Document #:P3090R0

Date: 2024-02-12

Project: Programming Language C++

Audience: Library Evolution Work Group

Reply-to:

Inbal Levi <sinbal2l@gmail.com>

Eric Niebler <eric.niebler@gmail.com>

`std::execution` Introduction

Abstract

P2300 introduced the sender/receiver async framework. Such a standard framework will give the broader C++ ecosystem a composable *lingua franca* that all libraries can use to express asynchrony.

This paper aims to provide a high-level introduction to the design of P2300 and its guiding principles. For detailed information on each component, please see P2300, P3143, and the experimental cppreference "execution" page.

Motivation for P2300

Mission and values

The mission of P2300 can be stated as:

To provide the basis for an ecosystem of standard and third-party async algorithms and execution contexts.

The following values were of foremost importance to the authors of P2300, in priority order:

1.	Safety :	aim for correct-by-construction design
2.	Composability :	ability to treat async components as “black boxes” that compose
3.	Universality :	able to accommodate any source of asynchrony
4.	Efficiency :	little to no abstraction penalty, few allocations, non-blocking
5.	Scalability :	support asynchronous systems of any scale

In accordance with the above, the asynchronous model of P2300 is designed to enable developers to:

➔ Work generically with *any* source of asynchrony, whether multi-threading, file or network IO, operating system timers and interrupts, or coprocessors and hardware accelerators like GPUs.
➔ Address the concern of **what** to execute separately from the concern of **where**.
➔ Compose asynchronous operations into non-blocking task graphs in a platform-agnostic way.
➔ Launch and orchestrate work that spans multiple different execution contexts.
➔ Cleanly respond to and propagate error conditions and cancellation requests.
➔ Integrate seamlessly with coroutines.
➔ Scale efficiently from single-threaded, zero-allocation embedded systems all the way to massively networked supercomputing clusters.

The hope is that asynchronous APIs will return objects that satisfy – or that are easily convertible to objects that satisfy – the *sender* concept described below. This allows them to be `co_await`-ed in coroutines and used with a standard suite of sender-based generic algorithms, which in turn will foster an ecosystem of third-party algorithms and execution contexts.

Structured concurrency

The design of P2300 encourages the use of *structured concurrency* constructs, where child operations complete before parents, analogous to how functions complete before their callers, allowing P2300 to be alignd with coroutines, which are similarly structured. Structured concurrency (as with structured programming in general) allows components to be treated as black boxes with well-defined control flow that is easy to reason about.
It is essential to making concurrency **safely composable**.

P2300’s design fits comfortably in, and relies heavily on, the C++ object model by supporting nested lifetimes, ordered construction and destruction, scopes and local variables. It obviates most uses of reference counting for lifetime management.

Tiered design

P2300 is build with a tiered (layered) design:

- Facilities for consumers of asynchrony:
    - These users will see only senders and schedulers (and not with the internal components).
    - They will compose senders into other senders (by pipe or by algorithm), or that await them in coroutines, or that block waiting for their completion.
- Facilities for producers of asynchrony:
    - These are developers extending the framework with custom senders, algorithms, and schedulers.
    - These developers will be writing “receivers” (structured callbacks) and “operation states”, and will be following the sender/receiver protocol to ensure composability with other sender-based facilities.

This paper doesn't distinguish between these user- and implementer-oriented facilities; it aims to show how all of these parts fit together. However, the "hello world" example below shows how things may look from a "naive" user's perspective.

Introduction to the Framework

Basic Building Blocks

- **Sender**: A description of asynchronous work to be sent for execution. Produces an operation state (below).
  - Senders asynchronously "send" their results to listeners called "receivers" (below).
  - Senders can be composed into **task graphs** using generic algorithms.
  - **Sender factories and adaptors** are generic algorithms that capture common async patterns in objects satisfying the `sender` concept.
- **Receiver**: A generalized callback that consumes or "receives" the asynchronous results produced by a sender.
  - Receivers have three different "channels" through which a sender may propagate results: success, failure, and canceled, so-named "value", "error", and "stopped".
  - Receivers provide an extensible **execution environment**: a set of key/value pairs that the consumer can use to parameterize the asynchronous operation.
- **Operation State**: An object that contains the state needed by the asynchronous operation.
  - A sender and receiver are connected when they are passed to the `std::execution::connect` customization point.
  - The result of connecting a sender and a receiver is an operation state.
  - Work is not enqueued for execution until "start" is called on an operation state.
  - Once started, the operation state's lifetime cannot end before the async operation is complete, and its address must be stable.
- **Scheduler**: A lightweight handle to an execution context.
  - An execution context is a source of asynchronous execution such as a thread pool or a GPU stream.
  - A scheduler is a factory for a sender that completes its receiver from a thread of execution owned by the execution context.

