Bloomberg Analysis of Unified Executors

Document #: P2024R0
Date: 2020-01-10
Project: Programming Language C++
Library Evolution
Reply-to: Dietmar Kühl
<dkuhl@bloomberg.net>
Frank Birbacher
<frank.birbacher@gmail.com>
Marina Efimova
<mefimova1@bloomberg.net>
Michael V. Riedlin
<mriedlin@bloomberg.net>
David Sankel
<dsankel@bloomberg.net>

Contents

1 Abstract

[P0443R11] proposes the addition of an executor and sender/receiver framework to the C++ Standard Library. The November 2019 discussion in Library Evolution called for a study of this framework with the (non-binding) goal of progressing the proposal at the February 2020 meeting. Bloomberg formed a study group of experts in

Networking and API design to review the proposal and this paper summarizes the results of this study. Our preliminary conclusion is that the approach is promising, but lacks maturity; more user experience and study is necessary to determine if it is appropriate for standardization.

2 Introduction

The C++ standard sorely lacks basic vocabulary types and concepts for asynchronous code. Not only does this deficiency prevent Open Source asynchronous code from interoperating, but it has stalled progress on important standardization proposals such as Futures and Networking.

Given this state of affairs, the authors would very much like to see a reasonable solution progress to the C++ standard. This paper is intended to aid this effort by supplying feedback on the proposal from a developer's perspective and a preliminary evaluation of the suitability of [P0443R11] for standardization.

2.1 Methodology

At Bloomberg it is very common for applications to use an in-house RPC framework (BAS; Bloomberg Application Services). The typical application of this framework is using synchronous operations although the framework does support asynchronous operations. There is a desire within the company to simplify the usage of asynchronous operations. To see how [P0443R11] could affect our infrastructure we implemented very simplified versions of a request using different parts proposed by [P0443R11].

In addition to the experiments we reviewed the design and compared it to other similar designs, in particular to type classes in Haskell.

2.2 Brief Review of [P0443R11]

2.2.1 Executor Framework

Executors[1] provide C++ programmers a way to control the execution process of a function object, i.e. *where* to execute and *how* to execute. The *where* is an *execution context* that represents resources available for execution. A thread pool, for example, is a familiar *execution context*. *Execution agents* utilize the resources of an execution context in order to invoke a callable object. Thus, the execution agent ties an operation and *execution resources* together. The `executor` concept provides an interface to create execution agents on a specified execution context. From a developer's perspective, an `executor` is a factory that creates *execution agents*, entities that associate a function object with a resource involved in its processing.

Executor properties impose requirements on executors, specifying *how* the executor behaves. This is particularly useful in generic contexts where, e.g., a class or function requires its executor template parameter to have certain behaviors.

Consider the "blocking" property, which describes the guaranteed blocking behavior an executor provides for its execution function:

Property Object Name	Requirements
blocking.possibly	Invocation of an executor's execution function may block pending completion of one or more invocations of the submitted function object
blocking.always	Invocation of an executor's execution function shall block until completion of all invocations of submitted function object.
blocking.never	Invocation of an executor's execution function shall not block pending completion of the invocations of the submitted function object.

The property mechanism is described in a separate paper, [P1393R0]. Three property "extraction" functions are

[1] See [P0761R2] for details

available[2]:

```
auto e1 = std::require(e0, p) // e1 is a version of e0 that has property p
auto e1 = std::prefer(e0, p) // e1 is a version of e0 that has property p if possible
auto v = std::query(e0, p) // v is the value of the p property for e0
```

This property mechanism (which makes use of "customization point objects") is available so that user-defined properties (in addition to several properties defined in the standard) can be associated with executors.

The core operations on executors are customization point objects as well. These are as follows:

```
execution::execute(e, f)         // execute the function object f on the executor e
execution::bulk_execute(e, f, n) // execute the function object f on the executor e, n times
```

Every executor type must be `nothrow_copy_constructable`, `nothrow_destructible`, and `equality_comparable`. Executors are additionally required to provide at least an implementation for the `execution::execute` customization point object.

[P0443R11] additionally introduces the `any_executor` class template. This template implements a type-erased executor that has certain properties that are accessible to the property mechanism.

