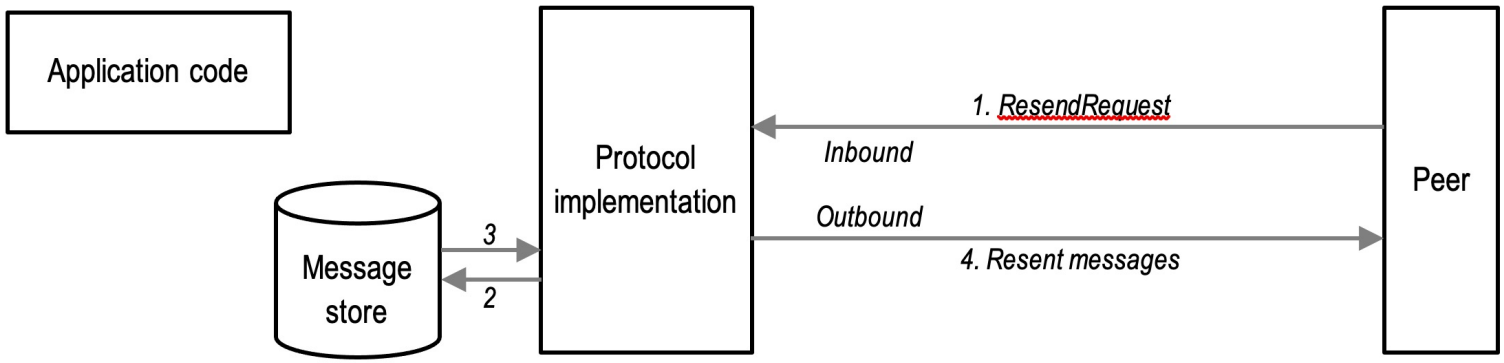
No message sent to peer for a period, generate a Heartbeat



TestRequest received from peer, respond with Heartbeat



ResendRequest received from peer, asynchronously retrieve stored messages and resend

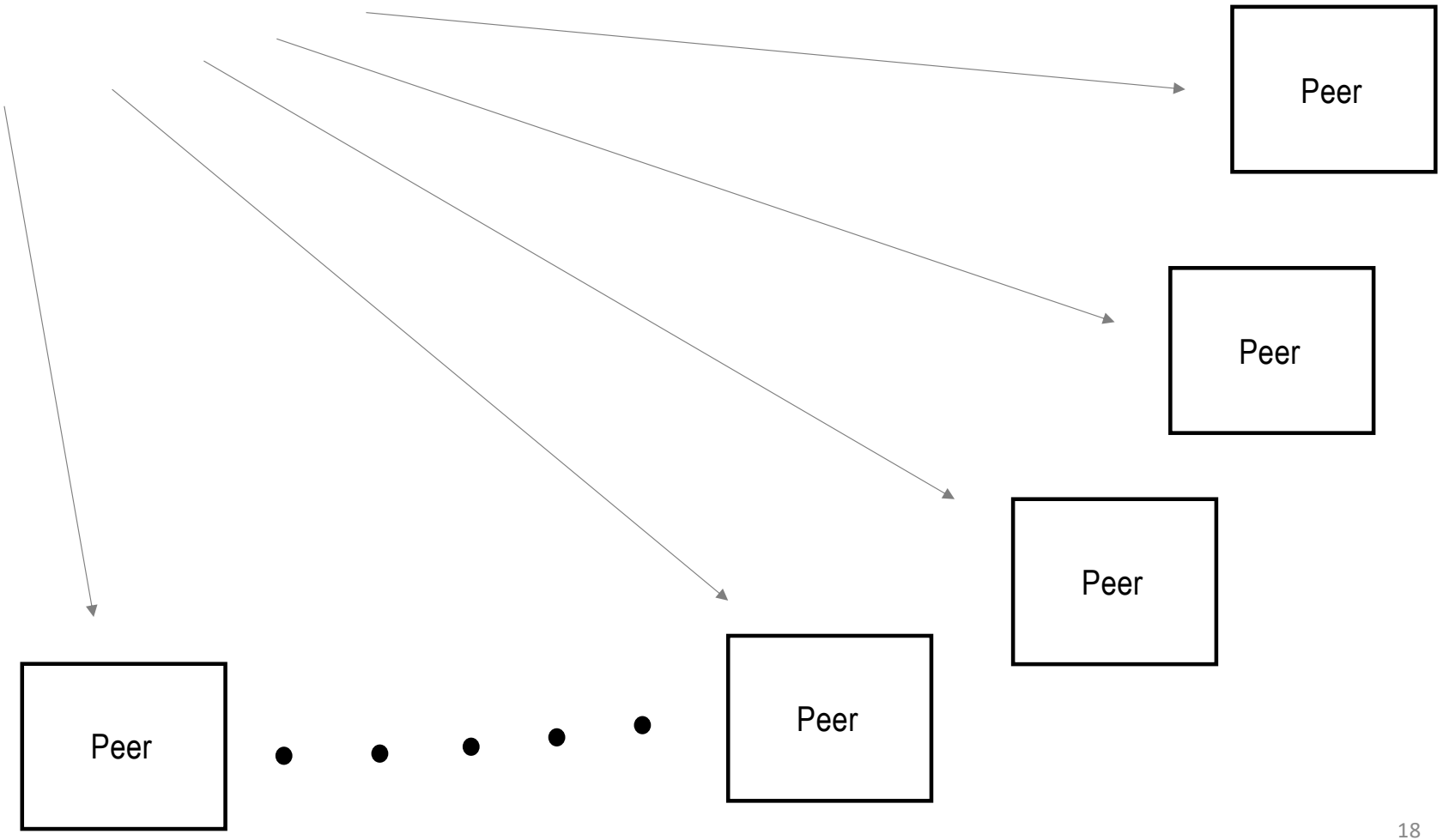


```
awaitable<void> transfer(tcp::socket& from, tcp::socket& to, steady_clock::time_point& deadline);
awaitable<void> watchdog(steady_clock::time_point& deadline);

awaitable<void> proxy(tcp::socket client, tcp::endpoint target)
{
  tcp::socket server(client.get_executor());
  steady_clock::time_point deadline{};

  auto [e] = co_await server.async_connect(target, use_nothrow_awaitable);
  if (!e)
  {
    co_await (
        transfer(client, server, deadline) ||
        transfer(server, client, deadline) ||
        watchdog(deadline)
      );
  }
}
```

```
awaitable<void> transfer(tcp::socket& from, tcp::socket& to, steady_clock::time_point& deadline)
{
  std::array<char, 1024> data;

  for (;;)
  {
    deadline = std::max(deadline, steady_clock::now() + 5s);

    auto [e1, n1] = co_await from.async_read_some(buffer(data), use_nothrow_awaitable);
    if (e1)
      co_return;

    auto [e2, n2] = co_await async_write(to, buffer(data, n1), use_nothrow_awaitable);
    if (e2)
      co_return;
  }
}

awaitable<void> watchdog(steady_clock::time_point& deadline)
{
  steady_timer timer(co_await this_coro::executor);

  auto now = steady_clock::now();
  while (deadline > now)
  {
    timer.expires_at(deadline);
    co_await timer.async_wait(use_nothrow_awaitable);
    now = steady_clock::now();
  }
}
```

```
tcp::socket selected(std::variant<tcp::socket, tcp::socket> v)
{
  switch (v.index())
  {
  case 0: return std::move(std::get<0>(v));
  case 1: return std::move(std::get<1>(v));
  default: throw std::logic_error(__func__);
  }
}

awaitable<tcp::socket> connect(ip::tcp::endpoint ep)
{
  auto sock = tcp::socket(co_await this_coro::executor);
  co_await sock.async_connect(ep, use_awaitable);
  co_return std::move(sock);
}

awaitable<tcp::socket> connect_range(tcp::resolver::results_type::const_iterator first, tcp::resolver::results_type::const_iterator last)
{
  auto next = std::next(first);
  if (next == last)
    co_return co_await connect(first->endpoint());
  else
    co_return selected(co_await(connect(first->endpoint()) || connect_range(next, last)));
}

awaitable<tcp::socket> connect_by_name(std::string host, std::string service)
{
  auto resolver = tcp::resolver(co_await this_coro::executor);
  auto results = co_await resolver.async_resolve(host, service, use_awaitable);
  co_return co_await connect_range(results.begin(), results.end());
}
```