Execution Guarantees

Every asynchronous operation (*i.e.*, the result of connecting a sender and a receiver), once started, is guaranteed[1] to call exactly one of the following members on the receiver:

1. set_value
2. set_error
3. set_stopped

These functions shall not be called before start is called. Once one of these is called, the **contract between the sender and receiver is satisfied**, and the operation state can be destroyed.

[1] Unless the operation never completes.

Analyzing an Example Sender Pipeline

In this section we will break the code into:

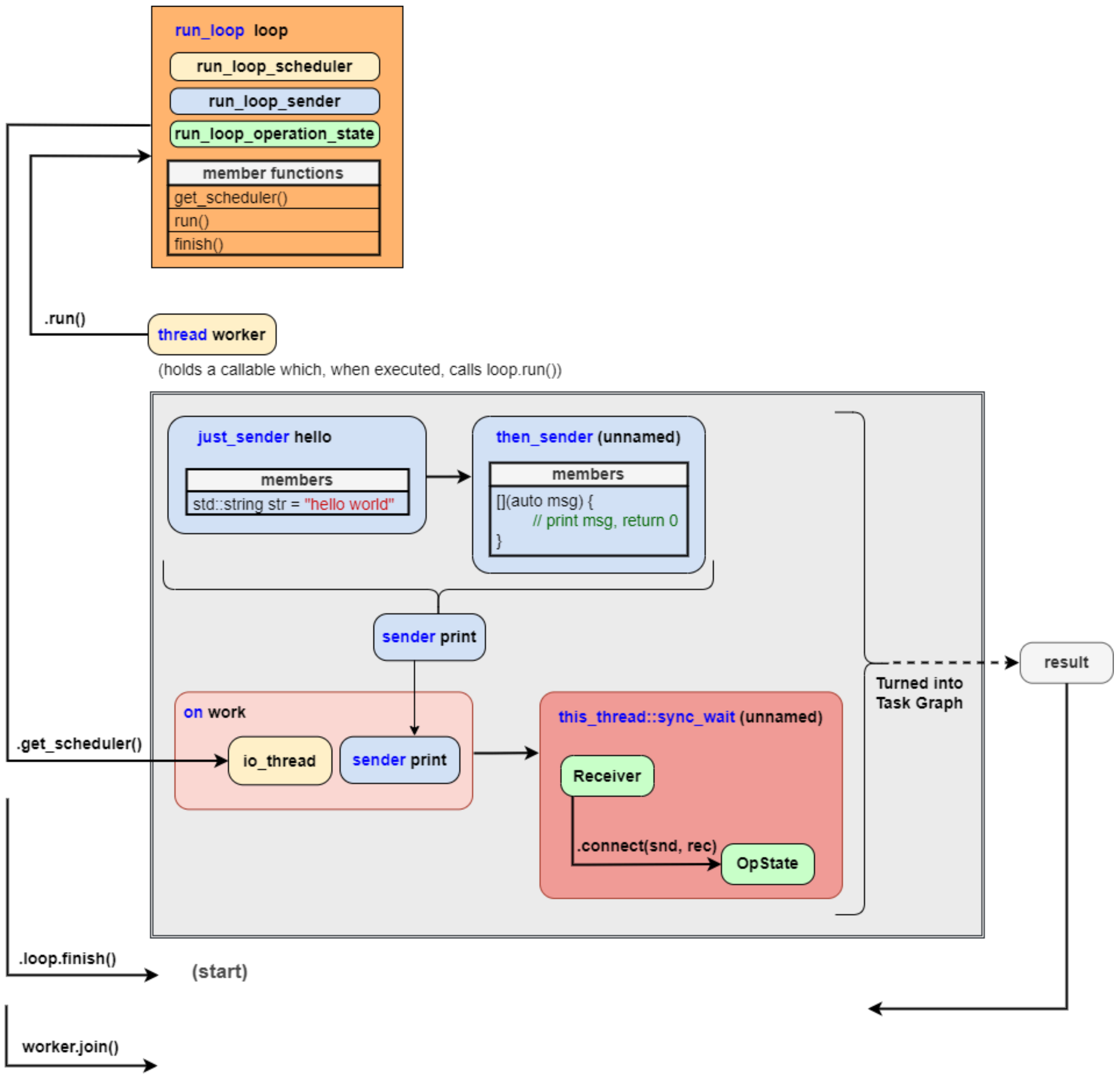
1. Compile time – composing a sender and connecting it to a receiver), and
2. Runtime – executing the sender on the worker thread driving the run_loop.

```cpp
#include <thread>
#include <iostream>
#include <execution>
using namespace std::literals;
namespace stdex = std::execution;

stdex::run_loop loop;
std::thread worker([]{ loop.run(); });

int main()
{
   stdex::sender auto hello = stdex::just("hello world"s);
   stdex::sender auto print = hello
                            | stdex::then([](auto msg) {
                                std::puts(msg.c_str());
                                return 0; // This will be returned as the
                                          // result of the async op
                              });

   stdex::scheduler auto io_thread = loop.get_scheduler();
   stdex::sender auto work = stdex::on(io_thread, print);

   auto [result] = std::this_thread::sync_wait(work).value();
   loop.finish(); // Close registration of tasks (senders) on the loop
   worker.join(); // Blocks till the worker thread exits
   return result; // return 0
}
```