The following member functions are of note:

```
template <class... SupportableProperties>
class any_executor
{
  public:
    //...

    // Return an 'any_executor' object corresponding to 'std::require(e, p)',
    // where 'e' is the underlying object of *this. 'Property' must be in
    // 'SupportableProperties' and "requirable".
    template <class Property>
      any_executor require(Property p) const;

    // Return a Property-specific "polymorphic" type corresponding to
    // 'std::query(e, p) where 'e' is the underlying object of *this.
    // 'Property' must be in 'SupportableProperties' and "queryable".
    template <class Property>
      typename Property::polymorphic_query_result_type query(Property p) const;

    // Return 'true' if *this has an underlying object, and 'false' otherwise.
    explicit operator bool() const noexcept;

    // Return the underlying executor if it has type 'Executor'.
    template<executor Executor> Executor* target() noexcept;
    template<executor Executor> const Executor* target() const noexcept;
};
```

2.2.2 Sender/Receiver Framework

The sender/receiver framework aims at the ability to create asynchronous algorithms. The corresponding concepts are intended to become the fundamental building blocks for asynchronous algorithms similar to how iterators are the building blocks for algorithms on sequences. The central aspect of senders/receivers is to decouple the creation of the operation and its actual launch: the algorithm reifies what needs to be done into a sender object.

[2] Variadic versions are omitted for brevity.

To actually execute anything the sender gets connected to a receiver yielding another object which can then be started in a suitable context to kick off the operation which eventually gets to the result delivery.

The results are delivered to a receiver through function calls: `set_value(std::move(receiver), result)` is called upon successful execution. Calling a function does allow delivery of multiple results without combining them into an object like a `std::tuple<...>`. Different kinds of results can be delivered by calling overloaded functions, i.e., it isn't necessary to aggregate all possible results int a `std::variant<...>`.

When an operation cannot run to successful completion it delivers its result by calling `set_error(std::move(receiver), error)`. This function can be overloaded but has exactly one error parameter. Arbitrary error types rather than a specific set of error types covered by overloads can be delivered through an `std::exception_ptr` or a similar facility. Using a separate function to deliver errors allows asynchronous algorithms to detect that results can't be fully achieved on the path executed so far and use an alternative path.

As algorithms may start multiple asynchronous operations whose results may become unnecessary when some of the operations deliver a result it is useful to cancel operations. A cancelled operation delivers its result by calling `set_done(std::move(receiver))`.

The use of futures and promises doesn't support composing different asynchronous operations because the operation computing the result is already scheduled when a future is received. As a result, there is a need for synchronization and the operations are type erased.

In [P0443R11] the fundamental customization points are defined. This paper does not propose any asynchronous algorithms. The experimental [Unifex] library from Facebook implements the various customization points and adds some asynchronous algorithms. This library differs in some ways from the interfaces proposed by [P0443R11]. For example, the `submit(sender, receiver)` function from [P0443R11] is called `connect(sender, receiver)`. However, our experiments are based on [Unifex].

2.2.2.1 Thoughts on Using Senders

The previous section outlined how senders and receivers interact. Additionally [Unifex] provides some basic algorithms to combine senders into new senders, such as `sequence` and `when_all`. The following paragraphs discuss a few questions on their design.

Senders can represent asynchronous work items such as reading from or writing to a socket. These operations result in some data processing that may or may not have a natural result. A read operation naturally delivers the data read in some way, while a write operation doesn't have any useful results besides errors. Since errors are handled with a separate `set_error` function, the `set_value` function just wouldn't receive any arguments and that's fine.

When using `sequence` to compose several actions, all actions, save the last, are required to not produce arguments for `set_value`. If such a sequence starts with a socket read operation the resulting data cannot be delivered through the `set_value` channel. Instead, the user is required to communicate such data through a captured shared data structure. In order to have that data structure live through the process, the user will need to use the provided `let` or use management facilities such as shared pointers to control the lifetime.

Intuition somehow would suggest that results from one step will be delivered to the next. Instead the design is in a way that every step of a `sequence` cannot actually consume the results of the previous step. We're left unclear about why the design was chosen this way and how it helps a user write asynchronous programs.

