Document Number: P2444r0
Date: 2021-09-15
Reply To Christopher Kohlhoff <chris@kohlhoff.com>

The Asio asynchronous model

1 Introduction

The networking domain has long employed event-driven and asynchronous programming designs to develop efficient, scalable, network-oriented software. The use of a proactor-based event model, which consists of asynchronous operations with continuations, offers a good conceptual model for abstraction and composition. Asynchronous operations may be chained, with each continuation initiating the next operation. The composed operations may then be abstracted away behind a single, higher level asynchronous operation with its own continuation.

However, as asynchronous compositions grow, a purely callback-based approach can increase apparent code complexity and harm readability. Programmers reach for alternative composition mechanisms such as state machines, fibers, and, as of C++20, language-based coroutines to improve code clarity while retaining the benefits of an asynchronous implementation. There is no one-size-fits-all approach to composition.

This paper presents a high-level overview of the asynchronous model at the core of the Asio library. This model enshrines asynchronous operations as the fundamental building block of asynchronous composition, but in a way that decouples them from the composition mechanism. The asynchronous operations in Asio support callbacks, futures (both eager and lazy), fibers, coroutines, and approaches yet to be imagined. This allows the application programmer to select an approach based on appropriate trade-offs.

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
    );

  // ...
}
```

2 Motivation

2.1 Synchronous operations as inspiration

The simplest network programs sometimes employ a thread-per-connection approach. Take for example a basic echo server, written purely in terms of synchronous operations:

```
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}});
}
```

This structure and flow of this program is clear, as synchronous operations are just functions. This inherently imbues them with several beneficial syntactic and semantic properties, including:

- 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.

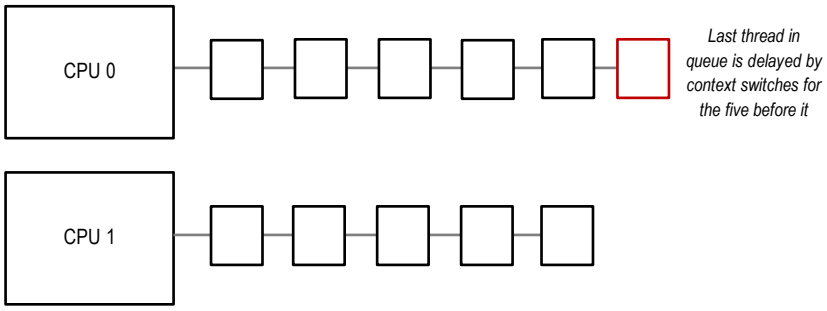
However, the use of a thread-per-connection approach has several issues that limits its applicability in general.

2.2 Limited scalability of threads

Thread-per-connection designs, as the name suggests, employ a separate thread to handle each connection. For servers that handle thousands, or perhaps millions, of concurrent connections this represents significant resource usage within the program, although in more recent years the widespread availability of 64-bit operating systems has mitigated this.

For performance sensitive use cases, however, the cost of context switching between threads may be a more important consideration. The cost of a context switch between general purpose OS threads is measured in

thousands of cycles. When runnable threads outnumber execution resources (like CPUs), queuing occurs and the last task to be queued is delayed by the cost of many context switches:



Even when a network server appears to be lightly loaded overall, temporal correlation of events can still produce queuing. For example, in financial markets all participants are processing and responding to the same market data streams, and consequently it is highly likely that more than one participant will respond to the same stimulus by sending a transaction to the server. This queueing increases the average latency and jitter experienced by the participants.

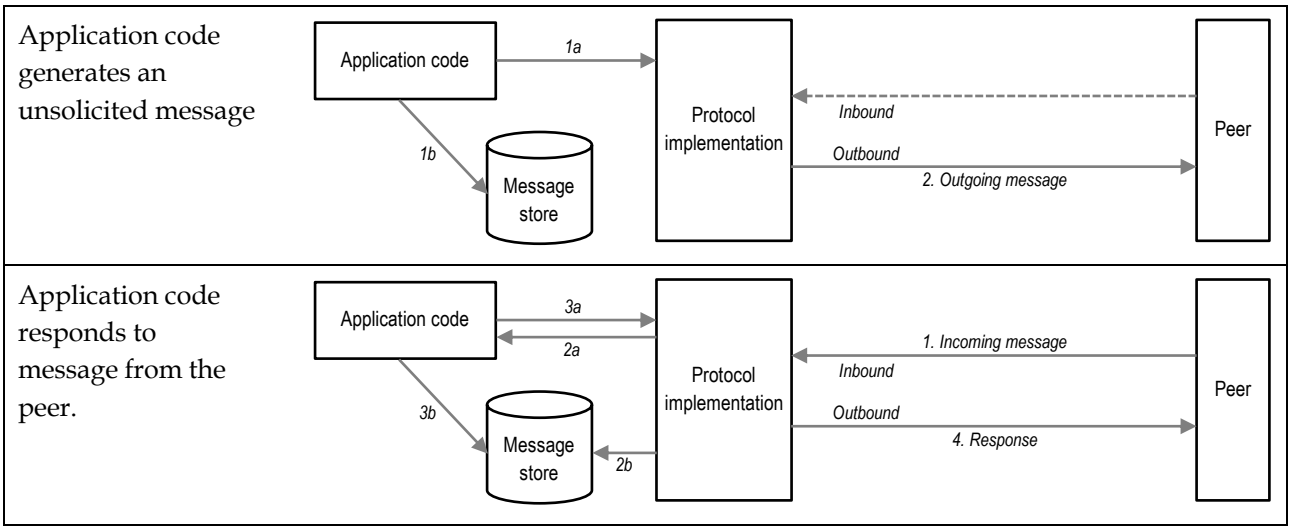
For comparison, a scheduler designed specifically for event handling can "context switch" between tasks one-to-two orders of magnitude faster, in tens to hundreds of cycles. Queueing may still occur, but the overhead associated with processing the queue is substantially reduced.

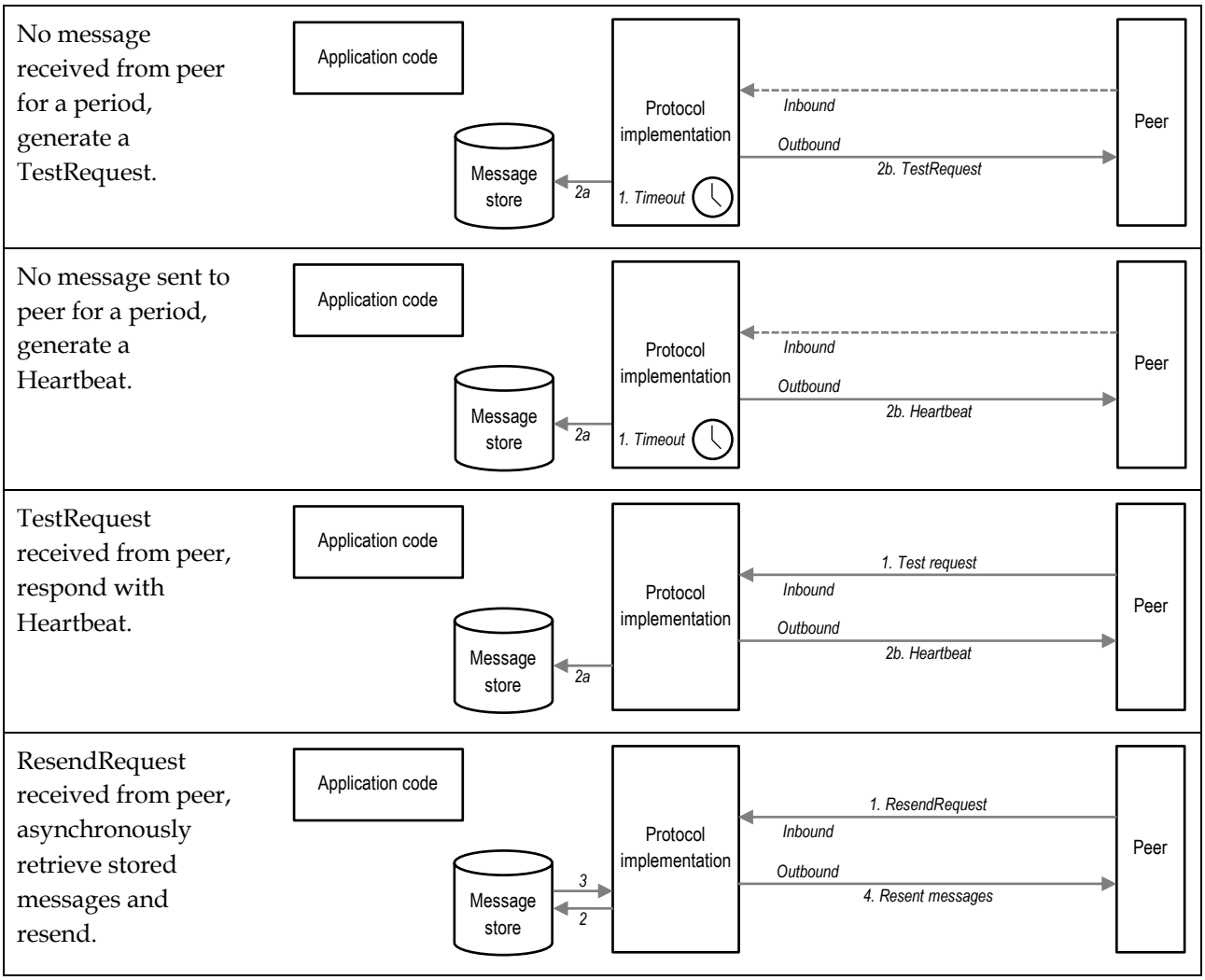
Finally, we must also note that our thread-per-connection echo server was extremely simple in that each thread, once launched, is able to operate independently. In real-world use cases, the server program may need to access shared data to respond to clients, resulting in synchronisation costs, data movement between CPUs, and increased code complexity.

2.3 Half-duplex vs full-duplex protocols

A thread-per-connection approach may be suitable for simple protocols like the echo server shown above, as this is a half-duplex protocol. The server is either sending or receiving, but never both at the same time.

However, many real-world application protocols are full duplex, meaning that data may be transmitted in either direction at any time. As an example, consider some of the message exchange scenarios applicable to the FIX protocol:





As you can see, protocols like these necessitate responding to events from many different sources. This has several implications:

- Different parts of the protocol logic, that are executing concurrently, may need to access shared state.
- The complex event handling may not be easily represented in linear form (such as that enabled by a thread-per-connection design, or indeed even when using coroutines).

Consequently, we often find authors of these protocols utilising other composition mechanisms, such as state machines, as a way to manage the complexity and ensure correctness.

2.4 Eager execution matters for performance

Some network applications require the delivery of a single message to many consumers. One such example is the dissemination of market data, in real-time, to all participants. When delivering this information asynchronously, a common approach is to wrap the message in a reference counted pointer (such as `shared_ptr`) that keeps the memory valid until it has been transmitted to all.

However, for efficiency, each of these transmission operations attempts a speculative send. The happy path, which statistically occurs almost all the time (due to efforts in ensuring that all hardware and software is correctly sized based on expected loads), is that this speculative send succeeds and the data is transmitted immediately. If this occurs, it is not necessary to maintain a valid shared pointer any longer. This avoids the overhead of ticking the reference count up and down.

For comparison, atomic reference counting cost is measured in tens of cycles, compared to the transmission system call itself which is measured in the hundreds. Avoiding this additional cost can represent an efficiency gain of 5-10% in practice.

A lazy execution model is unable to avoid this cost, as it must copy the shared data when the operation is first invoked.

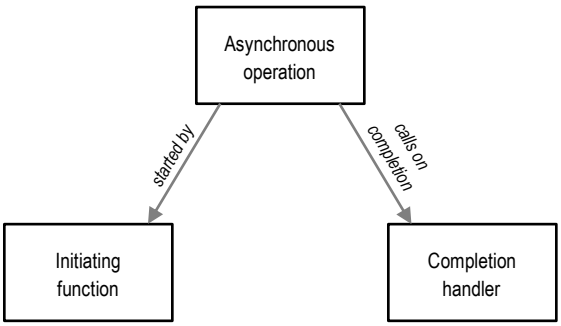
2.5 Design philosophy

The concerns described above motivate the following design philosophy for an asynchronous model:

- 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.

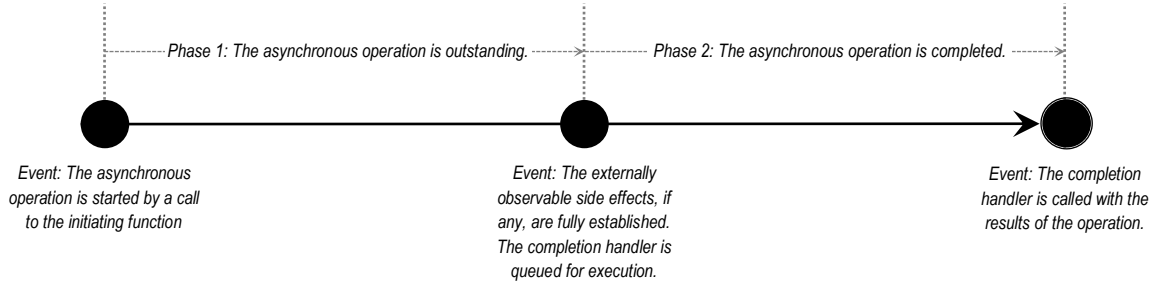
3 Model

3.1 Asynchronous operations



An *asynchronous operation* is the basic unit of composition in the Asio asynchronous model. Asynchronous operations represent work that is launched and performed in the background, while the user's code that initiated the work can continue with other things.

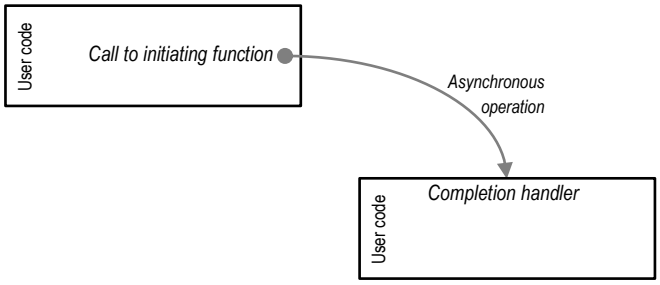
Conceptually, the lifecycle of an asynchronous operation can be described in terms of the following events and phases:



An *initiating function* is a function which may be called by the user to start an asynchronous operation.

A *completion handler* is a user-provided, move-only function object that will be invoked, at most once, with the result of the asynchronous operation. The invocation of the completion handler tells the user about something that has already happened: the operation completed, and the side effects of the operation were established.

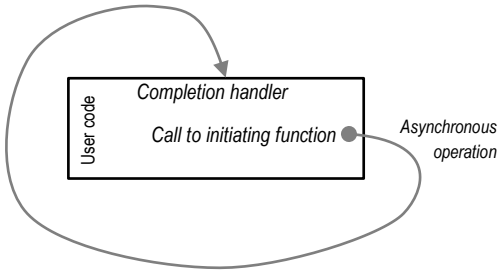
The initiating function and completion handler are incorporated into the user's code as follows:



Synchronous operations, being embodied as single functions, have several inherent semantic properties as a consequence. Asynchronous operations adopt some of these semantic properties from their synchronous counterparts, in order to facilitate flexible and efficient composition.

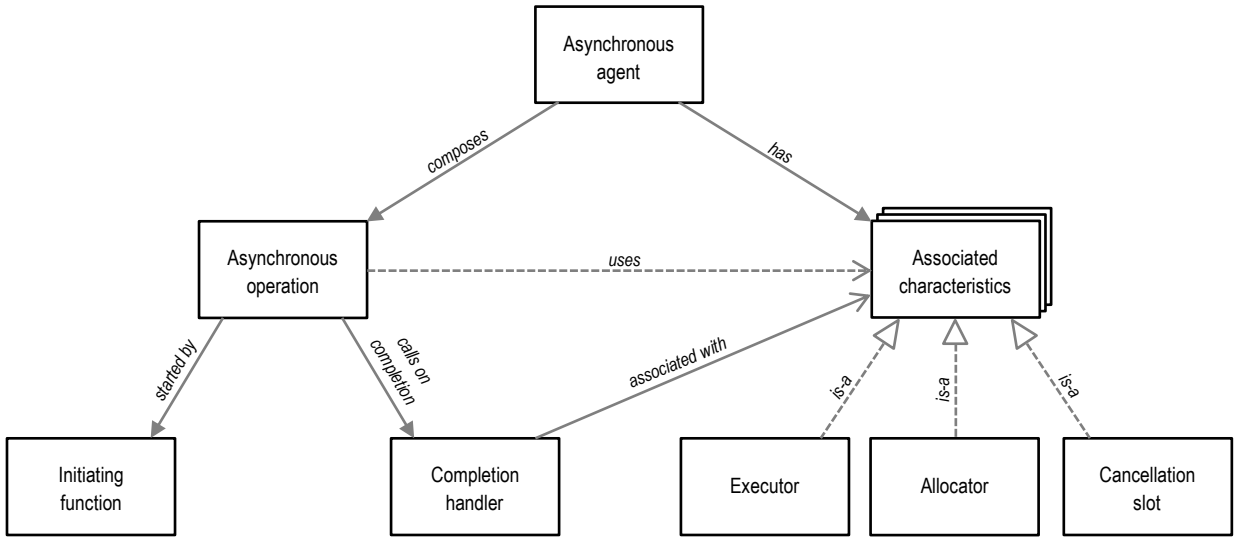
Property of synchronous operations	**Equivalent property of asynchronous operations**
