Executors review – Senders and Receivers

Document #: P2202R0
Date: 2020-07-15
Project: Programming Language C++
Audience: Library Evolution Working Group
Reply-to: Gašper Ažman
<gasper.azman@gmail.com>
Tony van Eerd
<tvaneerd@gmail.com>
Thomas Rodgers
<rodgert@twrodgers.com>
Tomasz Kamiński
<tomaszkam@gmail.com>
Corentin Jabot
<corentin.jabot@gmail.com>
Robert Leahy
<rleahy@rleahy.ca>
Gordon Brown (author)
<gordon@codeplay.com>
Kirk Shoop (author)
<kirkshoop@fb.com>
Eric Niebler
<eniebler@fb.com>
Dietmar Kühl
<dkuhl@bloomberg.com>

Contents

1 Abstract

This is the report of the Executors review group 2: Senders/Receivers, of paper [P0443R13].

We found a few issues with the paper. Some have to do with a general lack of examples and the fact that the paper isn't finished. Due to the seeming generality of the mechanism and lack of example guidance and discussion, it is difficult to program against it without running into subtle impedance mismatches in interfaces. Clearer guidance on "how you should use this" as opposed to just concept definitions would be very welcome.

As expected, the paper authors were extremely responsive during the review, and have already taken a number of issues under advisement.

We are looking forward to the next iteration of the paper.

2 Findings

2.1 Receivers that match any scheduler are impossible to write

The `scheduler` concept is underconstrained, because it does not define a sensible constraint for the produced sender. Therefore, one cannot write a receiver that would match any scheduler.

Specifically, `scheduler` does not define what kind of sender `execution::schedule` returns, as what will be propagated to `set_value` of the receiver is undefined. From our understanding, the produced sender should be a `void`-sender (call `set_value()`).

Is this an omission in requirements, or are senders produced from `schedule` all allowed to produce values? If the latter is true, how should algorithms like `via`, `on` handle these values?

This is https://github.com/executors/executors/issues/467 and https://github.com/atomgalaxy/review-executor-sendrecv/issues/9 .

To illustrate, it is unclear what types may be used as `Args...` and `E` for the below receiver, for `execution::submit(execution::schedule(sch), my_receiver)` to be valid for any `scheduler sch`:

```
struct my_receiver
{
   void set_value(Args...);
   void set_error(E);
   void set_done();
};
```

2.2 More examples needed in order to explain misunderstandings

We ran into trouble understanding the current intention for usage of sender/receiver when trying to implement a telnet client. Kirk Shoop was kind enough to elaborate, but we never finished it regardless (although that was partly due to being dragged into other projects).

The standard might not be a teaching document, but the papers should be. Kirk graciously supplied the following example of a telnetish-client:

```
char const    hello[] = { 'h', 'e', 'l', 'l', 'o', '\n' };
char          buffer[1024];
io_context    context;

sync_wait(
   just(endpoint)                                // should really be resolve
```

```
    | async_connect(stream_socket(context))    // uses the result of the previous step
    | async_send(hello)                        // uses the result of the previous step
    | async_receive(buffer)                    // uses the result of the previous step
    | tap([](auto&&){ std::cout << "received response\n";})
    | async_close(),                           // uses the result of the previous step
  context);
```

This snippet, obviously, just does something akin to a "ping". It does illustrate how an asynchronous client could be implemented. The experimental implementation of the Networking TS using sender/receiver got to the point to make the above code operational. An example of a telnet-like client implementation, for instance, would be welcome.

The experimental implementation was done natively instead of using the facilities from the Networking TS. In retrospect, it seems possible to use the interfaces from the Networking TS to implement the asynchronous operations in terms of sender/receiver, though: the sender would still be a custom class producing an operation state object. However, the actual asynchronous operation could use the completion token abstraction to plug into the sender/receiver protocol using the operation state. Such an implementation of the asynchronous operations isn't done, yet.

It's also noted that the paper currently comes with quite a few TODOs, mainly about streams, which should probably be worked out before we ship the design. Without a streams concept, it is unclear how to use sender/receivers to process repeated operations like repeatedly reading packets from a socket or repeatedly accepting a client connection when listening on a port.

Main discussion: https://github.com/atomgalaxy/review-executor-sendrecv/issues/11

2.3 The `typed_sender` :: has-error-types definition has wrong template parameter kind

The `typed_sender` definition uses the following `S::error_types` detection template:

```
template<template<class...> class Variant>
struct has-error-types; // exposition only
```

However, the `S::error_types` is a template that accepts a single `Variant` parameter, so it should be:

```
template<template<template<class...> class Variant> class>
struct has-error-types; // exposition only
```

This is issue https://github.com/atomgalaxy/review-executor-sendrecv/issues/10.