On the other hand `when_all` contains a fair amount of logic just to support correctly forwarding any kinds of results. All of the combined senders may produce any set of arguments for `set_value` and the sender returned by `when_all` will send a tuple of variants of all possible argument combinations that are announced by the individual senders. If, however, the programming model in general favors senders that don't produce arguments and instead communicate via shared state then the `when_all` can be simplified. It would not forward results and require senders to not deliver any arguments, just like `sequence` does. It's unclear what the intended use is.

Additionally these functions take the list of senders as variadic arguments. We need to see how to process a dynamic list of items in the same way. For example given a vector of URL, how to download all of them and do

something when all are done? The `when_all` doesn't really provide for that.

2.2.2.2 Comparison to Haskell Type Classes

The previous section discussed some unclarity about whether senders are generally expected to produce values or not. This section will continue on these thoughts and compare senders to monads and related type classes found in Haskell.

The `sequence` algorithm seems to prefer no results while `when_all` produces a tuple of the collected results of the combined senders. From what we observed so far a sender conceptually has some associated result type like a regular function has a return type. The type can be observed with the nested template alias `value_types` where an empty tuple stands in for `void`.

In this sense a sender conceptually encapsulates a value that is to be produced when the operation it represents is run and finishes. This sounds a lot like a monad which also represents one or more values produced by giving the monad to some sort of executing function. In [Unifex] this would be the `start` function on the object returned by `connect`.

In Unifex, connecting a sender to a receiver yields an operation that can be "started." Looking at the implementation of `sequence` and `when_all`, this is done inside some sender that is the result of these algorithms. So, in order to code up a sender that combines other senders the implementation will probably create a receiver, call `connect`, and then call `start`. This is all to have means to observe the values produced by the given senders, process them, and present the whole thing as a new sender while being oblivious of the operational details of the sources. This basically resembles monadic binding.

A monadic value in Haskell represents an operation with a result and can be used any number of times in order to execute this operation any number of times. This is like functions that can be called any number of times. However, senders are for one-time use only despite looking like a monad. For every use they need to be created again, thus all the code to combine them must also be executed again. Senders don't truly represent just the operation as a value. With this in mind, for an asynchronous program written against Boost.ASIO with callbacks or even with coroutines, how would senders improve that code? Senders don't seem to allow to code the structure of the asynchronous control flow separate from tying in all relevant objects.

The relationship to monads goes further. The `let` construct solves an object life-time issue, but also looks just like a monadic bind. The `just` sender implements the monadic `return` which is the same as the applicative `pure` from Haskell. The more fundamental `fmap` is found in the `transform` algorithm.

3 Bloomberg Use-cases

To evaluate [P0443R11], we attempted to solve some asynchrony problems that are representative to the kind of work we do at Bloomberg. This research is exploratory and is meant to demonstrate use cases of asynchronous programming implemented in terms of executors and sender/receivers rather than verify the correctness of the chosen implementation or measure its performance.

3.1 Creation of a Thread Pool Executor

Before we illustrate our core examples, we first create an executor that schedules using Bloomberg's thread pool library[3]. We will compare this to the thread pool executor found in the example code found in Eric Niebler's executors-impl GitHub repository [threadpoolexecutor1].

The interface to a BDE thread pool `bdlmt::ThreadPool` is rather small. Apart from setup and tear down functions there is only one member for scheduling a function execution on any thread of the pool: `enqueueJob(job)`. The `job` parameter is a `bsl::function<void()>`[4]. An executor wrapping one such thread pool should call `enqueueJob` from within its `execute` member function. Because the executor is a cheaply copyable object it should reference the pool by pointer. This also gives rise to the definition of equality.

[3] See bdlmt_threadpool in the [BDE] library.