When a synchronous operation is generic (i.e. a template) the return type is deterministically derived from the function and its arguments.	When an asynchronous operation is generic, the completion handler's arguments' 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.

The latter is an important property of asynchronous operations, in that it allows a completion handler to initiate further asynchronous operations *without* overlapping resource usage. Consider the trivial (and relatively common) case of the same operation being repeated over and over in a chain:



By ensuring that resources are released before the completion handler runs, we avoid doubling the peak resource usage of the chain of operations.

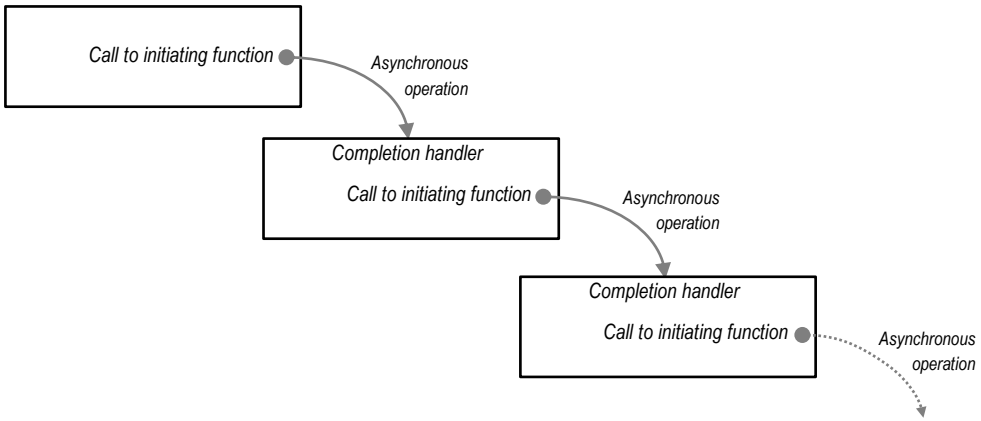
3.2 Asynchronous agents



An *asynchronous agent* is a sequential composition of asynchronous operations. Every asynchronous operation is considered to run as a part of an asynchronous agent, even if that agent contains only that single operation. An asynchronous agent is an entity that may perform work concurrently with other agents. Asynchronous agents are to asynchronous operations as threads are to synchronous operations.

However, an asynchronous agent is a purely notional construct that allows us to reason about the context for, and composition of, asynchronous operations in a program. The name "asynchronous agent" does not appear in the library, nor is it important which concrete mechanism[1] is used to compose the asynchronous operations in an agent.

We can visualise an asynchronous agent as follows:



Asynchronous agents alternately wait for an asynchronous operation to complete, and then run a completion handler for that operation. Within the context of an agent, these completion handlers represent indivisible units of schedulable work.

3.3 Associated characteristics and associators

An asynchronous agent has *associated characteristics* that specify how asynchronous operations should behave when composed as part of that agent, such as:

- An allocator, which determines how the agent's asynchronous operations obtain memory resources.
- A cancellation slot, which determines how the agent's asynchronous operations support cancellation.

[1] Such as chains of lambdas, coroutines, fibers, state machines, etc.

- An executor, which determines how the agent's completion handlers will be queued and run.

When an asynchronous operation is run within an asynchronous agent, its implementation may query these associated characteristics and use them to satisfy the requirements or preferences they represent. The asynchronous operation performs these queries by applying *associator* traits to the completion handler. Each characteristic has a corresponding associator trait.

An associator trait may be specialised for concrete completion handler types to:

- accept the default characteristic supplied by the asynchronous operation, returning this default as-is
- return an unrelated implementation of the characteristic, or
- adapt the supplied default to introduce additional behaviour required by the completion handler.

Specification of an associator

Given an associator trait named[2] `associated_R`, having:

- a source value `s` of type `S`, in this case the completion handler and its type,
- a set of type requirements (or a concept) `R` that define the syntactic and semantic requirements of the associated characteristic, and
- a candidate value `c` of type `C` that meets the type requirements `R`, which represents a default implementation of the associated characteristic, supplied by the asynchronous operation

the asynchronous operation uses the associator trait to compute:

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

that meet the requirements defined in `R`. For convenience, these are also accessible via type alias `associated_R_t<S, C>` and free function `get_associated_R(s, c)`, respectively.

The trait's primary template is specified such that:

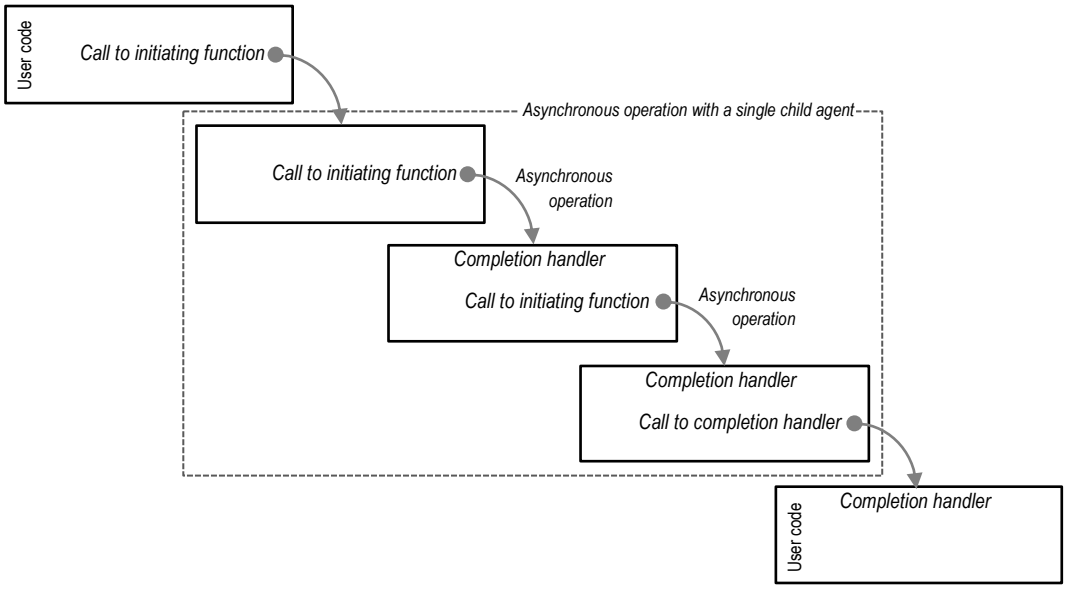
- if `S::R_type` is well-formed, defines a nested type alias `type` as `S::R_type`, and a static member function `get` that returns `s.get_R()`
- otherwise, if `associator<associated_R, S, C>::type` is well-formed and denotes a type, inherits from `associator<associated_R, S, C>`
- otherwise, defines a nested type alias `type` as `C`, and a static member function `get` that returns `c`.

3.4 Child agents

The asynchronous operations within an agent may themselves be implemented in terms of child agents.[3] As far as the parent agent is concerned, it is waiting for the completion of a single asynchronous operation. The asynchronous operations that constitute the child agent run in sequence, and when the final completion handler runs the parent agent is resumed.

[2] The associator traits are named `associated_allocator`, `associated_executor`, and `associated_cancellation_slot`.

[3] In Asio these asynchronous operations are referred to as *composed operations*.



As with individual asynchronous operations, the asynchronous operations built on child agents must release their temporary resources prior to calling the completion handler. We may also think of these child agents as resources that end their lifetimes before the completion handler is invoked.

When an asynchronous operation creates a child agent, it may propagate[4] the associated characteristics of the parent agent to the child agent. These associated characteristics may then be recursively propagated through further layers of asynchronous operations and child agents. This stacking of asynchronous operations replicates another property of synchronous operations.

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.

Finally, some asynchronous operations may be implemented in terms of multiple child agents that run concurrently. In this case, the asynchronous operation may choose to selectively propagate the associated characteristics of the parent agent.

3.5 Executors

Every asynchronous agent has an associated *executor*. An agent's executor determines how the agent's completion handlers are queued and ultimately run.

Example uses of executors include:

- Coordinating a group of asynchronous agents that operate on shared data structures, ensuring that the agents' completion handlers never run concurrently[5].
- Ensuring that agents are run on specified execution resource (e.g. a CPU) that is proximal to data or an event source (e.g. a NIC).
