Sender/Receiver Interface For Networking

Document #:	P2762R0
Date:	2023-01-13
Project:	Programming Language C++
Audience:	Networking Study Group (SG4) Library Evolution Working Group

Reply-to:	Dietmar Kühl (Bloomberg) <dkuhl@bloomberg.net>

Contents

This document proposes the addition of senders for asynchronous networking operations to the Networking TS and, ultimately, to the C++ Standard. As the `std::execution` proposal isn't landed, yet, this proposal is kept at a high level and primarily intended to discuss what a potential interface for asynchronous networking operations could look like.

1 Motivation

The `std::execution` proposal (P2300) proposes sender/receiver as a general framework for structured and composable concurrency. The currently proposed components define a framework primary targeted at concurrent execution within a program. If this framework gets adopted, it should be possible to integrate other asynchronous

work like networking. To facilitate such integration, it is necessary to define a suitable set of senders for the relevant asynchronous network operations.

2 Related Work

The components defined by P2300 provide a complete framework for managing asynchronous operations and no other facilities beyond senders for the managing the networking operations are needed. The Networking TS defines its own framework for asynchronous operations. This paper does *not* propose the removal of the other framework; whether the asynchronous framework from the Networking TS should be retained or removed is a separate discussion.

There is a proposal for Standard Secure Networking (P2586). The current proposal consists of a high level description and a few possibly usage examples. Based on the usage example, this proposal does *not* include any binding to an asynchronous system. At most, it gets to the question on whether a coroutine interface should be provided to its "poll" facility. So far I haven't created a binding of the interfaces in this proposal to the facilities proposed by P2586 but I don't think there would be any problem doing so. From the current document, it isn't clear to me whether an active "poll" can be interrupted to add new work. Whether it would be a reasonable implementation choice to use P2586 as the base implementation isn't quite clear, as it seems beneficial to potentially use a completion interface, e.g., io_uring, directly.

The focus of P2586 is secured networking and I haven't managed to experiment with a secured version of the proposed networking senders, yet.

P2586 makes some claims about allocations needed for a design based on P2300; there is actually no need to do any allocations at all! The current experimental implementation (it is part of my experimental standard library) doesn't use any allocations in the networking senders (unless variable sized scatter/gather buffers are specified in a way incompatible with an array of `iovec`). When using `poll()` to wait for activity, currently a `std::vector<::pollfd>` and a `std::vector` of completions are used. However, it would be easily possible to specify an interface to a suitable I/O context providing control over the maximum size of these arrays and their required memory to avoid any allocations

3 Design Choices

The basic interface of the senders for the asynchronous network operations is informed by the Networking TS: the available operations and their arguments will be similar. Even so there are some design choices. In most cases, the alternatives aren't exclusive and multiple variations can be supported to support different uses.

The sample code for the different considerations concentrates on the respective choice being considered. As other design choices may affect the resulting code, one of the corresponding options is picked. The different design choices are mostly orthogonal, although some of the choices (notably whether the operations should be member functions) may limit the possibilities for other considerations.

3.1 Obtaining the Scheduler

The networking operations need a suitable context dealing with the asynchronousity, i.e., something using `poll(2)`, `epoll(2)`, `kqueue(2)`, `io_uring(2)`, completion ports, etc., to schedule the operation. The context is abstracted by a scheduler capable of scheduling the respective networking operations. There are a few options for how the scheduler can be obtained:

1. The operation is used as a sender factory and the scheduler is passed in as an argument. This approach makes the scheduling explict when creating the asynchronous operation, e.g.:

```
auto make_accept(auto scheduler, auto& socket) {
    return async::accept(scheduler, socket);
}
```

2. The operation is used as a sender adatper and the scheduler is obtained using `get_completion_scheduler<set_value_t>(s)` from the upstream sender, e.g.:

```
auto make_accept(auto scheduler, auto& socket) {
    return schedule(scheduler)
         | async::accept(socket)
         ;
}
```

3. The operation is used as a sender factory and the sheduler is obtained using `get_scheduler(get_env(r))` from the downstream sender. This approach allows imbuing a work graph with a scheduler from the usage end, e.g.:

```
auto make_accept(auto scheduler, auto& socket) {
    return on(scheduler, async::accept(socket));
}
```

The most useful of these options seems to be the third one, i.e., injecting the used scheduler from the point where the asynchronous work is actually used. The other two options require knowledge of the scheduler while building up the asynchronous work.