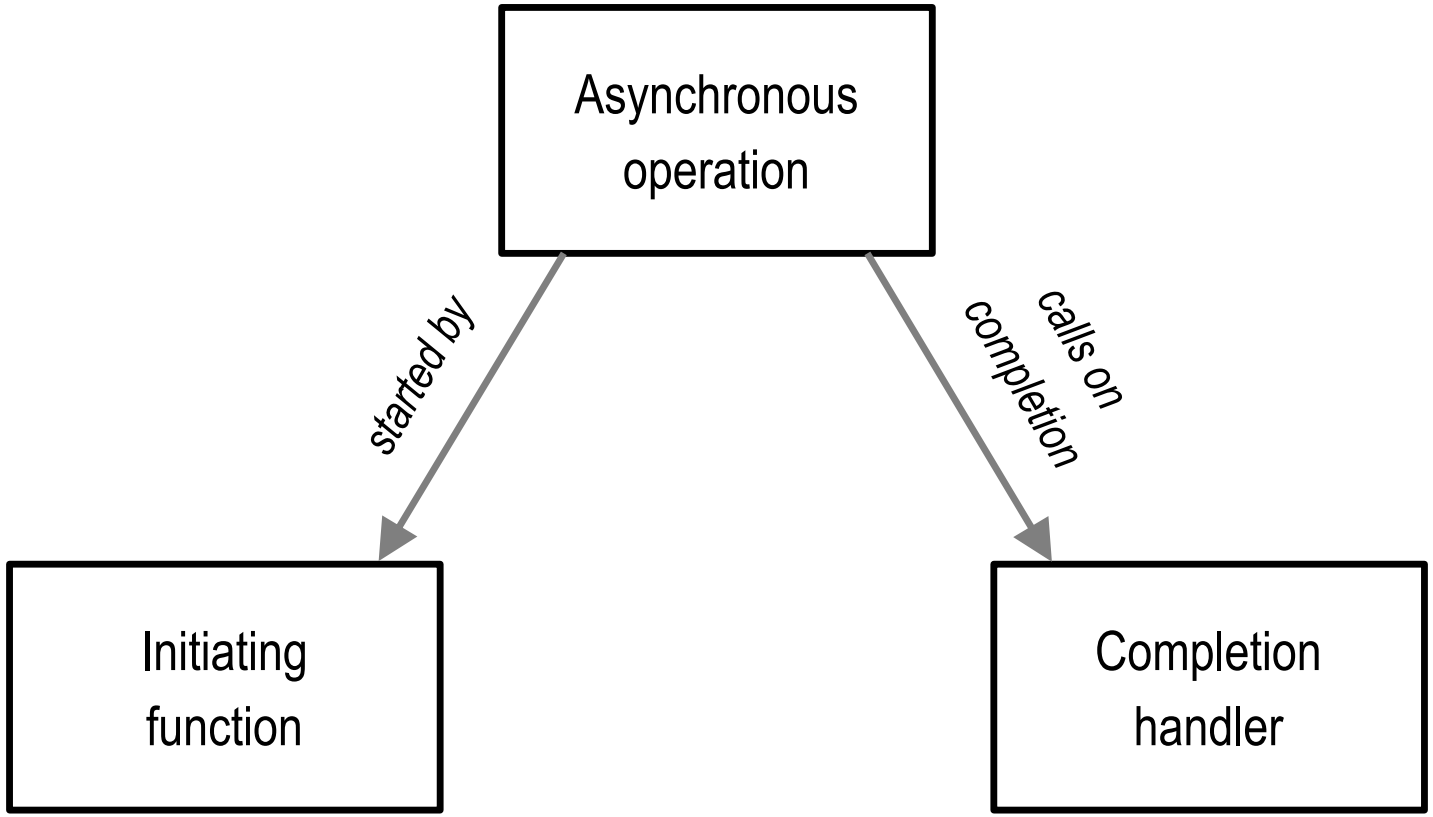
```
transmit(const shared_ptr<message>& m);
```