[4] `bsl::function` is an allocator aware variation of `std::function`.

```
// Wrap our own thread pool into an executor
class PoolExecutor {
    bdlmt::ThreadPool *d_pool;
public:
    explicit PoolExecutor(bdlmt::ThreadPool *);
    void execute(const bsl::function<void()>& job);
    friend bool operator==(PoolExecutor lhs, PoolExecutor rhs) = default;

    // ... more here which is to be discussed ...
};

//...

int main()
{
    bdlmt::ThreadPool pool(40);
    TicketClient client(PoolExecutor(&pool));
}
```

The part of the `PoolExecutor` implementation shown above is intuitive and doesn't require much thought. However, we're missing the support for properties. As can be seen in the example implementation [threadpoolexecutor1] this requires providing a range of `query` and `require` member functions overloaded on properties and their values. If this, as it seems, is necessary in order to implement a new executor it would actually bind this implementation to the list of properties known at that time. The amount of boilerplate around properties needs to be justified by the benefits we get from them.

It's unclear whether the proposed set of properties has been discovered as a universal set of necessary properties or whether this set has been invented to fit current needs. Can users add to this set? It seems the answer is yes. Are new properties expected to work with existing executors, e.g. the ones provided by an C++ implementation? It seems the answer is no.

3.2 General Description of `TicketClient`

Within Bloomberg there is a ticketing system used to raise issues for various teams. The aforementioned BAS RPC framework is used to operate programmatically with the system. For our examples we created various forms of a `TicketClient` class that facilitates access to the system. It has a single operation, `getInfo` that, when provided a ticket id, asynchronously fetches information about the corresponding ticket.

The intent is that the client of `getInfo`, aside from specifying *what* should be done with the ticket information, should specify *where* to schedule/execute that work.

3.3 Templated, Executor-based `TicketClient`

Our first implementation of `TicketClient` will be executor based and takes various executors by template arguments.

The first thing to note is that `TicketClient` will be performing network IO operations. The executor used to schedule these operations cannot be any arbitrary executor as it needs to support, for example, asynchronous write and read operations. To accomplish this we create custom operations that are enabled for any executor that has appropriate support[5].

We define customization point objects for operations `connect`, `read` and `write` on a socket:

```
class const_buffer;
```

[5] Note that in a post-networking TS world these operations will more likely be associated with an execution context.

```
inline constexpr struct connect_socket_cpo
{
    template <typename Socket, typename Address>
    auto operator()(Socket& socket, Address const& address) const
    {
        return unifex::tag_invoke(*this, socket, address);
    }
} connect_socket;

inline constexpr struct async_write_cpo
{
    template <typename Socket>
    auto operator()(Socket& socket, const_buffer const& buffer) const
    {
        return unifex::tag_invoke(*this, socket, buffer);
    }
} async_write;

inline constexpr struct async_read_cpo
{
    template <typename Socket>
    auto operator()(Socket& socket, bsl::vector<char>& buffer) const
    {
        return unifex::tag_invoke(*this, socket, buffer);
    }
} async_read;
```

Using *executor-of-impl* concept proposed in the [P0443R11] and the defined `connect_socket`, `async_write` and `async_read` we can introduce a `net_io_executor` concept:

```
template<class S, class B>
concept net_io_invocable =
requires(S& s, B const& b){
    async_write(s, b);
} &&
requires(S& s, B& b){
    async_read(s, b);
};

template<class E>
concept net_io_executor = std::executor-of-impl<E, net_io_invocable>;
```

The interface of our `TicketClient` class now makes use of `net_io_executor`:

```
template<net_io_executor IOExecutor>
class TicketClient
{
  IOExecutor d_ioExecutor;
  public:
    TicketClient(IOExecutor exec) : d_ioExecutor(exec) {}
    //...
};
```

The `getInfo` function is implemented with a callback argument. It takes an additional executor parameter so

where the callback is run is explicitly specified by the caller[6].

```
template<net_io_executor IOExecutor>
class TicketClient
{
    //...

    // Asynchronously call the 'getInfo' request for ticket service and execute
    // the specified 'callback' with the result on the specified 'executor'.
    template<std::executor Exec>
    void getInfo( int ticketId,
                  std::function<void (std::error_code, TicketInfo)> callback,
                  Exec executor );
};
```

One open question we have is whether or not 'IOExecutor' could have (or should have) alternatively been implemented with a new executor "requires" property. Our thinking is that such a `TicketClient` would look as follows:

```
template<std::executor IOExecutor>
requires std::can_require_v(IOExecutor, IOSchedulerProperty)
class TicketClient
{
  decltype(std::require(std::declval<IOExecutor>(), ioSchedulerProperty)) d_ioExecutor;
  public:
    TicketClient(Exec exec) : d_ioExecutor(std::require(exec, has_io_scheduler)) {}
  //...
};
```