As a potential variation of the second and third option, a specific customization point name, e.g., `get_completion_io_scheduler` or `get_io_scheduler`, could be used. Using different names could enable the separation of the I/O scheduler from schedulers dedicated to doing work. These queries could fall back to the respective non-specific queries when not provided.

For the examples below, the third option is assumed. However, all three options are viable condidates.

3.2 Error Reporting

For the Networking TS, errors of asynchronous operations are reported using an `std::error_code` argument as part of the completion signature. As there is exactly one completion function used, there isn't really a different alternative. Using receivers supports multiple completion functions, thereby allowing multiple choices:

1. The operation could complete using one `set_value` call using the same fused completion consisting of an `std::error_code` and the other completion arguments as the Networking TS does, e.g.:

```
auto sender
    = async::read(socket, buffer)
    | then([](error_code const& ec, int n) { ... }
    ;
```

   Within a coroutine, structured binding could be used to decompose the result, e.g.:

```
auto[ec, n] = co_await async::read(socket, buffer);
```

2. As the error path is different in the completing functions, it can be reasonable to call different `set_value` functions: one with an `std::error_code` argument and another one (or even multiple ones) with the arguments for the success case. This approch wouldn't work with coroutines as these are restricted to using just one completion signature. Also, a downstream sender would need an overloaded `set_value` to deal with the result:

```
auto sender
    = async::read(socket, buffer)
    | overload(
        [](int n){ /* success path */ },
        [](error_code const& ec){ /* error path */ }
      )
    ;
```

3. Similar to the previous alternative but instead of reporting errors using the `set_value` channel using the `set_error` channel, e.g.:

```
auto sender = async::read(socket, buffer)
    | then([](int n) { /* success path */ }
    | upon_error([](error_code const& ec) { /* error path */ }
    ;
```

   While this approach works with coroutines, it would end up using exceptions, e.g.:

```
try {
    int n = co_await async::read(socket, buffer);
    // success path
} catch (error_code const& ec) {
    // error path
}
```

   However, when using coroutines, it would be possible to use a generic algorithm fusing the `set_value` and the `set_error` results back into one `set_value` result to avoid an exception.

4. There may even be space for a combination of reporting some errors using `set_value` while reporting others using `set_error`, depending on the severity of the error.

5. With senders getting composed in a structured form, it may be reasonable to offer passing a reference to an `std::error_code` and populating that when present and otherwise reporting the error on the `set_error` channel. That would be similar to the synchronous networking operations of the Networking TS reporting errors through the passed argument or an exception. This approach would work reasonably well using coroutines:

```
error_code ec;
int n = co_await async::read(socket, buffer, ec);
if (!ec)
    /* success path */;
else
    /* error path */
```

The most basic variations seems to use a combination of `set_value` for the successful case and `set_error` for the failure cases; the other combinations can be build from that. Also, recognizing an error can be used by algorithms to decide continuing differently upon error, e.g., cancelling other operation for a `when_all`.

However, some of the error cases may have been partial successes. In that case, using the `set_error` channel taking just one argument is somewhat limiting. On the other hand, when substantial work is done and partial successes become reasonable, it is likely that intermediate results are to be produced and algorithms of a different shape are used anyway.

When using asynchronous operations within a coroutine, there is only one `set_value` supported which can, however, return multiple values using a `std::tuple`, that is then likely decomposed using structured binding. That is when using coroutine defining different `set_value` channels isn't an option. For a coroutine, the `set_error` channel would be turned into an exception. With variations of the asynchronous operations taking an optional `std::error_code` reference as an argument, the coroutine experience would be similar to the synchronous code. Likewise, a coroutine-friendly version of the operations can be provided.

It is possible to offer a combination of the different options. The design choice would name the operations (or the namespace they live in) appropriately. The examples here assume using `set_value` and `set_error` for success and error handling.