- Denoting a group of related agents, and so enabling dynamic thread pools to make smarter scheduling decisions (such as moving the agents between execution resources as a unit).
- Queuing all completion handlers to run on a GUI application thread, so that they may safely update user interface elements.

[4] Typically, by specialising the associator trait and forwarding to the outer completion handler.

[5] In Asio, this kind of executor is called a `strand`.

- Returning an asynchronous operation's default executor as-is, to run completion handlers as close as possible to the event that triggered the operation's completion.
- Adapting an asynchronous operation's default 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.

The asynchronous operations within an asynchronous agent use the agent's associated executor to:

- Track the existence of the work that the asynchronous operation represents, while the operation is outstanding.
- Enqueue the completion handler for execution on completion of an operation.
- Ensure that completion handlers do not run re-entrantly, if doing so might lead to inadvertent recursion and stack overflow.

Thus, an asynchronous agent's associated executor represents a policy of how, where, and when the agent should run, specified as a cross-cutting concern to the code that makes up the agent.

3.6 Allocators

Every asynchronous agent has an associated *allocator*. An agent's allocator is an interface used by the agent's asynchronous operations to obtain *per-operation stable memory* resources (POSMs). This name reflects the fact that the memory is per-operation because the memory is only retained for the lifetime of that operation, and stable, because the memory is guaranteed to be available at that location throughout the operation.

Asynchronous operations may utilise POSMs in a number of different ways:

- The operation doesn't require any POSMs. For example, the operation wraps an existing API that performs its own memory management, or is copying the long lived state into existing memory like a circular buffer.
- The operation uses a single, fixed-size POSM for as long as the operation is outstanding. For example, the operation stores some state in a linked list.
- The operation uses a single, runtime-sized POSM. For example, the operation stores a copy of a user-supplied buffer, or a runtime-sized array of `iovec` structures.
- The operation uses multiple POSMs concurrently. For example, a fixed size POSM for a linked list plus a runtime-sized POSM for a buffer.
- The operation uses multiple POSMs serially, which may vary in size.

Associated allocators allow users to treat POSM optimisation as a cross-cutting concern to the composition of asynchronous operations. Furthermore, using allocators as the interface to obtain POSMs grant substantial flexibility to both the implementers and users of asynchronous operations:

- Users can ignore the allocator and accept whatever default strategy is employed by the application.
- Implementers can ignore the allocator, especially if the operation is not considered performance-sensitive.
- Users can co-locate POSMs for related asynchronous operations, for better locality of reference.
- For compositions that involve serial POSMs of different sizes, memory usage need only be as great as the currently extant POSM. For example, consider a composition that contains a short-lived operation that uses large POSMs (connection establishment and handshake) followed by a long-lived operation that uses small POSMs (transferring data to and from the peer).

As noted previously, all resources must be released prior to calling the completion handler. This enables memory to be recycled for subsequent asynchronous operations within an agent. This allows applications with long-lived asynchronous agents to have no hot-path memory allocations, even though the user code is unaware of associated allocators.

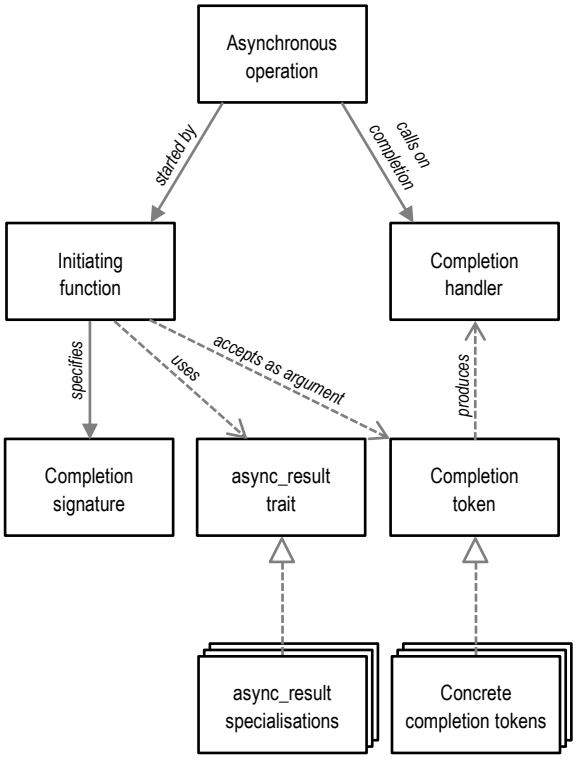
3.7 Cancellation

In Asio, many objects, such as sockets and timers, support object-wide cancellation of outstanding asynchronous operations via their close or cancel member functions. However, certain asynchronous operations also support individual, targeted cancellation. This per-operation cancellation is enabled by specifying that every asynchronous agent has an associated *cancellation slot*.

To support cancellation, an asynchronous operation installs a cancellation handler into the agent's slot. The cancellation handler is a function object that will be invoked when a cancellation signal is emitted by the user into the slot. Since a cancellation slot is associated with a single agent, the slot holds at most one handler at a time, and installing a new handler will overwrite any previously installed handler. Thus, the same slot is reused for subsequent asynchronous operations within the agent.

Cancellation is particularly useful when an asynchronous operation contains multiple child agents. For example, one child agent may be complete and the other is then cancelled, as its side effects are no longer required.

3.8 Completion tokens



A key goal of Asio's asynchronous model is to support multiple composition mechanisms. This is achieved via a *completion token*, which the user passes to a asynchronous operations' initiating function to customise the API surface of the library. By convention, the completion token is the final argument to an asynchronous operation's initiating function.

For example, if the user passes a lambda (or other function object) as the completion token, the asynchronous operation behaves as previously described: the operation begins, and when the operation completes the result is passed to the lambda.

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

When the user passes the `use_future` completion token, the operation behaves as though implemented in terms of a `promise` and `future` pair. The initiating function does not just launch the operation, but also returns a `future` that may be used to await the result.

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

Similarly, when the user passes the `use_awaitable` completion token, the initiating function behaves as though it is an `awaitable`-based coroutine[6]. However, in this case the initiating function does not launch the asynchronous operation directly. It only returns the `awaitable`, which in turn launches the operation when it is `co_await`-ed.

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

  // ...
}
```