2.4 `as_receiver::set_value` should preserve the value category of the invocable

Given that:

- `set_value` (when it succeeds) is "terminal" and
- `as-receiver::set_error` and `::set_done` don't make use of `as-receiver::f_`

`as-receiver::set_value` could be changed from:

```
void set_value() noexcept(is_nothrow_invocable_v<F&>) {
  invoke(f_);
}
```

to

```
void set_value() noexcept(is_nothrow_invocable_v<F&&>) {
  invoke(move(f_));
}
```

The executors team noted that this was because of the way in which `executor_of` is specified (it only requires that the provided function object type be lvalue invocable). This could be an argument for changing both `as-receiver::set_value` and `executor_of`.

If we examine `executor_of` with respect to `inline_executor` (which is in P0443 as an example) we find that:

- `inline_executor::execute` is sensible in that it propagates the value category of the function object when invoking it
- For some type `F` which is lvalue invocable but not rvalue invocable (and which is also default constructible for ease of exposition):
  - `executor_of<inline_executor, F>` is satisfied but
  - `inline_executor{}.execute(F{})` is ill-formed

This is issue https://github.com/atomgalaxy/review-executor-sendrecv/issues/8.

2.5 Clarify when senders are reusable

In §1.6.2 a hypothetical algorithm `retry` is described which would make repeated use of a sender:

> The `set_error` member, on the other hand, reconstructs the operation state in-place by making another call to `connect` with the original sender and a new instance of the custom receiver.

The paper doesn't seem to discuss the possibility of sender reuse in describing the senders which the following open questions:

- Is reuse of senders intended?
- If yes should this be conditional or unconditional?
- If conditional under which conditions?

Note that the example of `_then_sender` in §1.6.1 is suggests that senders are either:

- Not intended to be reused
- Only intended to be conditionally reused

Given that `_then_sender::connect` is rvalue qualified (suggesting that once the function is called `*this` must not be reused).

Based on a reference implementation (libunifex) the executors team suggested that `then` is meant to be transparent to the value category of the sender simply forwarding it through. While this may disqualify `_then_sender` as a counterexample to sender reusability it leaves open the question of intention and direction (i.e. the three questions enumerated above).

This is issue https://github.com/atomgalaxy/review-executor-sendrecv/issues/7.

2.6 `as-invocable::operator()` ref qualification

The motivation for the function call operator of `as-invocable` being mutable lvalue ref qualified is unclear.

Since `as-invocable::operator()` will either call `set_value` or `set_done` on the underlying receiver then that receiver's lifecycle is complete as per the "receiver contract" (i.e. one may no longer call receiver functions on it). Accordingly invoking `as-invocable::operator()` a second time would be invalid.

While this suggests lvalueness, this semantic is instead evocative of rvalueness. This is bolstered by the fact that the implementation of `as-invocable` actually does move from its indirect receiver in `operator()`.

More confusingly:

— §2.2.3.4 doesn't seem to prohibit `std::execution::execute` from treating submitted function objects as rvalues when invoking them
— The `executor_of` concept suggests function objects should be invoked as lvalues
— The `inline_executor` example doesn't work with the current formulation of `as-invocable`

This is part of issue https://github.com/atomgalaxy/review-executor-sendrecv/issues/5.

2.7 `as-invocable` moves-from pointee receiver

`as-invocable::operator()` moves from the receiver its member `r_` points to when calling `set_value` however that may throw an exception. In that case it then proceeds to move from `*r_` again when calling `set_error`. This seems to run afoul of best practices regarding moving from objects.

The executors team pointed out that this was discussed and that the alternative is to allow `set_value` to be implicitly consuming.

Another alternative would be to require that `set_value` offer an exception guarantee (i.e. if it throws it will leave the receiver in a state suitable for `set_error` to be called).

This is part of issue https://github.com/atomgalaxy/review-executor-sendrecv/issues/5.

2.8 The fallback wrapping providing implicit convertibility between types satisfying disparate concepts is confusing

Issue: https://github.com/atomgalaxy/review-executor-sendrecv/issues/4

The current definitions of customization points are conflating the concepts, by providing a fallback wrapping. For example:

1. `executor` is `scheduler`: The `exeuction::schedule` has a fallback defintion for the `executor`, that wraps int into void-sender. That means that for every `executor e` the `execution::schedule(e)` is well-formed, as consequence every `executor` is `scheduler` (both are `copy_constructible` and `equality_comparable`), but the `executor` does not subsume `scheduler`.