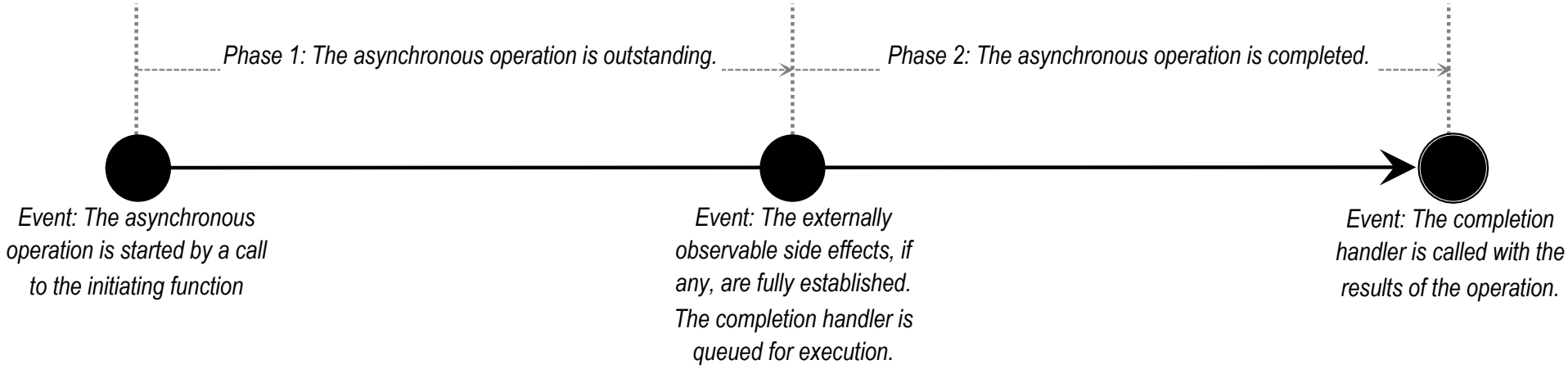


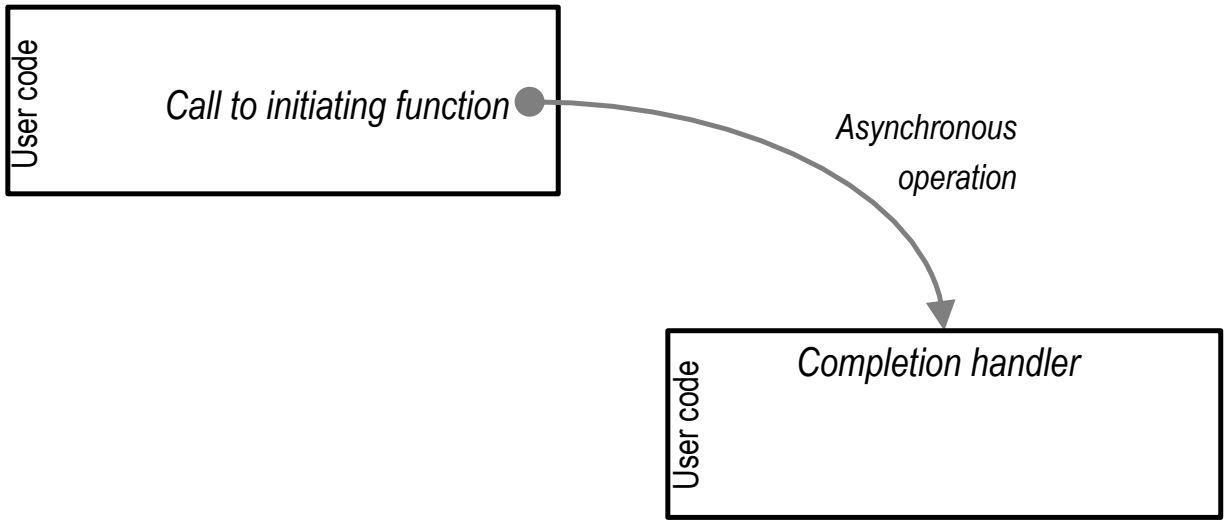
Design philosophy

- Be flexible in supporting composition mechanisms, since the appropriate choice depends on the specific use case.
- Aim to support as many of the semantic and syntactic properties of synchronous operations as possible, since they enable simpler composition and abstraction.
- Application code should be largely shielded from the complexity of threads and synchronisation, due to the complexity of handling events from different sources.

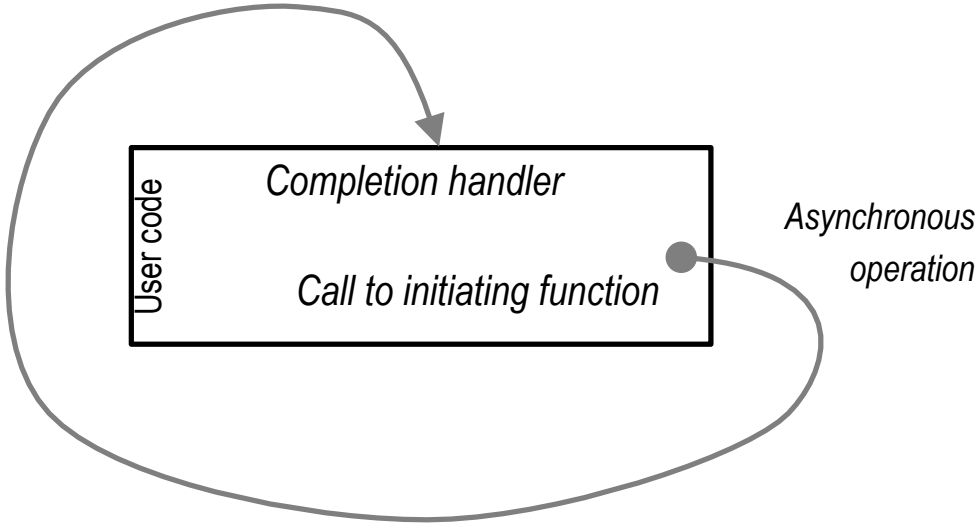
Asynchronous operations

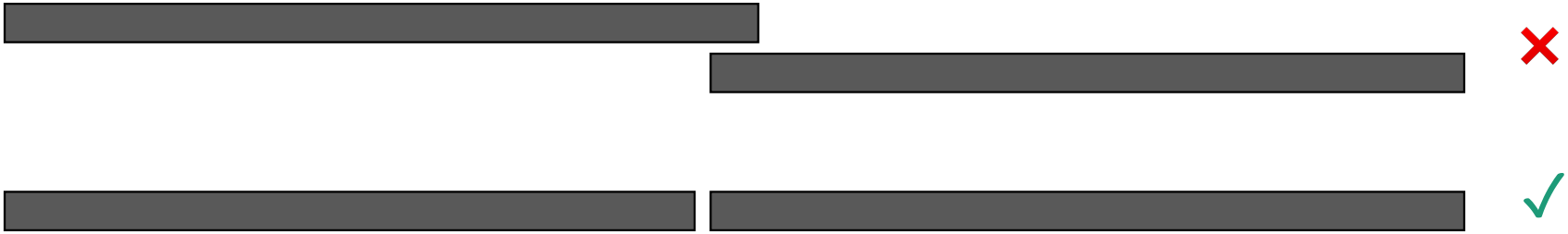



Phase 1: The asynchronous operation is outstanding.
Phase 2: The asynchronous operation is completed.
Event: The asynchronous operation is started by a call to the initiating function
Event: The externally observable side effects, if any, are fully established. The completion handler is queued for execution.
Event: The completion handler is called with the results of the operation.