Finally, the `fibers::yield` completion token causes the initiating function to behave as a fiber-aware synchronous operation. It not only begins the asynchronous operation, but blocks the fiber until it is complete. From the point of the fiber, it *is* a synchronous operation.

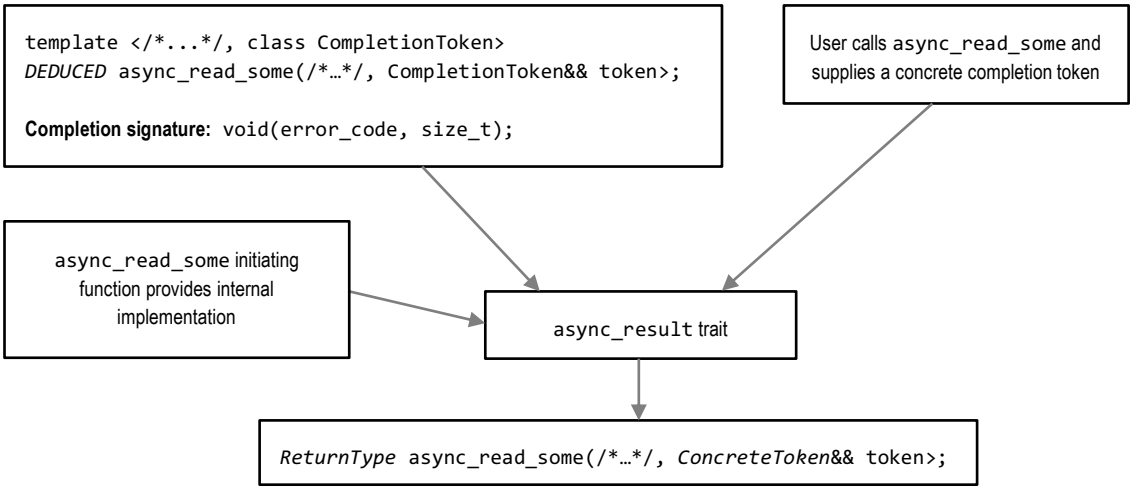
```
void foo()
{
  size_t n = socket.async_read_some(
      buffer, fibers::yield
    );

  // ...
}
```