2. `executor` is `void-sender`, `void-sender` is `executor`: The fallbacks in `connect` automatically wraps executor in sender, but again executor does not subsume `void_sender`. Also `execution::execute` works with any `void_sender`, so `copy_constructible` and `equality_comparable` sender is executor.

This makes it difficult to understand the ergonomics of the design.

The `sender_traits<S>` have as specialization for `void_sender` that is enabled if `executor-of-impl<S,as-invocable<void-rec` which imposes `copy_constructible` and `equality_comparable` requirements, instead of `sender_to<S, void-receiver>`. This means that this specialization is enabled for `copy_constructible` senders (because they are `executors`), but not for normal senders.

The problem is not the ability to adapt the `executor` to `void-sender`, but that this adaptation is implicit, this making executors model both `sender` and `scheduler` implicitly. The paper does not discuss the reason for the implicit wrapping.

We encourage exploring a design where `executor` implicitly models `scheduler`, and `schedule(e)` is a syntax transforming an `executor` into a `sender`, without a special case in `execution::connect`. The current automatic wrapping is inefficient due to being partial, for example `execution::submit(e, r)` falls back to the default cause that uses `submit-receiver` and causes allocation, while submit can be implemented more efficiently for `executor` as:

```
template<typename Receiver>
void executor_sender::submit(Receiver&& r)
{
    execution::execute(e, [r = std::forward<R>(r)] {
         try {
            execution::set_value(r);
        } catch {
            execution::set_error(r, std::current_exception());
        }
    });
}
```

(Because receivers are movable).

The paper also does not explain why `sender` can implicitly be treated as an `executor` via `execution::execute`. The problem with that wrapping is that it chooses one error handling strategy (terminate) when others may be possible, e.g. if a passed `invocable` has an `exception_ptr` overload, or receives an `error_code` argument. We encourage the authors to explore removing this implicit adaptation and eventually provide `as_receiver` as a library utility.

The authors of 0443 have noted this is a sensible direction.

This is issue https://github.com/atomgalaxy/review-executor-sendrecv/issues/4

2.9 Simplify types in concept definitions

This one is editorial.

In multiple places, the document refers to types using a phrase like the one below (this specific one is from 2.2.3.4):

```
... let `E` be a type such that `decltype((e))` is `E` ...
```

This should be simplified to:

```
... let `E` be `decltype((e))` ...
```

Casey Carter has acknowledged this and it seems to make sense.

2.10 Implementation of critical section-capable scheduling, a need to extract scheduler from context

This is issue https://github.com/atomgalaxy/review-executor-sendrecv/issues/1.

Tomasz implemented a scheduler and support for critical-section capable sender, receiver, and scheduler, along with an `async_mutex`.

The major requirement for the projects were:

— avoid dynamic allocation
— provide RAII semantics (mutex is automatically locked/unlocked)
— preserve the execution context of the work

The final requirement is not expressible with the current design (P0443), as to resume an “blocked” work, one needs to have the ability to extract the scheduler from the current execution context.

Initially, Tomasz used his own `sender_with_scheduler` concept to provide this functionality, as `schedule_provider`, as proposed in P1898, seemed unsuitable - it was impossible to continue on the same executor where `set_value` was invoked, and not the one provided by the associated `receiver`.

After a discussion with the authors in the issue, the group found a way to use the proposed `schedule_provider` for this particular case. However, the semantic (forward/backward) for the propagation of the scheduler is still unclear.

We suggest another example be provided to avoid further misunderstandings regarding the intended usage.

2.11 Forward/Backward propagation of executors

This is related to the extension of the senders/receiver design, proposed in P1898. However, this touches a very basic semantics of the algorithms defined in terms of sender/receiver, and we think that it would be beneficial to clarify this up-front.

With the `schedule_provider` concept it is possible to have two competing `schedulers` for the single operation, and it is unclear where it will be actually executed.

Specifically:

```
auto s1 = on(std::move(previous), sch1);
auto s2 = work(std::move(s1));
auto s3 = on(std::move(s2), sch2);
```

Does `work` run on the `sch1` propagated from `s1` or use `s2` provided by `schedule_provider` `on(..., sch2)`, and why?

Another example:

```
on(just(val), sch);
```

Does `just` extract the scheduler from the receiver of `on` and run there, or does `just` execute inline?

3 References

[P0443R13] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, D. S. Hollman, Lee Howes, Kirk Shoop, Lewis Baker, Eric Niebler. 2020. A Unified Executors Proposal for C++. https://wg21.link/p0443r13