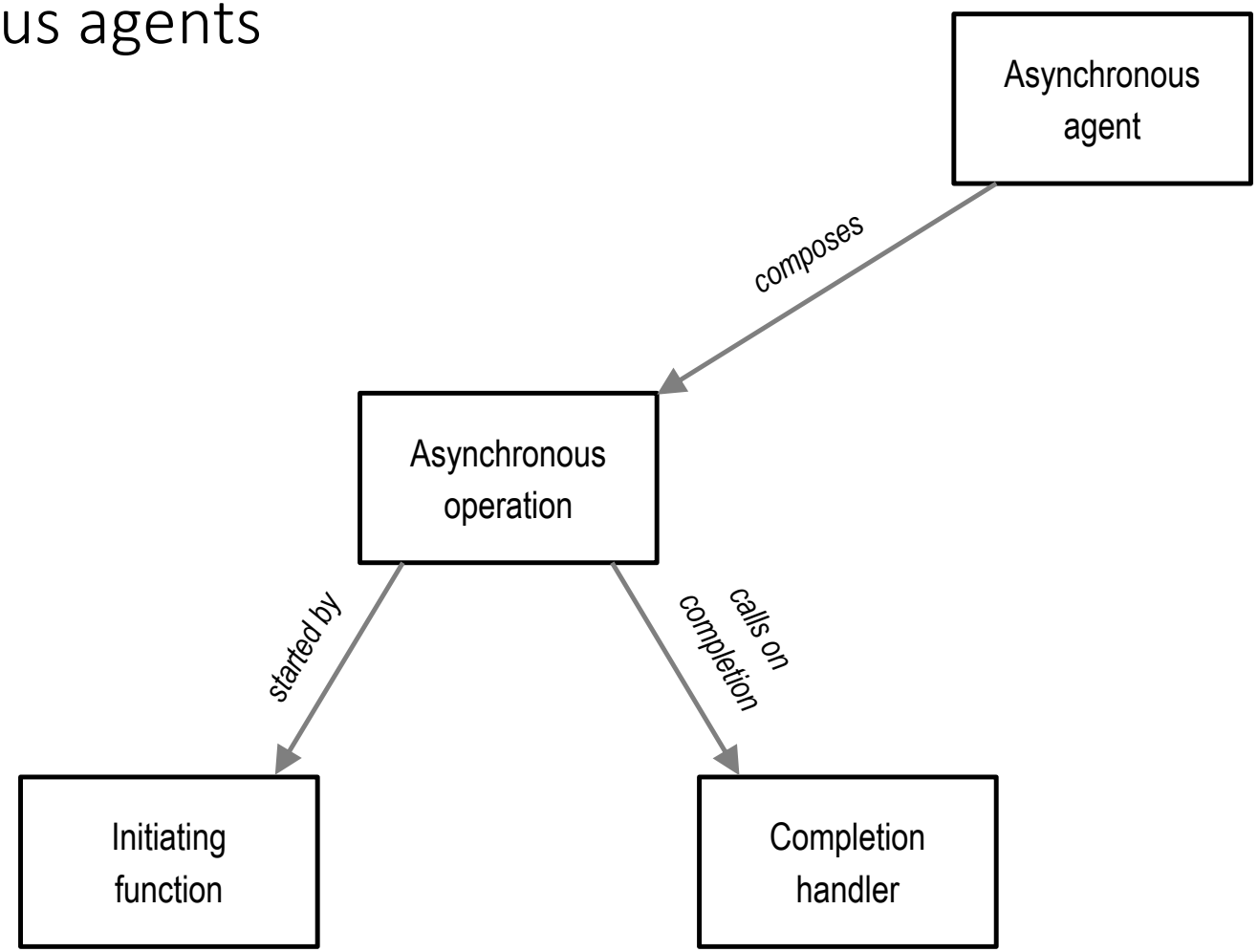

User code
Call to initiating function
Asynchronous
operation
Completion handler
User code

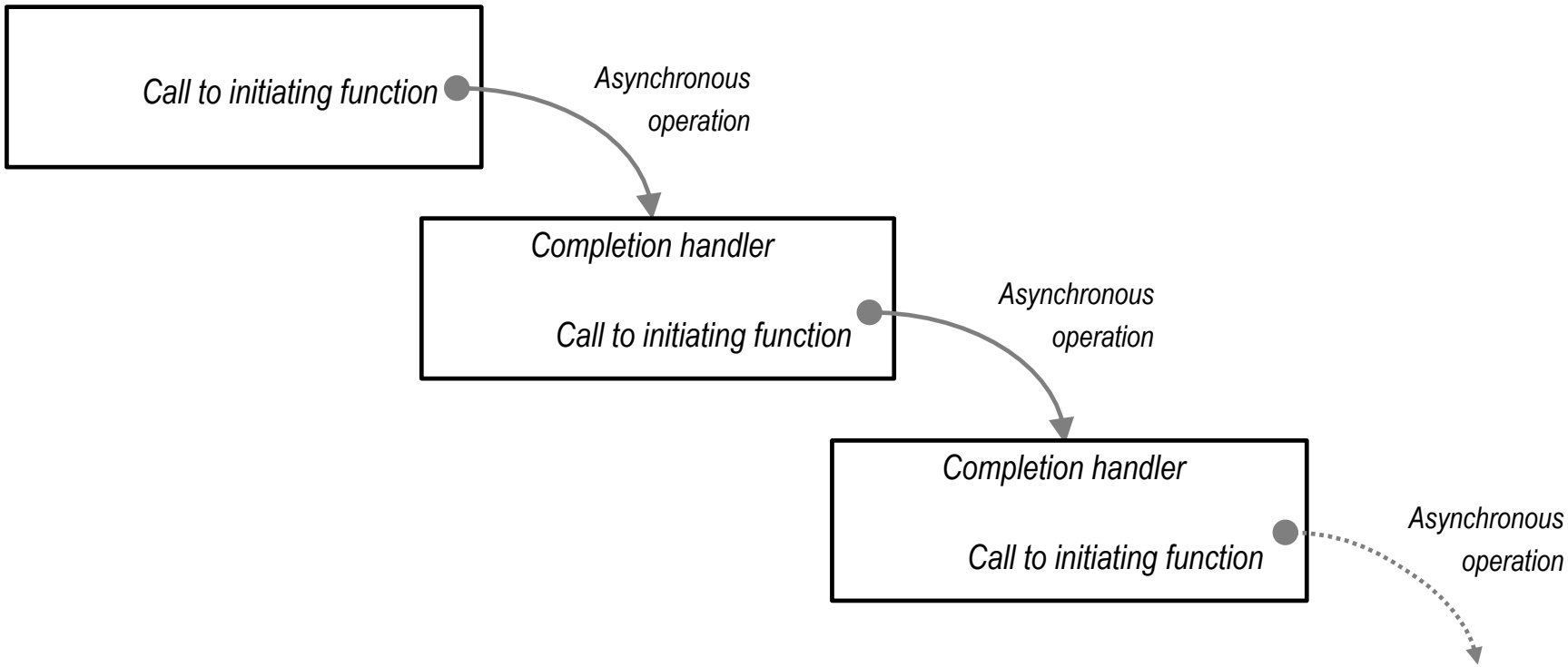
Property of synchronous operations	**Equivalent property of asynchronous operations**
When a synchronous operation is generic (i.e. a template) the return type is deterministically derived from the function and its arguments.	When an asynchronous operation is generic, the completion handler's arguments' types and order are deterministically derived from the initiating function and its arguments.
If a synchronous operation requires a temporary resource (such as memory, a file descriptor, or a thread), this resource is released before returning from the function.	If an asynchronous operation requires a temporary resource (such as memory, a file descriptor, or a thread), this resource is released before calling the completion handler.