3.4 Non-templated, Executor-based `TicketClient`

Our next variant of `TicketClient` intends to make use of polymorphic executors. Which executor used with `TicketClient` should, in theory, be selectable at runtime. While we could figure out how appropriately restrict polymorphic executors using properties, we were unable to figure out a good way how to use them for running custom operations.

```
class TicketClient
{
  std::any_executor<IOSchedulerProperty> d_ioExecutor;
  public:
    TicketClient(std::any_executor<IOSchedulerProperty> exec)
        : d_ioExecutor(exec.require(ioSchedulerProperty)) {}

    void getInfo(
        int ticketId,
        std::function<void (std::error_code, TicketInfo)> cb,
        std::any_executor<> executor )
    {
        // ??? How to access custom operations on d_ioExecutor.
    }
};
```

One possible avenue would be to use `any_executor::query` call with a type-erasing function collection type specified in `IOSchedulerProperty::polymorphic_query_result_type`, but more time is required to investigate this approach.

[6] While the caller could specify this implicitly in the function object it provides, that approach is generally considered error-prone.

3.5 Sender/Receiver `TicketClient`

In this section we describe the implementation of `TicketClient` using the sender/receiver framework. This approach is substantially different from those used in our prior examples.

For the networking we created a simple networking library which is somewhat modelled after the Networking TS [networkingts] but only implements what is needed for this experiment. It changed the `async_*` operations to return a Sender instead of using a completion token. We have not, yet, tried to integrate the Sender concept into the completion token although doing so could be possible.

Implementing I/O operations using Senders was relatively straightforward once we understood the concept. So far our code doesn't abstract any of the Sender implementation into components. As a result there is quite a bit of boilerplate code for each of these operations. For example, the implementation of `async_write` looks like this:

```
template <typename Protocol>
class basic_stream_socket
   : public basic_socket<Protocol>
{
public:
   // ...
   class write_sender { ... };

   friend write_sender tag_invoke(unifex::tag_t<async_write>,
                                  basic_stream_socket& socket,
                                  const_buffer const&  buffer)
   {
       return write_sender(&socket, buffer);
   }
};
```

`async_write` is a customization point object. This use is an application of the proposed approach for defining customizable functions [tag_invoke]. The Unifex library uses this approach through its implementation. The actual reification of the asynchronous write is `write_sender`:

```
class write_sender
{
private:
   basic_stream_socket *d_socket;
   const_buffer         d_buffer;

public:
   template <template <typename...> class Variant>
   using error_types = Variant<std::error_code>;
   template <
       template <typename...> class Variant,
       template <typename...> class Tuple>
   using value_types = Variant<Tuple<>>;

   template <typename Receiver>
   class operation { ... };

   write_sender(basic_stream_socket*, const_buffer const&);

   template <typename Receiver>
   operation<std::decay_t<Receiver>> connect(Receiver&& r);
};
```

First, the `write_sender` captures the arguments for the operation in its constructor for later execution. The alias template `error_types` and `value_types` can be used to determine the possible parameter types for the `set_error()` (in this case just `std::error_code`) and the possible parameter types for `set_value()` (in this case no argument). The declarations are used by asynchronous algorithms for forwarding the results.

The class template `operation` represents the operation state and is obtained from the `write_sender` when connecting it to a receiver. The `connect()` function just creates this object and directly returns. The `operation` objects are not required to copyable or movable and are constructed into their final location using copy-elision. The receiver is only move constructible:

```
template <typename Receiver>
operation<std::decay_t<Receiver>> connect(Receiver&& r)
{
   return operation<std::decay_t<Receiver>>(this->d_socket,
                                            this->d_buffer,
                                            std::forward<Receiver>(r));
}
```