All of these uses are supported by a single implementation of the `async_read_some` initiating function.

To achieve this, an asynchronous operation must first specify a *completion signature* (or, possibly, signatures) that describes the arguments that will be passed to its completion handler.

Then, the operation's initiating function takes the completion signature, completion token, and its internal implementation and passes them to the *async_result* trait. The `async_result` trait is a customisation point that combines these to first produce a concrete completion handler, and then launch the operation.



[6] The `awaitable` class template is included with the Asio library as a return type for C++20 coroutines. These coroutines can be trivially implemented in terms of other `awaitable`-based coroutines.

To see this in practice, let's use a detached thread to adapt a synchronous operation into an asynchronous one:[7]

```
template <class CompletionToken>
auto async_read_some(tcp::socket& s, const mutable_buffer& b, CompletionToken&& token)
{
  auto init = [](
      auto completion_handler,
      tcp::socket* s,
      const mutable_buffer& b)
  {
    std::thread(
        [](
            auto completion_handler,
            tcp::socket* s,
            const mutable_buffer& b
          )
        {
          error_code ec;
          size_t n = s->read_some(b, ec);
          std::move(completion_handler)(ec, n);
        },
        std::move(completion_handler),
        s,
        b
      ).detach();
  };

  return async_result<
      decay_t<CompletionToken>,
      void(error_code, size_t)
    >::initiate(
        init,
        std::forward<CompletionToken>(token),
        &s,
        b
      );
}
```

Define a function object that contains the code to launch the asynchronous operation. This is passed the concrete completion handler, followed by any additional arguments that were passed through the async_result trait.

The body of the function object spawns a new thread to perform the operation.

Once the operation completes, the completion handler is passed the result.

The async_result trait is passed the (decayed) completion token type, and the completion signatures of the asynchronous operation.

Call the trait's initiate member function, first passing the function object that launches the operation.

Next comes the forwarded completion token. The trait implementation will convert this into a concrete completion handler.

Finally, pass any additional arguments for the function object. Assume that these may be decay-copied and moved by the trait implementation.

We can think of the completion token as a kind of proto completion handler. In the case where we pass a function object (like a lambda) as the completion token, it already satisfies the completion handler requirements. The `async_result` primary template handles this case by trivially forwarding the arguments through:

```
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)...);
  }
};
```

[7] For illustrative purposes only. Not recommended for real world use!

