When Do You Know `connect` Doesn't Throw?

Document Number: P3388R0
Date: 2024-09-09
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG

Abstract

This paper considers/proposes two tweaks which will enable earlier determination that connecting a certain sender with a certain receiver will never throw an exception.

Background

In order to be useful, senders must be connected to a receiver (by `std::execution::connect`). Only thereafter can the asynchronous operation represented by that sender be started (by `std::execution::start` on the operation state resulting from `std::execution::connect`). `std::execution` permits the former to throw exceptions but does not allow the latter to throw exceptions ([1] at §34.8.1).

Incredibly banal `std::execution` use can be divided into three phases:

1. Selection of sender and receiver
2. Connecting the chosen sender and receiver
3. Starting the resulting operation state

Both 1 and 2 are able to throw whereas 3 cannot. Therefore a transition from regular, synchronous code consists of two setup phases which can throw, followed by a transition to asynchronous code which cannot throw (instead errors must be transmitted via the error channel of the corresponding receiver).

More interesting uses of `std::execution` don't just involve step 1, followed by step 2, followed by step 3. The steps loop and nest. An asynchronous operation starts, selects senders and receivers, connects them to form a new operation state, and starts that operation from within its asynchronous context (for example `std::execution::let_value`).

All of the above takes place behind the `noexcept` barrier of `std::execution::start`. Therefore an exception resulting from any of the aforementioned steps must be transmitted by `std::execution::set_error`.

Regular, synchronous functions may emit exceptions or they may not. They can declare this property via the `noexcept` annotation. Examination of `noexcept`-ness can be used to decide at compile time whether a function throws depending on the properties of the operations it composes. Faithfully annotating functions with this information can lead to optimizations (for example `std::move_if_noexcept`).

Analogously senders may send exceptions (by sending `std::exception_ptr` down the error channel) or they may not. They can declare this property through the presence or absence of `std::set_error_t(std::exception_ptr)` in the collection of completion signatures which may be interrogated via `std::execution::completion_signatures_of_t`. Senders that compose other senders can inspect the completion signatures of the senders they compose to determine whether or not they should include `std::set_error_t(std::exception_ptr)` transitively in their list of completion signatures.

Note, however, that `std::execution::completion_signatures_of_t` only accepts the type of the:

- Sender, and
- Environment

Which means that when computing the values, errors, et cetera it sends a sender only has access to the type of the environment ([1] at §34.4), not the type of the receiver to which it will be connected. This decision/design is neither accidental nor arbitrary but is rather informed by the fact that attempting to provide both the type of the sender and receiver at this point can lead to circular type dependencies which will not compile. The indirection of the environment was introduced intentionally to address this.

Discussion

Salient Properties of a Receiver

In the analogy between synchronous and asynchronous code a receiver fills the niche of the return channel of a synchronous function [2]. This analogy has predictive power when considering the `get_env` member function of receivers (which is called by `std::execution::get_env`), which provides the receiver's "environment:" Synchronous code runs in the "environment" of the code to which it returns.

Continuing in the above vein synchronous code may have different behaviors in different environments. This is unsurprising. However it would be quite surprising if synchronous code had different behaviors depending on the location to which it is returning (one would be astonished, for example, to find code which inspects the call stack and behaves differently depending on which function it was called from).

Returning our analysis to the asynchronous domain we can therefore reason that the environment in which an asynchronous operation runs (i.e. the environment provided by the receiver) can affect the properties of that operation (verily this is the raison d'être of environments). This expectation is borne out by the fact that `std::execution::completion_signatures_of_t` accepts the environment type as a template parameter.

On the other hand the exact location to which the asynchronous operation is “returning” (i.e. particularities of the receiver beyond its environment) should be of no consequence. If some property of a sender is true (or false) when connected to a receiver of some type with a certain environment it should also be true (or false) when connected to a receiver of some other type with that same environment.

We can formalize the above: The only salient property of a receiver vis-à-vis a sender is its environment.

This formulation raises a problem (see the next section) and a question: What is meant by “its environment?” Does it refer to the exact type of the environment or some other means of reckoning same-ness of environments? One could even go so far as to recursively define salience for environments and say that the salient properties of an environment are:

- The set of queries it supports, and
- The values yielded by those queries

And therefore say that two environments:

- With the same set of queries, and
- Yielding the same values from those queries

Should be indistinguishable.

Throwing Move Constructors

Despite the neatness of the analogy presented above there is at least one important difference between the point to which a synchronous function returns and the receiver provided when connecting a sender to form an asynchronous operation: One (the receiver) must be stored. The point to which a synchronous function returns exists before the function is called, and continues to exist until the function returns, the function must do nothing to preserve this point. A receiver on the other hand is a C++ object and must therefore be stored in the operation state of the asynchronous operation (otherwise the receiver contract ([1] at §5.1) could not be fulfilled).