All the actual execution happens in the class template `operation`. As it reifies the actual operation and the destination where to send the result it also holds the arguments in addition to a receiver:

```
template <typename Receiver>
class operation
{
private:
   basic_stream_socket *d_socket;
   const_buffer         d_buffer;
   Receiver             d_receiver;

public:
   template <typename R>
   operation(basic_stream_socket*, const_buffer const&, R&&);

   void start() noexcept { this->write(this->d_buffer); }
   void write(const_buffer const& buffer) noexcept {
       int rc = ::write(this->socket->native_handle(),
                        buffer.data(), buffer.size());
       if (rc < 0) {
           unifex::set_error(std::move(this->d_receiver),
                             std::error_code(errno, std::system_category()));
       }
       else if (std::size_t(rc) < buffer.size())
       {
           this->d_socket->d_context->add_work(
               this->d_socket->native_handle(),
               POLLOUT,
               [this, buffer=buffer + rc, receiver](short){
               this->write(buffer);
           });
       }
       else
       {
           unifex::set_value(std::move(this->d_receiver));
       }
   }
};
```

The operation is kicked off by calling `start()`. In this case this call just delegates to `write()` which tries to write the buffer until either an error is encountered or the entire buffer was written. The socket is set up to be non-blocking. If there is remaining work, the next call to `write()` is added to the `io_context` associated with the socket. The implementation specific function `add_work()` takes the `native_handle()` of the socket, the condition when to execute the function (in this case when `poll()` indicates that some output can be written), and the function to be executed (in this case `write()` with the remaining buffer). Upon error the receiver's `set_error()` function is called; upon success the receiver's `set_value()` function is called.

After we implemented this version we realized that it could be implemented in terms of the corresponding Networking TS version: invoking the `async_write()` with a suitable callback from the `operation::start()` function would delivery of the results to the receiver too.

We used correspondingly implemented `async_read` and `async_connnect` functions to implement a simple client request to a server. The first attempt incorrectly created a dedicated sender for this purpose; while doable, that approach was unnecessarily complex. It is much easier to use [Unifex] functions to create an asynchronous request:

```
class TicketClient
{
public:
   // ...
   auto asyncGetInfo(int id)
   {
       bsl::string req("get Ticket " + bsl::to_string(id) + "\n");
       return unifex::transform(
           unifex::sequence(
               unifex::let(unifex::just(req),
                   [this](bsl::string const& r){
                       return async_write(
                           this->d_socket,
                           const_buffer(r.data(), r.size()));
                    }),
               async_read(this->d_socket, buffer(this->d_buffer),
                   [b = &this->d_buffer, p = 0u](bsl::error_code const&,
                                                 bsl::size_t s) mutable{
                       auto e = bsl::find(b->begin() + p, b->begin() + s, '\n');
                       return e == b->begin() + (p = s)? b->end() - e: 0;
                   })
           ),
           [this](bsl::size_t size) {
               return TicketInfo(begin(this->d_buffer),
                                 begin(this->d_buffer) + size);
           }
       );
   }
   // ...
};
```

This implementation uses `unifex::let()` to pass an object living long enough to `async_write()` to finish. `unifex::sequence(s1, ..., sn)` starts the next sender in the sequence once a sender is finished. This way the read operation isn't started before the write operation completed. The `async_read()` uses a condition determining whether a complete line was read and pretends that there isn't any data following a newline to keep thing simple (a real implementation would buffer any excess data for the next read). Finally, the result is converted into the destination type using `unifex::transform()`.

Note that nothing in `asyncGetInfo()` actually starts an operation: the result is a sender which can be connected

to a receiver. This operation could be use like this:

```
unifex::sync_wait(
    unifex::sequence(
        client.asyncConnect(endpoint),
        unifex::transform(client.asyncGetInfo(123),
        [](TicketClient::TicketInfo const& value){
            std::cout << "asyncGetInfo(123)='"
                      << std::string(begin(value), end(value))
                      << "'\n";
        })
    )
);
```

The resulting asynchronous code looks reasonably readable. On the other hand during development we did encounter some rather unreadable error messages. These will hopefully be improved when the various operations require appropriate concepts. Once we got more familiar with the use of Unifex, though, we got better at guessing what the error is actually hinting at without trying to read all of the errors.

For some of the code tested we observed fairly long compile-times. These happened when using a slightly bigger number of operations inside a `sequence()`.