We can see here that this default implementation avoids copies of all arguments, thus ensuring that eager initiation is as efficient as possible.

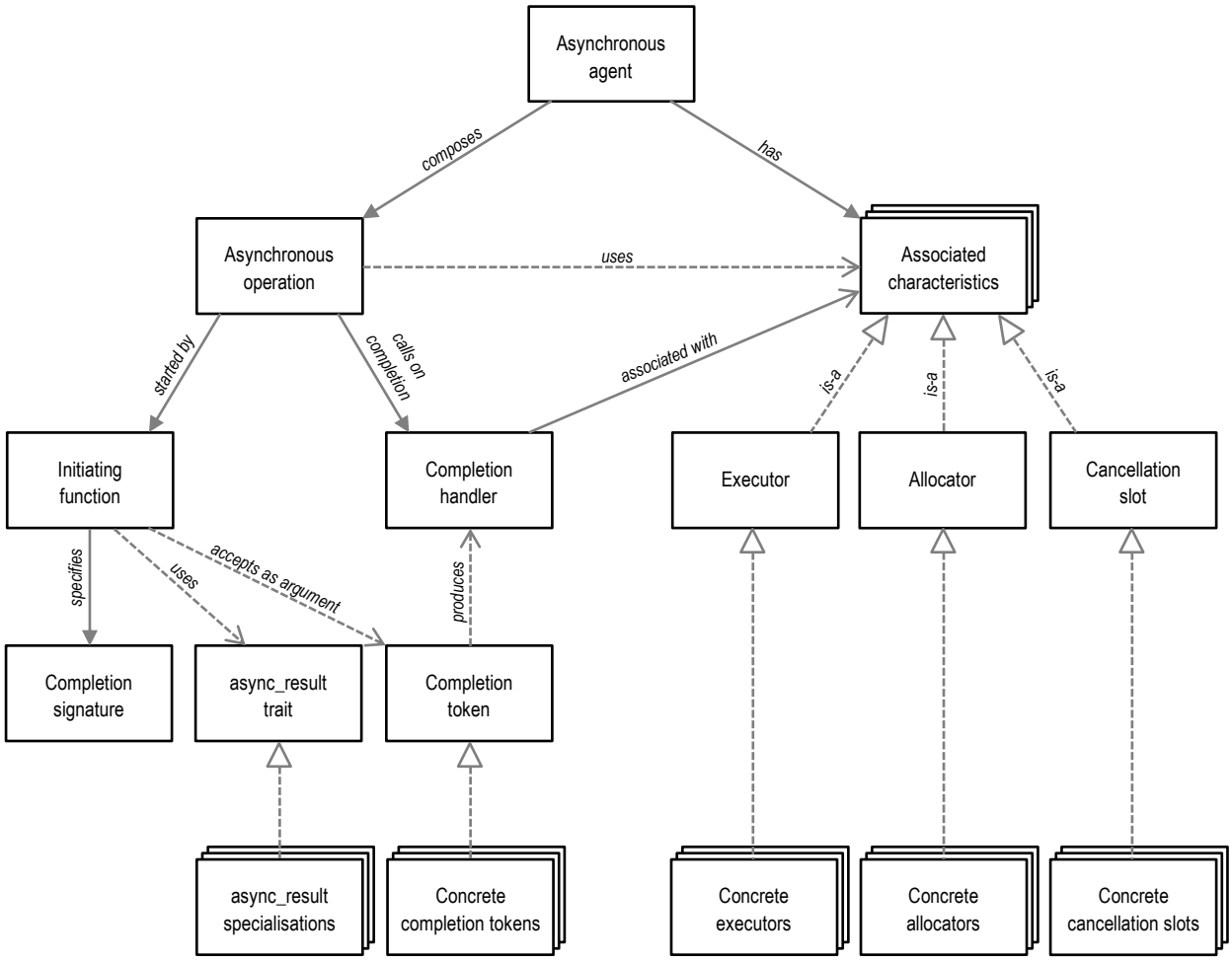
On the other hand, a lazy completion token (such as `use_awaitable` above) may capture these arguments for deferred launching of the operation. For example, a specialisation for a trivial `deferred` token (that simply packages the operation for later) might look something like this:

```
template <completion_signature... Signatures>
struct async_result<deferred_t, Signatures...>
{
  template <class Initiation, class... Args>
  static auto initiate(Initiation initiation, deferred_t, Args... args)
  {
    return [
        initiation = std::move(initiation),
        arg_pack = std::make_tuple(std::move(args)...)
      ](auto&& token) mutable
    {
      return std::apply(
          [&](auto&&... args)
          {
            return async_result<decay_t<decltype(token)>, Signatures...>::initiate(
                std::move(initiation),
                std::forward<decltype(token)>(token),
                std::forward<decltype(args)>(args)...
              );
          },
          std::move(arg_pack)
        );
    };
  }
};
```

3.9 Summary of library elements

- `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.
- `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 – defies an asynchronous agent's associated cancellation slot.
- `associated_cancellation_slot_t` template type alias
- `get_associated_cancellation_slot` function

3.10 Higher level abstractions



The asynchronous model presented in this paper provides a basis for defining higher level abstractions, but specifying these is considered beyond the scope of this paper. Instead, its scope is limited to specifying the asynchronous operations that are the building blocks of this composition.

However, the Asio library already builds on this core model to provide these additional facilities, such as:

- I/O object such as sockets and timers that expose asynchronous operations on top of this model.
- Concrete executors, such as the `io_context` executor, `thread_pool` executor, and the `strand` adapter which guarantees non-concurrent execution of completion handlers.
- Completion tokens that facilitate different composition mechanisms, such as coroutines, fibers, futures, and deferred operations.
- High level support for C++ coroutines that combines support executors and cancellation slots to allow for easy coordination of concurrent asynchronous agents.

4 Examples

Many examples that build on this core asynchronous model can be found as part of the Asio library distribution, as well as in the many applications and libraries that are build on top of Asio. However, to illustrate how high-level compositions can be built on top of this model, the following examples utilise the C++20 coroutine support that is included with Asio.

To begin, let's compare our original thread-per-connection echo server with a coroutine-based one.