User code
Completion handler
Call to initiating function
Asynchronous
operation

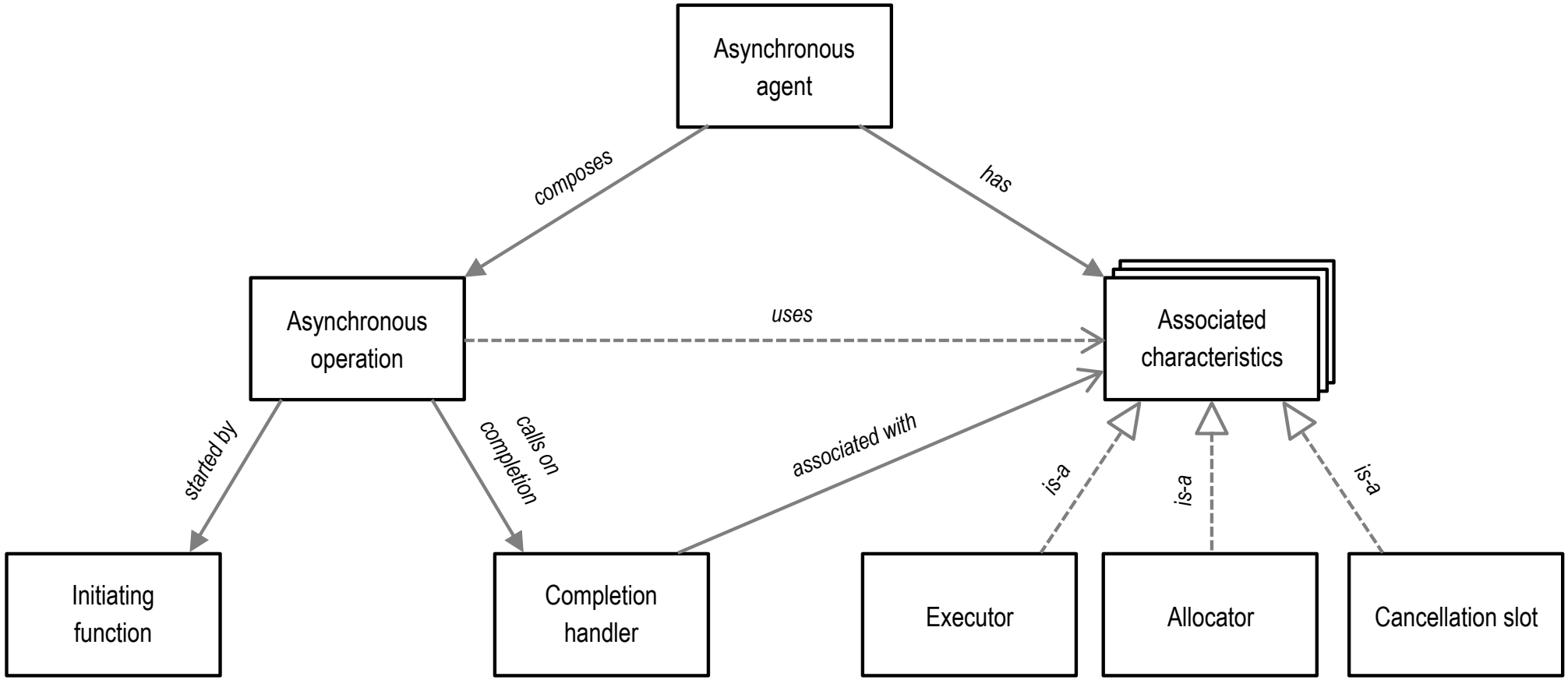


Asynchronous agents




Call to initiating function
Asynchronous operation
Completion handler
Call to initiating function
Asynchronous operation
Completion handler
Call to initiating function
Asynchronous operation

Associated characteristics and associators



An asynchronous operation uses an associator trait to compute:

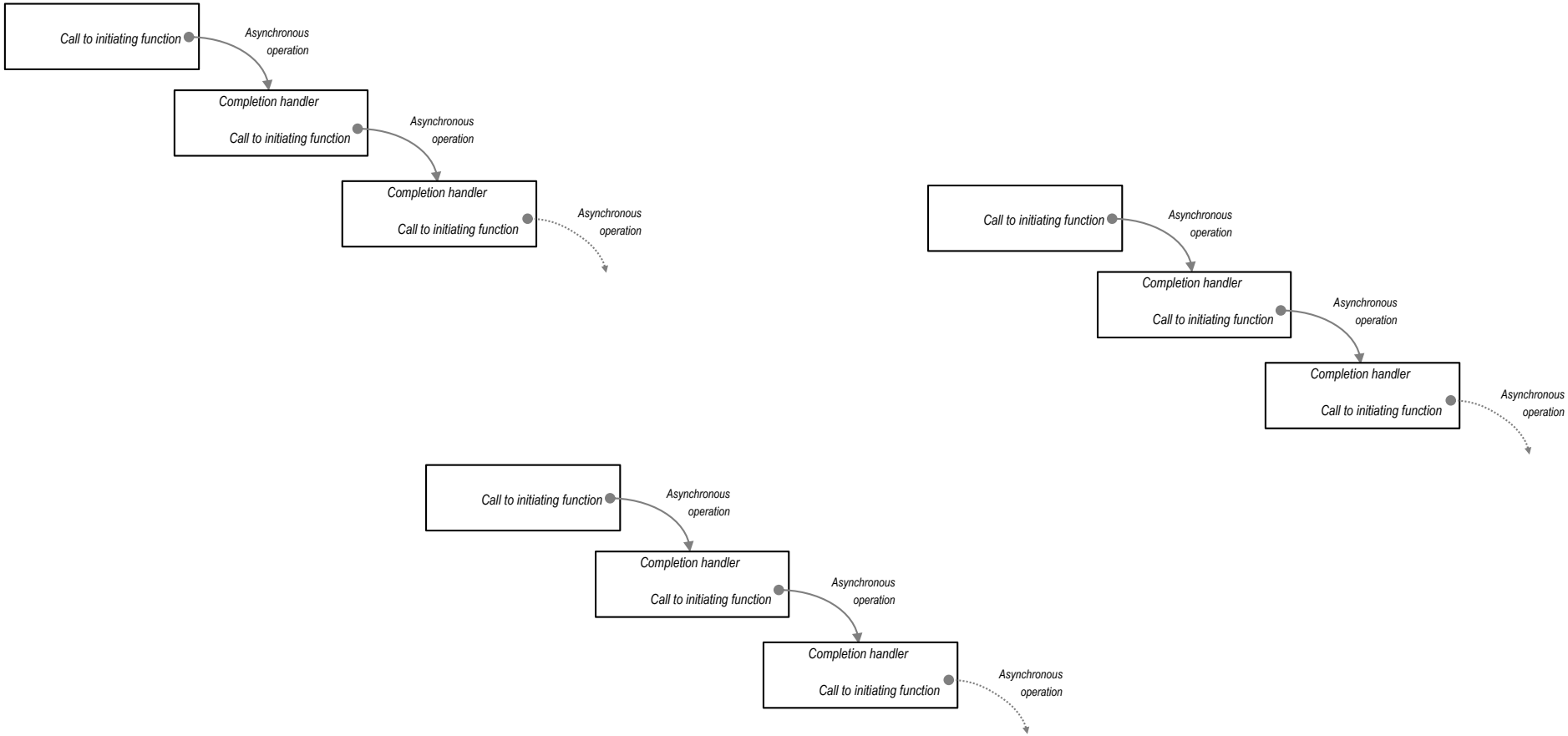
- the type `associated_R<S, C>::type`, and
- the value `associated_R<S, C>::get(s, c)`

```
using my_executor_type = associated_executor_t<handler_type, system_executor>;

auto my_executor = get_associated_executor(handler, system_executor{}),
```