https://godbolt.org/z/b8bWYMbqW

Note: Some details (such as std::move) in the code snippet above were dropped for the sake of simplicity.

Note: The following diagram highlights the main parts of the framework. Detailed flow can be found in [P3143](P3143).



User facing utilities:

Execution resource	- Provides execution context
Scheduler	- A handle to the execution resource
Sender	- A description of asynchronous work
Sender Adaptors	- Algorithm which transform one sender into another
Sender Consumers	- Algorithm which takes a sender and does not return sender

Implementer facing utilities:

OpState	- The reification of an asynchronous operation
Receiver	- Callback provided by the consumer

The code will be build and run as execution in two stages:

Stage 1: Composition

Building the **task graph**:

1. `just("hello…")` creates a sender that, when connected to a receiver and started, passes the string "hello world" to the receiver's value callback.
2. The `then` sender adapts the `just` sender, causing the `just` sender's result value to be passed to a user-specified invocable. It does that by `connect`-ing the `just` sender with a custom receiver. The result of the invocable is then passed to the receiver connected to the `then` sender.
3. The `on` algorithm takes the `then(just(…),…)` sender and a scheduler from the `run_loop` context, and returns a sender that – when connected and started – will cause the then/just sender to start execution on the thread driving the `run_loop`.
4. The `sync_wait` algorithm connects the sender to a receiver, calls start on the resulting operation state, and blocks waiting on a condition variable.

Stage 2: Execution

Calling "start" on the work, which will cause the asynchronous operation to be enqueued for execution:

1. The start call cascades through the operation states "outside in":
   a. The "on" algorithm, when started, will start the work produced by the `run_loop` scheduler, which causes a work item to be inserted into the `run_loop`'s queue.
   b. That work item will "start" the `then(just(…),…)` operation,
   c. … which "start"s the `just` operation,
   d. … which passes "hello world" to `then`'s receiver,
   e. … which passes the string to the invocable, passing the resulting integer to the `on` receiver,
   f. … which passes the results to `sync_wait`'s receiver, which saves the results into a buffer, sets a condition variable signaling completion, and then returns the results to the caller.
2. `loop.finish()` signals the loop that we're not going to add more work to the queue, causing `run()` to return the next time the queue is empty.
3. `worker.join()` blocks the main thread until the worker thread exits.

Summary

Producer of asynchronous interfaces offer a way to obtain a sender as the unit of asynchronous work. Consumers of asynchronous interfaces can then await the work in a coroutine or avoid the coroutine frame allocation by using a generic algorithm instead from the standard library or from an ecosystem of sender-savvy third-party libraries.

Acknowledgements

Thanks to:

- Lewis Baker and Ian Petersen for reviewing the paper thoroughly.
- Kirk Shoop, Lucian Radu Teodorescu, Gašper Ažman and Maikel Nadolski for providing valuable insights.