3.3 Member vs. Non-Member Operation

The Networking TS uses both member and non-member functions for its operations. Member functions are what users are used to from other languages, where there often aren't different option. The problem is compounded by many IDEs providing simple name completions for member functions. For example:

```
auto sender = socket.async_read(buffer)
    | then([](auto&&...){ /* use result */ }
    ;
```

Similarly, when using coroutines:

```
auto[ec, n] = co_await socket.async_read(buffer);
```

On the hand side, CPOs can't be member functions (well, CPOs are classes with function call *member* functions but they don't really *look* like member functions and they aren't a member of some entity operated on). Also, adding members to classes tends to lead to "kitchen sink" classes acquiring ever more operations over time (see, e.g., `std::basic_string`). The potentially fairly large number of variations (see other design choices) is probably easier managed using non-member function. For example, there may be groups of operations in different namespaces based on their intended use, e.g., `async` for the senders directly used to chain operations and `coro` for senders used within coroutines.

```
auto sender = async::read(socket, buffer)
    | then([](auto&&...){ /* use result */ }
    ;
```

Similarly, when using coroutines:

```
auto[ec, n] = co_await coro::read(socket, buffer);
```

As there is generally no entity used with the P2300 algorithms, these aren't member of classes. For the network operations there is the socket providing an entity and the operations could be defined as member functions of these. Using non-member names providing the full variation of options doesn't exclude using member functions for the expected likely use cases, probably just delegating to the respective non-member operations. The examples here don't use member functions.

3.4 I/O Scheduler Interface

The networking (or, more general, I/O) operations will require being scheduled on a special context and require being run on a corresponding scheduler. Also, it is likely desirable to support different schedulers, e.g., one using the most efficient real implementation, one being friendly to integration with other language's "run loop", and one allowing unit testing of networking operations. There are multiple options for how the networking operations are talking to the scheduler:

1. The networking operations and the scheduler use a secret channel. While that is probably the easiest to specify, it means that the networking operations can't accept a somehow adapted scheduler or there needs to be a protocol for how to extract the underlying scheduler.
2. Expose/abstract the various I/O operations somehow, possibly using virtual functions or, more likely, CPOs. While this approach is probably more generic, the interface to the operations is likely at a somewhat lower level than what the senders use and it is possibly platform specific. For example, a `read_some` operation used with `io_uring(2)` needs to provide a pointer to an `iovec` which needs to stay around until the operation is consumed from the completion ring buffer. That is a rather different interface than the generic buffers passed to `read_some`. It may be possible to define the scheduler interface such that it defines what the caller has to store until completion but I haven't tried implementing this approach, yet.
3. The scheduler interface may model multiple contracts (one for each support I/O operation) and each operation produces an object which gets embededded into the I/O operation's operation state object. Each supported underlying I/O interface could store its data in exactly the form needed.

4. The scheduler interface for I/O operations may be what is being proposed by the Low level file i/o library. I haven't tried to implement that.
5. The Networking TS may be doing something in that area and it may be possible to integrate with that or, at least, do something similar. I haven't tried to implement that. It seems the `io_context` uses a secret interface.

Most likely, it is preferable to have some form of I/O scheduler abstraction than using a secret interface. However, it isn't yet clear how such an interface would actually look like.

3.5 Timer Class or Just a Sender

The Networking TS defines a `basic_waitable_timer` class template. The type of this class encodes various timer properties like the underlying clock type and some wait traits. The primary need in the Networking TS for this class is the need for an entity to trigger cancellations: while operations are cancellable the cancellation needs to be explicitly wired up where necessary. Uses would look like

```
waitable_timer timer(/* timer settings */);
auto sender = wait_for(timer, 5s);
```

or using coroutines

```
co_await wait_for(timer, 5s);
```

When using sender/receiver cancellation and its necessary wiring is handled by the senders capable of cancelling operations by appropriate use of the receiver's stop token. Correspondingly, there isn't really a need for a timer class. Having to create a timer object and keeping it around is sometimes a bit annoying. Thus, it may be reasonable to allow defining timers simply by creating a suitable sender which is then scheduled on a suitable scheduler. The I/O schedulers are capabable of executing timers.

```
auto sender = wait_for(5s);

co_await wait_for(5s);
```