4 Our Analysis

4.1 Maturity of Sender/Receiver Framework

The sender/receiver abstraction seems to be rather fresh and somewhat complicated. The assessment of complexity may be due to being unfamiliar. Using alias templates taking template template arguments to create a structure of tuples nested inside a variant is surprising although it is helpful for computing actual types.

The concepts and algorithms in the Unifex library also seem a bit immature. For example, there is a concept for `ManySender` which calls `set_value()` any number of times and finishes with a call to either `set_done()` or `set_error()`. That is, a normal sequence of values is terminated with `set_done()`. For a Sender `set_done()` indicates an incomplete result, such as a cancelled operation. For a `ManySender` there is no obvious way to distinguish between a complete sequence and a partial sequence that was prematurely cancelled.

For the algorithm `when_all()` the result is a `std::tuple` (one entry for each sender) of `std::variant` of `std::tuple`s (one `std::tuple` for each result of the corresponding sender). It seems the outer tuple isn't really needed and could be replaced by multiple arguments to `set_value()`. Likewise, it seems odd that the `sequence()` algorithm doesn't propagate results between consecutive operations.

These examples don't imply that the entire approach is flawed and the design may actually be exactly correct. However, we have the impression that different choices can be made and may be a better fit.

It seems the Unifex library is the first publicly available implementation of these ideas. Correspondingly, there is a lack of user experience. It would be good to see some successful use of these abstractions, ideally in an Open Source project before dropping such a library into the standard.

4.2 Thoughts on `any_executor`

We were not able to get much implementation experience with `any_executor` in time for this paper but did have some thoughts.

For a typical application, we aren't sure `any_executor` is a compelling construct, particularly if the set of properties is for introspection only. For our representative TicketClient, it's sufficient to have the single executor which the application developer could choose or otherwise be aware of. If properties are only for introspection, they seem to have little to offer as a developer could simply choose an executor with the required properties without the added boilerplate.

It seems somewhat compelling to have the type erased `any_executor` as an enforcement mechanism. In this case though, it seems like defining a narrower custom concept as we did for the templated TicketClient above, would be more straightforward.

We would like to see examples using `any_executor` that better demonstrate its capabilities.

4.3 Potential Division of Communities

During the LEWG discussion the concern was raised that the communities between users of executors and senders/receivers would be fragmented. This concern doesn't seem to be warranted: executors and senders/receivers compose well. The point of senders/receivers is to produce a reification of an operation which can be further composed to achieve a higher-level goal. Eventually, this operation needs to be executed which can be done by executing the operation on an executor.

The Unifex library uses the `on(sender, scheduler)` algorithm to yield a Sender indicating that the operation was scheduled or failed to do so as its result. The Unifex library so far doesn't have direct support for executors but there are plans to add these.

5 Conclusion

The executor and sender/receiver framework in [P0443R11] are promising as a means to provide core vocabulary types for asynchrony in the C++ standard. While they may be the right abstractions for this work, our analysis concluded that more reports of user-experience would be required to make that determination.

6 References

[BDE] 2020. BDE Library.
https://github.com/bloomberg/bde/wiki

[networkingts] 2018. Working Draft, C++ Extensions for Networking.
http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/n4734.pdf

[P0443R11] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, David Hollman, Lee Howes, Kirk Shoop, Eric Niebler. 2019. A Unified Executors Proposal for C++.
https://wg21.link/p0443r11

[P0761R2] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, Michael Wong. 2018. Executors Design Document.
https://wg21.link/p0761r2

[P1393R0] David Hollman, Chris Kohlhoff, Bryce Lelbach, Jared Hoberock, Gordon Brown, Michał Dominiak. 2019. A General Property Customization Mechanism.
https://wg21.link/p1393r0

[tag_invoke] 2019. tag_invoke: A general pattern for supporting customisable functions.
http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1895r0.pdf

[threadpoolexecutor1] 2018. Thread Pool Executor with Property Support.
https://github.com/ericniebler/executors-impl/blob/5f25108b760e2f59707299739896f98efba59cc0/include/experimental/bits/static_thread_pool.h

[Unifex] 2020. libunifex.
https://github.com/facebookexperimental/libunifex