Associated executors

An interface for coordinating how, when and where completion handlers run. Example uses:

- Coordinating a group of asynchronous agents that operate on shared data structures.
- Ensuring that agents are run on specified CPU.
- Denoting a group of related agents.
- Queuing all completion handlers to run on a GUI application thread.
- Running completion handlers as close as possible to the event that triggered the operation's completion.
- Adapting an executor to run code before and after every completion handler, such as logging, user authorisation, or exception handling.
- Specifying a priority for an asynchronous agent and its completion handlers.


Call to initiating function
Asynchronous operation
Completion handler
Call to initiating function
Asynchronous operation
Completion handler
Call to initiating function
Asynchronous operation
Call to initiating function
Asynchronous operation
Completion handler
Call to initiating function
Asynchronous operation
Completion handler
Call to initiating function
Asynchronous operation
Call to initiating function
Asynchronous operation
Completion handler
Call to initiating function
Asynchronous operation
Completion handler
Call to initiating function
Asynchronous operation

```
awaitable<void> echo(tcp::socket s)
{
  try
  {
    char data[1024];
    for (;;)
    {
      std::size_t n = co_await s.async_read_some(buffer(data), use_awaitable);
      co_await async_write(s, buffer(data, n), use_awaitable);
    }
  }
  catch (const std::exception& e)
  {
  }
}

awaitable<void> listen(tcp::acceptor a)
{
  for (;;)
  {
    co_spawn(a.get_executor(), echo(co_await a.async_accept(use_awaitable)), detached);
  }
}

int main()
{
  asio::io_context ctx;
  co_spawn(ctx, listen({ctx, {tcp::v4(), 55555}}), detached);
  ctx.run();
}
```

Associated allocators

An interface for obtaining *per operation stable memory* (POSMs).
Example uses:

- No POSMs required.
- A single, fixed-size POSM for as long as the operation is outstanding.
- A single, runtime-sized POSM.
- Multiple POSMs used concurrently, mixing fixed and runtime sized.
- The operation uses multiple POSMs serially, varying in size.

Associated cancellation slots

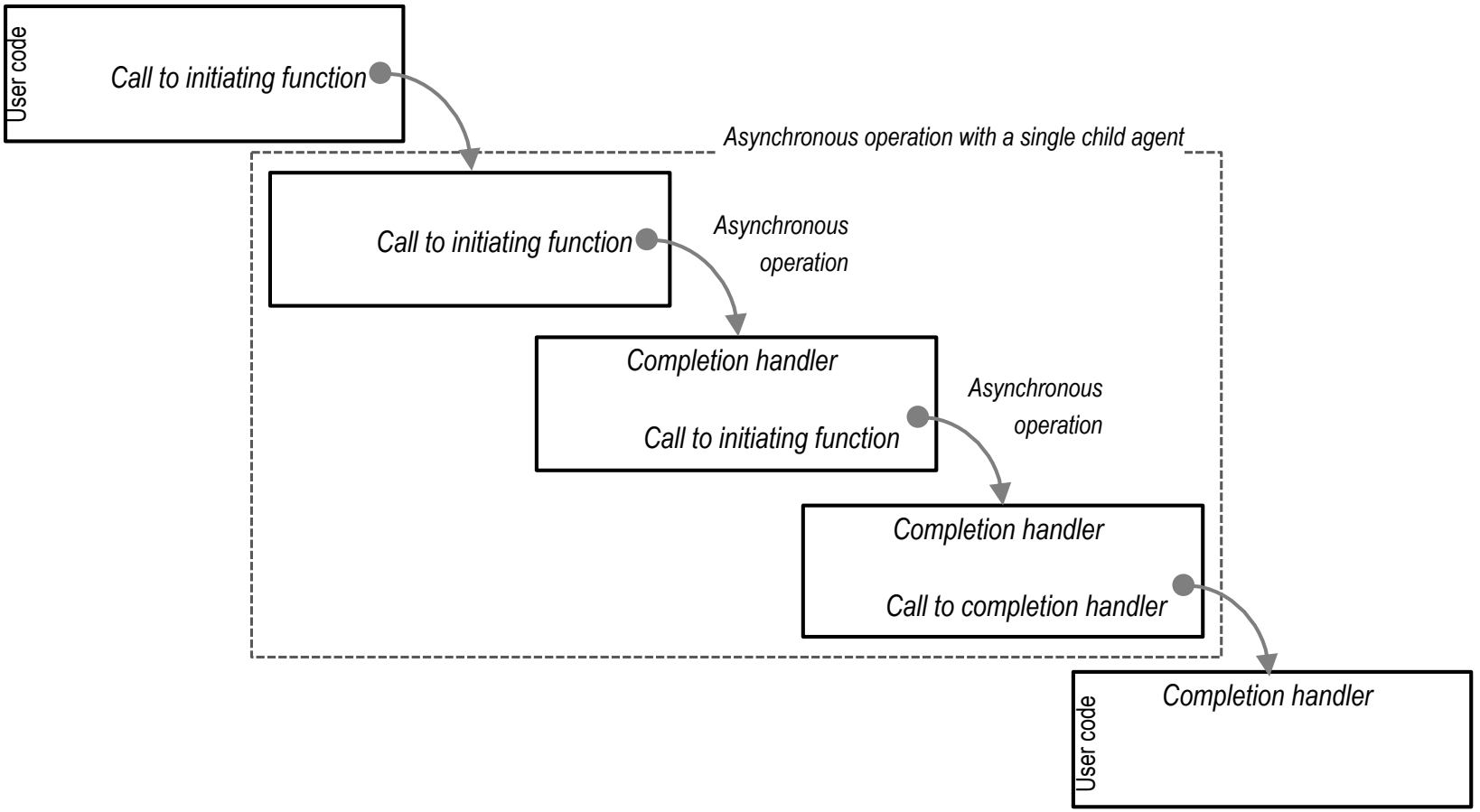
An interface for enabling per-operation cancellation.

Possible uses:

- Fine grained timeouts.
- Managing concurrent asynchronous agents.