As with most of the other design choices it may be reasonable to support both alternatives: sometimes it may be reasonable to just schedule a timed operation without the need for an object and specifying the required properties when doing so works. In other situations it may be preferable to encapsulate the timer properties into an object and using this object to schedule multiple timed operations. In that situation it *may* be possible to define the timed operations in a way which doesn't require the timer entity to stay alive until the timed operation completes: removing the need for the timer entity to remain valid until the timed operation completes should make their use simpler. The actual timed operation would be maintained by the scheduler.

3.6 Higher Level Tools

The basic networking or I/O operations are fairly straightforward and there are actually not that many of them (the `io_uring` operations are probably a good indication of the overall scope including operations beyond networking). Concentrating on networking operations the Networking TS doesn't provide everything `io_uring` does. Beyond the basic operations provided by the underlying system, the networking operations can reasonably be composed into higher level algorithms. For example, an `async::read_some` operation potentially reading partial buffers successfully can be composed into an `async::read` operations always reading a complete buffer or failing. The question is, what algorithms should be included in the proposal, if any?

Algorithms like `async::read` and `async::write` are somewhat obvious examples. Something like an `async::resolve` could be an example of a rather non-trivial algorithm: there is the synchronous `getaddrinfo(3)` function, but there doesn't seem to be an asynchronous alternative. With an asynchronous framework in place, it seems reasonable to include an asynchronous version of `getaddrinfo(3)`.

Beyond sender algorithms, there may also be some other interesting components:

— It may be useful to have a coroutine task (`io_task`) injecting a scheduler into asynchronous networking operations used within a coroutine together with a suitable scope (`io_scope`) similar to `async_scope` but also tied to some I/O scheduler. The corresponding task class probably needs to be templatized on the relevant scheduler type.
— For full-duplex operation of a socket, i.e., scheduling concurrent reading and writing operation, something like a ring buffer with sender interfaces to the production and consumption of buffers seems useful.

There are probably various other useful algorithms and components.

4 Cancellation Concern

The networking operations are generally inactive after the operation was started but the network operation hasn't completed yet. To cancel such an operation, it is necessary to actively trigger some cancellation function, i.e., a simple test of an atomic `bool` provided by a stop token generally won't work. Thus, the various operations need to register a callback with the receiver's stop token. In cases where the stop token isn't `no_stop_token`, this registration needs to do some synchronization both for the registration and the deregistration of the callback. Repeatedly doing that operation while processing data on a socket may be a performance concern.

It may be possible to avoid setting up cancellation for individual operations and rather hook the cancellation once to a suitable entity like a socket. The corresponding approaches will need some level of support by the library. For example, it may be necessary to support calls to cancel some operations on a socket.

Even when doing so, it may be necessary to inhibit registration of callbacks with a stop token. For example, `when_all` will use receivers using a different stop token than `no_stop_token` with its various senders. In that case it may be useful to have a sender adapter which passes through all completion signals received but always exposes a `no_stop_token` to its sender.

Some systems already have some cancellation support for common use cases. For exmaple, `io_uring(2)` supports timeouts for its operations (using `IORING_OP_LINK_TIMEOUT`). To easily tap into this pattern, it may be reasonable to have a corresponding `timeout` sender taking a time and sender which may be a networking operation: if either the time expires or the sender completes, the respective other operation would be cancelled. Where available, the `timeout` could then just setup the underlying system to provide the corresponding functionality. Otherwise, it would behave like a `timer` and another sender given to a `when_any` operation completing appropriately upon completion of the first operation and cancelling the other via a stop token.

This area still needs some experimentation and, I think, design. The general direction of encapsulating the cancellation into the asynchronous operations is rather interesting, but it isn't a priori clear how to avoid potential costs.