Since receivers are provided to the sender’s `connect` member function [3] as fully-materialized values they must be either copied or moved to propagate their value into the resulting operation state. This serves as a counterexample to the preceding section: Whether copy and move

operations throw is a property of a particular receiver type, and may not be discriminated based solely on the environment type associated therewith.

There is a simple solution for copy operations: Simply do not allow them to occur in the context of `connect`. There are two common/obvious ways to declare a sender's `connect` member function, either accepting the receiver by:

- Forwarding reference, or
- Value

In the former case copy operations can be pushed into `connect` (and must therefore be considered by the `noexcept` clause thereof). In the latter case the copy operation takes place in the context of the caller (and therefore need not be considered by the `noexcept` clause thereof).

In the case of move operations a solution is not so simple. Due to the fact the receiver is a fully-materialized value when provided to `connect` it must be either copied or moved therein. If not copied (see above) it must be moved. And while throwing moves are not necessarily common, or advisable, they are possible in general and must remain so [4].

Fortunately the strongest arguments in favor of throwing moves are those which deal with legacy (i.e. pre-C++11 and move semantics) types. As `std::execution` is being introduced to the language more than a decade after move semantics the possibility of any receiver type being "legacy" is effectively zero.

As such the above can be worked around by simply banning throwing move construction as part of the `std::execution::receiver` concept. This would restore the property (see previous section) that the only salient property of a receiver is its environment since the interaction between a sender and receiver would consist of the sender:

- Persisting the receiver (i.e. moving it, which must be possible ([1] at §34.7.1)),
- Obtaining the environment therefrom, and
- Completing the receiver contract therewith (which is already `noexcept` (id. at §34.7.2, §34.7.3, and §34.7.4))

And if:

- The `std::execution::receiver` concept mandated nothrow movability, and
- Senders' `connect` member functions accepted receivers by value (or equivalent)

Persisting the receiver could not throw.

Consistent `connect` `noexcept`-ness

The above sections detail that, if `std::execution::receiver` requires nothrow move construction, the only salient property of a receiver vis-à-vis a sender is the environment it

provides. The environment is available when determining completion signatures and therefore this provides a pathway to a solution to the problem laid out at the beginning of this paper.

However convincing ourselves that something is true does not mean, or require, it to be true. Therefore in order for this to constitute a solution to the problem we must required that when a sender type's `connect` member function is `noexcept` when instantiated for one type of receiver it must also be `noexcept` when instantiated on some other type of receiver which is "indistinguishable" (see above) therefrom. This would constitute a new semantic requirement on sender types.

Checking `connect` `noexcept`-ness

The problem this paper aims to solve is determining whether `std::execution::set_error_t(std::exception_ptr)` is one of the completion signatures for an operation which connects and starts a suboperation as part of its asynchronous part. Simply requiring that the `noexcept`-ness of connecting that suboperation's sender and receiver is consistent (see above) is insufficient to address this: It must also be possible to inspect this `noexcept`-ness in a context where no receiver, only the environment, is available (i.e. when computing its completion signatures).

There are two pathways to accomplishing this:

- Provide a new way to query senders to determine whether `connect` never throws for receivers with a certain environment, or
- Inspect the `noexcept`-ness of `connect` when instantiated on a receiver archetype which is associated with the environment of interest

The former is arguably simpler for consumers thereof. Additionally it may reduce the number of template instantiations necessary to form the completion signatures of the sender. However it:

- Pushes additional boilerplate onto sender authors, and
- Introduces a possible anomaly in the interface of senders (i.e. the result of the query may not actually match the `noexcept`-ness of `connect`)

The latter can be addressed by, as a matter of convention, always annotating `connect` with a `noexcept` clause which is in terms of the above-described query. However this constitutes additional boilerplate.

The latter option requires no new code/boilerplate from sender authors, but may come at the expense of consumer complexity (which could be mitigated by providing a free function or trait which cans the computation) and additional template instantiations.

Proposal

Require that receivers be nothrow movable as part of the `std::execution::receiver` concept.

Additionally, either:

- Introduce a semantic requirement that: If connecting an instance of a certain sender type to an instance of a certain receiver type doesn't throw, that connecting an instance of that sender type to an instance of any receiver type with the same associated environment type cannot throw,
- Similar to the above but strengthen the requirements such that a different associated environment type can be provided assuming that environment type supports exactly the same set of queries and yields exactly the same types thereto, or
- Introduce a query on senders which allows them to report whether or not they are always nothrow connectible to receivers with a certain associated environment type

Acknowledgements

The author would like to thank Lewis Baker and Eric Niebler for extensive discussion and insights regarding this issue.

References

[1] M. Dominiak et al. std::execution P2300R10
[2] R. Leahy. Of Operation States and Their Lifetimes P3373R0
[3] V. Voutilainen. Member customization points for Senders and Receivers P2855R0
[4] V. Voutilainen. We cannot (realistically) get rid of throwing moves P0129R0