Thread-per-connection echo server

```
#include <asio.hpp>

using asio::buffer;
using asio::ip::tcp;




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}});




}
```

Coroutine echo server

```
#include <asio.hpp>

using asio::awaitable;
using asio::buffer;
using asio::detached;
using asio::ip::tcp;
using asio::use_awaitable;

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();
}
```

This demonstrates how coroutine support can leverage the model to replicate both the semantic and syntactic properties of synchronous operations.

The next example is a snippet from a simple TCP socket proxy. It demonstrates how coroutines can combine with cancellation support to provide elegant coordination of concurrent asynchronous agents, and efficient timeout support:

```
constexpr auto use_nothrow_awaitable = as_tuple(use_awaitable);

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();
  }
}

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> listen(tcp::acceptor& acceptor, tcp::endpoint target)
{
  for (;;)
  {
    auto [e, client] = co_await acceptor.async_accept(use_nothrow_awaitable);
    if (e)
      break;

    auto ex = client.get_executor();
    co_spawn(ex, proxy(std::move(client), target), detached);
  }
}
```

Finally, a snippet that illustrates a connect-by-name implementation. This coroutine-based algorithm attempts connections to multiple hosts in parallel. As soon as one succeeds, the remaining operations are automatically cancelled.

```
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)
{
  assert(first != 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());
}
```

5 Sample implementation

The following implementation of the core asynchronous model is derived from the Asio source code. For clarity, it has been updated to use C++20 features, and to remove backwards feature compatibility.

```
// completion_signature concept

template <class T>
struct __is_completion_signature : false_type {};

template <class R, class... Args>
struct __is_completion_signature<R(Args...)> : true_type {};

template <class T>
concept completion_signature = __is_completion_signature<T>::value;

template <class T, class Signature>
struct __is_completion_handler_for : false_type {};

// completion_handler_for concept

template <class T, class R, class... Args>
struct __is_completion_handler_for<T, R(Args...)> : is_invocable<decay_t<T>&&, Args...> {};

template <class T, class... Signatures>
concept completion_handler_for = (completion_signature<Signatures> && ...)
  && (__is_completion_handler_for<T, Signatures>::value && ...);

// async_result trait

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)...);
  }
};

// async_initiate helper function

template <
    class CompletionToken,
    completion_signature... Signatures,
    class Initiation,
    class... Args>
auto async_initiate(
    Initiation&& initiation,
    type_identity_t<CompletionToken>& token,
    Args&&... args)
  -> decltype(
      async_result<decay_t<CompletionToken>, Signatures...>::initiate(
        std::forward<Initiation>(initiation),
        std::forward<CompletionToken>(token),
        std::forward<Args>(args)...))
{
```

```
  return async_result<decay_t<CompletionToken>, Signatures...>::initiate(
      std::forward<Initiation>(initiation),
      std::forward<CompletionToken>(token),
      std::forward<Args>(args)...);
}

// completion_token_for concept

template <class... Signatures>
struct __initiation_archetype
{
  template <completion_handler_for<Signatures...> CompletionHandler>
  void operator()(CompletionHandler&&) const {}
};

template <class T, class... Signatures>
concept completion_token_for = (completion_signature<Signatures> && ...)
  && requires(T&& t)
  {
    async_initiate<T, Signatures...>(__initiation_archetype<Signatures...>{}, t);
  };

// associator trait

template <template <class, class> class Associator, class S, class C>
struct associator {};

// associated_executor trait

template <class S, class C>
struct associated_executor
{
  static auto __get(const S& s, const C& c) noexcept
  {
    if constexpr (requires { typename S::executor_type; })
      return s.get_executor();
    else if constexpr (requires { typename associator<associated_executor, S, C>::type; })
      return associator<associated_executor, S, C>::get(s, c);
    else
      return c;
  }

  using type = decltype(associated_executor::__get(declval<const S&>(), declval<const C&>()));

  static type get(const S& s, const C& c) noexcept
  {
    return __get(s, c);
  }
};

template <class S, class C>
using associated_executor_t = typename associated_executor<S, C>::type;

template <class S, class C>
associated_executor_t<S, C> get_associated_executor(const S& s, const C& c) noexcept
{
  return associated_executor<S, C>::get(s, c);
}

// associated_allocator trait

template <class S, class C>
struct associated_allocator
{
  static auto __get(const S& s, const C& c) noexcept
  {
```

```
    if constexpr (requires { typename S::allocator_type; })
      return s.get_allocator();
    else if constexpr (requires { typename associator<associated_allocator, S, C>::type; })
      return associator<associated_allocator, S, C>::get(s, c);
    else
      return c;
  }

  using type = decltype(associated_allocator::__get(declval<const S&>(), declval<const C&>()));

  static type get(const S& s, const C& c) noexcept
  {
    return __get(s, c);
  }
};

template <class S, class C>
using associated_allocator_t = typename associated_allocator<S, C>::type;

template <class S, class C>
associated_allocator_t<S, C> get_associated_allocator(const S& s, const C& c) noexcept
{
  return associated_allocator<S, C>::get(s, c);
}

// associated_cancellation_slot trait

template <class S, class C>
struct associated_cancellation_slot
{
  static auto __get(const S& s, const C& c) noexcept
  {
    if constexpr (requires { typename S::cancellation_slot_type; })
      return s.get_cancellation_slot();
    else if constexpr (requires { typename associator<associated_cancellation_slot, S, C>::type; })
      return associator<associated_cancellation_slot, S, C>::get(s, c);
    else
      return c;
  }

  using type = decltype(associated_cancellation_slot::__get(declval<const S&>(), declval<const C&>()));

  static type get(const S& s, const C& c) noexcept
  {
    return __get(s, c);
  }
};

template <class S, class C>
using associated_cancellation_slot_t = typename associated_cancellation_slot<S, C>::type;

template <class S, class C>
associated_cancellation_slot_t<S, C> get_associated_cancellation_slot(const S& s, const C& c) noexcept
{
  return associated_cancellation_slot<S, C>::get(s, c);
}
```

6 Acknowledgements

The author would like to thank Klemens Morgenstern, Richard Hodges, and Jamie Allsop for providing feedback on this paper. The author would also like to acknowledge Richard Hodges as the author of one of the examples included above.