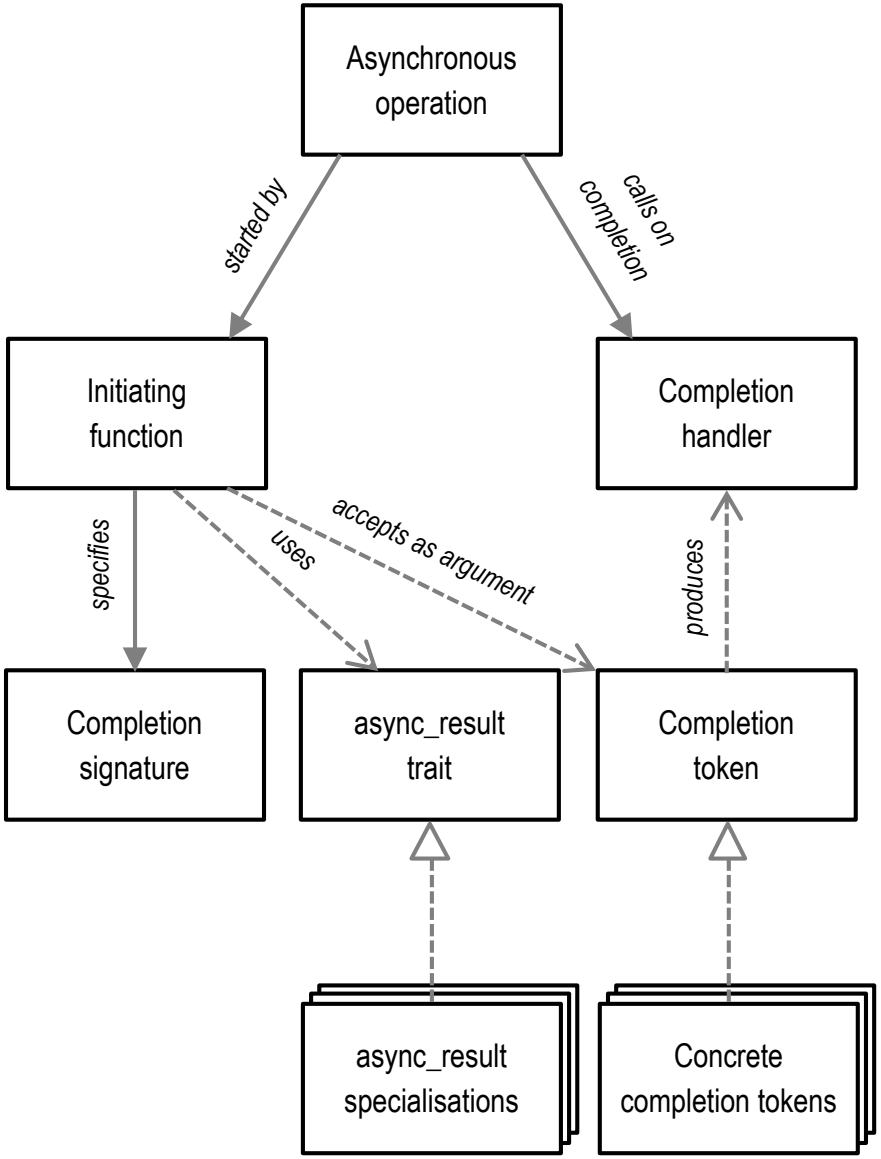
```
awaitable<void> transfer(tcp::socket& from, tcp::socket& to, steady_clock::time_point& deadline);
awaitable<void> watchdog(steady_clock::time_point& deadline);

awaitable<void> proxy(tcp::socket client, tcp::endpoint target)
{
  tcp::socket server(client.get_executor());
  steady_clock::time_point deadline{};

  auto [e] = co_await server.async_connect(target, use_nothrow_awaitable);
  if (!e)
  {
    co_await (
        transfer(client, server, deadline) ||
        transfer(server, client, deadline) ||
        watchdog(deadline)
      );
  }
}
```

Child agents



Property of synchronous operations	**Equivalent property of asynchronous operations**
Compositions of synchronous operations may be refactored to use child functions that run on the same thread (i.e. are simply called) without altering functionality.	Asynchronous agents may be refactored to use asynchronous operations and child agents that share the associated characteristics of the parent agent, without altering functionality.

Completion tokens



The same asynchronous operation used with a lambda…

```
socket.async_read_some(buffer,
    [](error_code e, size_t)
    {
      // ...
    }
  );
```

… with a future …

```
future<size_t> f =
  socket.async_read_some(
      buffer, use_future
    );

// ...

size_t n = f.get();
```

… in a coroutine …

```
awaitable<void> foo()
{
  size_t n =
    co_await socket.async_read_some(
        buffer, use_awaitable
      );

  // ...
}
```

… or in a fiber.

```
void foo()
{
  size_t n = socket.async_read_some(
      buffer, fibers::yield
    );

  // ...
}
```

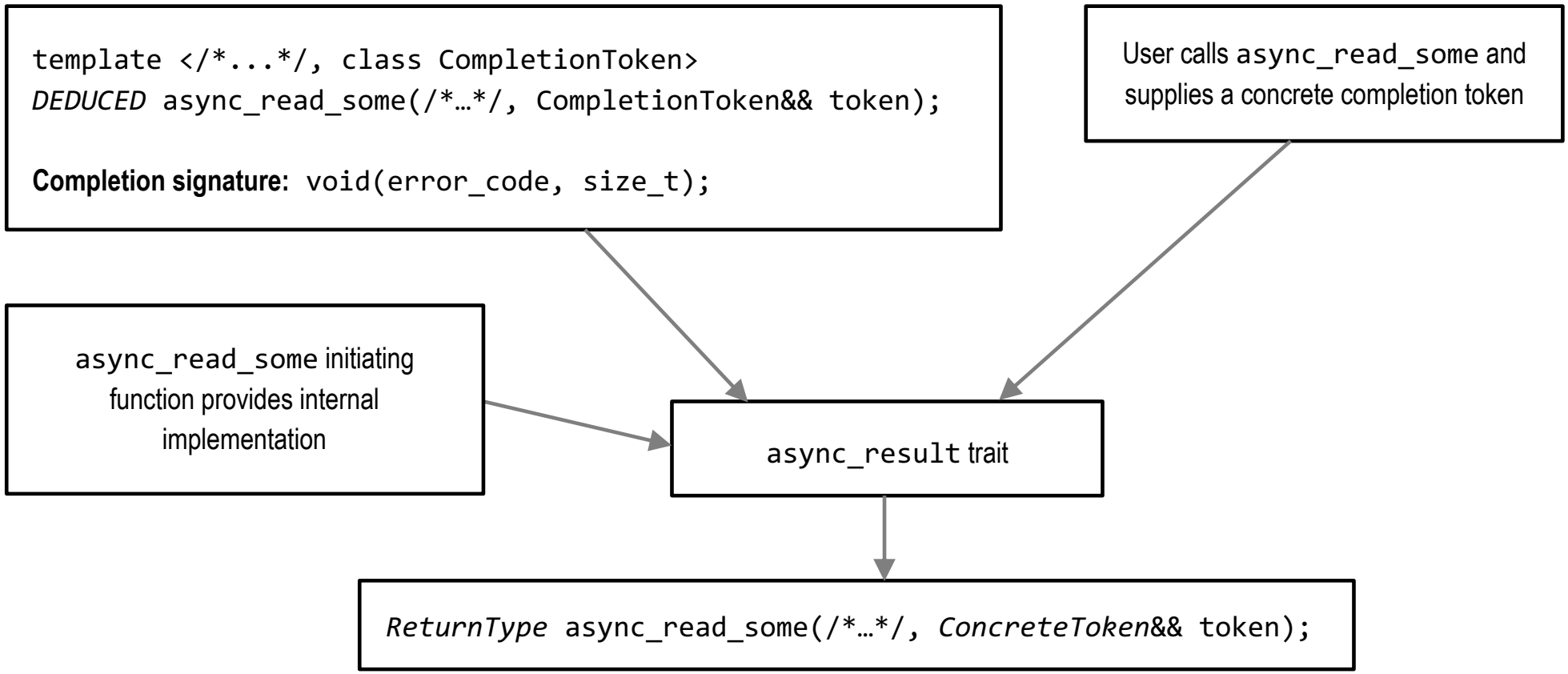
… or as a chain of "then" continuations …

```
auto f = socket.async_read_some(buffer(data), use_then)
  .then([&](error_code e, size_t n) {
    return async_write(socket, buffer(data, n), use_then);
  })
  .then([&](error_code e, size_t n) {
    return async_write(socket, buffer("\r\n", 2), use_then);
  })
 .then(use_future);
```