5 Wording for Networking CPOs

In 14.2 [io_context.io_context], add a `scheduler_type` and a `get_scheduler()` method to the synopsis:

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
    class io_context : public execution_context
    {
    public:
        ...
        class executor_type;
        [class scheduler_type;]{add}
        ...
        executor_type  get_executor() noexcept;
        [scheduler_type get_scheduler() noexcept;]{add}
```

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

In 14.2 [io_context.io_context] after paragraph 2, add a new paragraph:

2 `count_type` is an implementation-defined unsigned integral type of at least 32 bits.

3 `scheduler_type` is a type modelling *scheduler* [exec.sched].

In 14.2.1a [io_context.io_context.members] after paragraph 3 add a new paragraph:

`executor_type get_executor() noexcept;`

3 *Returns:* An executor that may be used for submitting function objects to the `io_context`.

`scheduler_type get_scheduler() noexcept;`

4 *Returns:* A scheduler that may be used for scheduling sender objects on the `io_context`.

Add a new section for the networking operations:

6 Networking Senders [net.sender]

6.1 General [net.sender.general]

1 Subclause [net.sender] defines sender factories for newtorking operations. When the corresponding operations are started, they do no block any thread. Instead they complete once the corresponding operation becomes ready. How the system determines that an operation is ready is implementation specific.

2 These sender factories share some common behavior:

(2.1) — When a sender `s` is `connect`ed [exec.connect] to a receiver `r` and the resulting operation state is `execution::start`ed [exec.op_state.start], a callback for cancellation is registered with the stop token obtained using `get_stop_token(get_env(r))`. When this callback is invoked the corresponding operation is cancelled and one of the completion signatures is invoked in a timely manner. The operation can complete using `set_stopped` [exec.set_stopped] but it may still complete with one of the other completion signals instead if the operation became ready otherwise.

(2.2) — Starting an operation state [exec.op.state.start] may complete the operation immediately from the starting thread if it is ready to be completed. Otherwise, the operation is initiated using the scheduler and gets completed once it becomes ready.

(2.3) — Any object referenced in the sender call needs to stay valid while the sender or an operation state obtained from the sender by connecting it to a receiver is used. A sender for a network operation stops being used when it gets connected to a a receiver or when it gets destroyed. An operation state stops being used when it is destroyed or when a completion signal is invoked after the operation state was `execution::start`ed [exec.op_state.start].

6.2 Network Operations

The shape of the exact operations isn't quite clear yet, as there are various design option (see above). Below the relevant operations are listed together with their arguments and their likely `completion_signatures`. The current list of operations may be incomplete. The "as if" code isn't necessarily ready to compile and may omit necessary casts and use private operations in some cases.

6.2.1 `net::async_accept` [net.sender.async.accept]

1 `async_accept` is a customization point object for accepting new connections.

2 `async_accept(acceptor)` takes a reference to an acceptor socket `acceptor` as parameter and returns a sender `s`. Let `Acceptor` be the type `remove_cvref_t<decltype(acceptor)>` and `env` be an environment object.

3 The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(3.1) — `execution::set_value_t(typename Acceptor::socket_type, typename Acceptor::endpoint_type)`
(3.2) — `execution::set_error_t(error_code)`
(3.3) — `execution::set_stopped_t()`

4 When `s` is connected to a receiver `r` and the resulting operation state is started, it initiates an asynchronous operation to extract a socket from the queue of pending connections for `acceptor`, as if by POSIX:

```
typename Acceptor::endpoint_type ep;
socklen_t        addrlen(ep.capacity());
auto h = accept(acceptor.native_handle(), ep.data(), &addrlen);
if (h < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    ep.resize(addrlen);
    execution::set_value(std::move(r), typename Acceptor::socket_type(h), ep);
}
```

5 The Networking TS passes the `endpoint` optionally as a reference argument, resulting in two interfaces: one where the endpoint isn't obtained and one where it is. The interface for `async_accept` could reference the `endpoint` as well, instead of providing it with `set_value`. An alternative design allowing omission of the `endpoint` is to have `async_accept` not providing an `endpoint` with `set_value` and `async_accept_from` with the completion signature above.

6.2.2 `net::async_connect` [net.sender.async.connect]

1 `async_connect` is a customization point object for connecting to a server socket.

2 `async_connect(socket, endpoint)` takes a reference `socket` and an `ep` as parameters and returns a sender `s`. Let `Socket` be the type `remove_cvref_t<decltype(socket)>` and `env` be an environment object.

3 The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(3.1) — `execution::set_value_t()`
(3.2) — `execution::set_error_t(error_code)`
(3.3) — `execution::set_stopped_t()`

4 When `s` is connected to a receiver `r` and the resulting operation state is started, it initiates an asynchronous operation to connect `socket` to a server socket, as if by POSIX:

```
if (connect(acceptor.native_handle(), ep.data(), ep.size()) < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    execution::set_value(std::move(r));
}
```

6.2.3 `net::async_read_some` [net.sender.async.read.some]

1 `async_read_some` is a customization point object for reading data from a socket into a buffer sequence.

[2] `async_read_some(socket, buffers)` takes a reference `socket` and a buffer sequence `buffers` as parameters and returns a sender `s`. Let `env` be an environment object.

[3] The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(3.1) — `execution::set_value_t(size_t)`
(3.2) — `execution::set_error_t(error_code)`
(3.3) — `execution::set_stopped_t()`

[4] The default behavior is equivalent to `net::async_receive(socket, buffers)`. [*Note:* Custom implementations may want to use non-socket operations for the implementation, e.g., to support the same interface to files. –*End Note*]

6.2.4 `net::async_receive` [net.sender.async.receive]

[1] `async_receive` is a customization point object for reading data from a socket into a buffer sequence.

[2] `async_receive(socket, buffers)` takes a reference `socket` and a buffer sequence `buffers` as parameters and returns `async_receive(socket, message_flags{}, buffers)`.

[3] `async_receive(socket, flags, buffers)` takes a reference `socket`, `flags` of type `message_flags`, and a buffer sequence `buffers` as parameters and returns a sender `s`. Let `env` be an environment object.

[4] The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(4.1) — `execution::set_value_t(size_t)`
(4.2) — `execution::set_error_t(error_code)`
(4.3) — `execution::set_stopped_t()`

[5] When `s` is connected to a receiver `r` and the resulting operation state is started, it initiates an asynchronous operation to read data from `socket` into the buffer sequence `buffers`. The operation constructs and array `iov` of POSIX `struct iovec` of size `length` corresponding to `buffers` and reads data as if by POSIX:

```
msghdr message;
message.msg_name        = nullptr;
message.msg_namelen     = 0;
message.msg_iov         = iov;
message.msg_iovlen      = length;
message.msg_control     = nullptr;
message.msg_controllen  = 0;
message.msg_flags       = 0;
auto n = recvmsg(socket.native_handle(), &message, static_cast<int>(flags);

if (n < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    execution::set_value(std::move(r), n);
}
```

6.2.5 `net::async_receive_from` [net.sender.async.receive.from]

[1] `async_receive_from` is a customization point object for reading data from a socket into a buffer sequence and also getting the source of the data.

[2] `async_receive_from(socket, buffers)` takes a reference `socket` and a buffer sequence `buffers` as parameters and returns `async_receive_from(socket, message_flags{}, buffers)`.

[3] `async_receive_from(socket, flags, buffers)` takes a reference `socket`, `flags` of type `message_flags`, and a buffer sequence `buffers` as parameters and returns a sender `s`. Let `Socket` be the type `remove_cvref_t<decltype(socket)>` and `env` be an environment object.

[4] The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(4.1) — `execution::set_value_t(size_t, typename Socket::endpoint_type)`
(4.2) — `execution::set_error_t(error_code)`
(4.3) — `execution::set_stopped_t()`

[5] When `s` is connected to a receiver `r` and the resulting operation state is started, it initiates an asynchronous operation to read data from `socket` into the buffer sequence `buffers`. The operation constructs and array `iov` of POSIX `struct iovec` of size `length` corresponding to `buffers` and reads data as if by POSIX:

```
typename Socket::endpoint_type addr;
socklen_t                      addrlen{ep.capacity()};
msghdr message;
message.msg_name        = addr.data();
message.msg_namelen     = &addrlen;
message.msg_iov         = iov;
message.msg_iovlen      = length;
message.msg_control     = nullptr;
message.msg_controllen  = 0;
message.msg_flags       = 0;
auto n = recvmsg(socket.native_handle(), &message, static_cast<int>(flags);

if (n < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    addr.resize(addrlen);
    execution::set_value(std::move(r), n, addr);
}
```

6.2.6 `net::async_send` [net.sender.async.send]

[1] `async_send` is a customization point object for writing data from a buffer sequence into a socket.

[2] `async_send(socket, buffers)` takes a reference `socket` and a buffer sequence `buffers` as parameters and returns `async_send(socket, message_flags{}, buffers)`.

[3] `async_send(socket, flags, buffers)` takes a reference `socket`, `flags` of type `message_flags`, and a buffer sequence `buffers` as parameters and returns a sender `s`. Let `env` be an environment object.

[4] The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(4.1) — `execution::set_value_t(size_t)`
(4.2) — `execution::set_error_t(error_code)`
(4.3) — `execution::set_stopped_t()`

[5] When `s` is connected to a receiver `r` and the resulting operation state is started, it initiates an asynchronous operation to write data from the buffer sequence `buffers` to `socket`. The operation constructs and array `iov` of POSIX `struct iovec` of size `length` corresponding to `buffers` and writes data as if by POSIX:

```
msghdr message;
message.msg_name        = nullptr;
message.msg_namelen     = 0;
```

```
message.msg_iov         = iov;
message.msg_iovlen      = length;
message.msg_control     = nullptr;
message.msg_controllen  = 0;
message.msg_flags       = 0;
auto n = sendmsg(socket.native_handle(), &message, static_cast<int>(flags);

if (n < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    execution::set_value(std::move(r), n);
}
```

6.2.7 `net::async_send_to` [net.sender.async.send.to]

1 `async_send_to` is a customization point object for writing data from a buffer sequence into a socket to a specific address.

2 `async_send_to(socket, buffers, ep)` takes a reference `socket`, a buffer sequence `buffers`, and an endpoint `ep` as parameters and returns `async_send_to(socket, message_flags{}, buffers, ep)`.

3 `async_send_to(socket, flags, buffers, ep)` takes a reference `socket`, `flags` of type `message_flags`, a buffer sequence `buffers`, and an endpoint `ep` as parameters and returns a sender `s`. Let `env` be an environment object.

4 The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(4.1) — `execution::set_value_t(size_t)`
(4.2) — `execution::set_error_t(error_code)`
(4.3) — `execution::set_stopped_t()`

5 When `s` is connected to a receiver `r` and the resulting operation state is started, it initiates an asynchronous operation to write data from the buffer sequence `buffers` to `socket` using the endpoint `ep` as the name. The operation constructs and array `iov` of POSIX `struct iovec` of size `length` corresponding to `buffers` and writes data as if by POSIX:

```
msghdr message;
message.msg_name        = ep.data();
message.msg_namelen     = ep.size();
message.msg_iov         = iov;
message.msg_iovlen      = length;
message.msg_control     = nullptr;
message.msg_controllen  = 0;
message.msg_flags       = 0;
auto n = sendmsg(socket.native_handle(), &message, static_cast<int>(flags);

if (n < 0) {
    execution::set_error(std::move(r), error_code(errno, system_category()));
}
else {
    execution::set_value(std::move(r), n);
}
```

6.2.8 `net::async_write_some` [net.sender.async.write.some]

1 `async_write_some` is a customization point object for writing data from a buffer sequence into a socket.

2 `async_write_some(socket, buffers)` takes a reference `socket` and a buffer sequence `buffers` as parameters and returns a sender `s`. Let `env` be an environment object.

3 The completion signatures returned from `execution::get_completion_signatures(s, env)` contain three elements:

(3.1) — `execution::set_value_t(`(3.2)`size_t)` - `execution::set_error_t(error_code)`(3.3) - `execution::set_stopped_t()`

4 The default behavior is equivalent to `net::async_send(socket, buffers)`. [*Note:* Custom implementations may want to use non-socket operations for the implementation, e.g., to support the same interface to files. –*End Note*]