… or as an argument to a generic algorithm …

```
co_await make_linked_group(
    socket.async_read_some(s, buffer(data), deferred),
    copy(device, data.begin(), data.end(), device_data.begin(), deferred),
    checksum_chunks(device, device_data, device_results, deferred),
    reduce_results(device, device_results, deferred),
    copy(device, device_results.begin(), device_results.end(), results.begin(), deferred),
    async_write(s, buffer(results), deferred)
  ).async_wait(use_awaitable);
```

… or ???


template </*...*/, class CompletionToken>
DEDUCED async_read_some(/*…*/, CompletionToken&& token);
Completion signature: void(error_code, size_t);
User calls async_read_some and supplies a concrete completion token
async_read_some initiating function provides internal implementation
async_result trait
ReturnType async_read_some(/*…*/, ConcreteToken&& token);

```
template <class CompletionToken>
auto async_read_input(const std::string& prompt, CompletionToken&& token)
{
  auto init = [](auto handler, const std::string& prompt)
  {
    auto ex = asio::prefer(
        get_associated_executor(handler, system_executor{}),
        execution::blocking.possibly,
        execution::outstanding_work.tracked
      );

    auto cb = [ex, handler = std::move(handler)](std::string input) mutable
    {
      execution::execute(
          ex,
          [handler = std::move(handler), input = std::move(input)]() mutable
          {
            std::move(handler)(std::error_code{}, std::move(input));
          }
        );
    };

    read_input(prompt, std::move(cb));
  };

  return async_result<
      std::decay_t<CompletionToken>,
      void(std::error_code, std::string)
    >::initiate(init, std::forward<CompletionToken>(token), prompt);
}
```

```
async_read_input(
    "enter your name",
    [](std::error_code error, std::string input)
    {
      if (!error)
      {
        std::cout << "your name is: " << input << "\n";
      }
    }
  );
```

```
awaitable<void> send_name()
{
  tcp::socket socket = co_await connect_by_name("localhost", "55555");
  std::string input = co_await async_read_input("enter your name", use_awaitable);
  co_await async_write(socket, buffer(input), use_awaitable);
}
```

```
template <class CompletionToken, completion_signature... Signatures>
struct async_result
{
  template <
      class Initiation,
      completion_handler_for<Signatures...> CompletionHandler,
      class... Args>
  static void initiate(
      Initiation&& initiation,
      CompletionHandler&& completion_handler,
      Args&&... args)
  {
    std::forward<Initiation>(initiation)(
        std::forward<CompletionHandler>(completion_handler),
        std::forward<Args>(args)...);
  }
};
```

```
constexpr struct use_then_t {} use_then;

template <completion_signature... Signatures>
struct async_result<use_then_t, Signatures...>
{
  template <class Initiation, class... Args>
  struct then_impl
  {
    Initiation initiation;
    std::tuple<Args...> arg_pack;

    template <completion_token_for<Signatures...> CompletionToken>
    auto then(CompletionToken&& token) &&
    {
      return std::apply(
          [&](auto&&... args)
          {
            return async_result<decay_t<CompletionToken>, Signatures...>::initiate(
                std::move(initiation),
                std::forward<CompletionToken>(token),
                std::forward<decltype(args)>(args)...
              );
          },
          std::move(arg_pack)
        );
    }
  };

  template <class Initiation, class... Args>
  static auto initiate(Initiation initiation, use_then_t, Args... args)
  {
    return then_impl<Initiation, Args...>{
        .initiation = std::move(initiation),
        .arg_pack = std::make_tuple(std::move(args)...)
      };
  }
};
```

```
template <class R, class... ResultArgs>
struct async_result<simple_awaitable, R(ResultArgs...)>
{
  template <class Init, class... InitArgs>
  static auto initiate(Init init, simple_awaitable, InitArgs... init_args)
  {
    struct awaitable
    {
      Init init_;
      std::tuple<InitArgs...> init_args_;
      std::tuple<ResultArgs...>* result_;

      bool await_ready() const noexcept { return false; }
      std::tuple<ResultArgs...> await_resume() { return std::move(*result_); }

      void await_suspend(coroutine_handle<> handle)
      {
        std::apply(
            init_,
            std::tuple_cat(
              std::tuple(
                [this, handle](ResultArgs... result_args) mutable
                {
                  std::tuple<ResultArgs...> result(std::move(result_args)...);
                  result_ = &result;
                  handle.resume();
                }
              ),
              std::move(init_args_)
            )
          );
      }
    };

    return awaitable{std::move(init), {std::move(init_args)...}, nullptr};
  }
};
```

Library summary

- `completion_signature` concept – defines valid completion signature forms.
- `completion_handler_for` concept – determines whether a completion handler is callable with a given set of completion signatures.
- `async_result` trait – converts a completion signature and completion token into a concrete completion handler, and launches the operation.
- `async_initiate` function – helper function to simplify use of the async_result trait.
- `completion_token_for` concept – determines whether a completion token produces a handler for the given set of completion signatures.
- `associator` trait – automatically propagates all associators through layers of abstraction.
- `associated_executor` trait – defines an asynchronous agent's associated executor.
- `associated_executor_t` template type alias
- `get_associated_executor` function
- `associated_allocator` trait – defines an asynchronous agent's associated allocator.
- `associated_allocator_t` template type alias
- `get_associated_allocator` function
- `associated_cancellation_slot` trait – defines an asynchronous agent's associated cancellation slot.
- `associated_cancellation_slot_t` template type alias
- `get_associated_cancellation_slot` function

Enabling further abstraction