async-resource - aka async-RAII

Draft Proposal

Document #:	D0000R0
Date:	2023-04-09
Project:	Programming Language C++
Audience:	LEWG Library Evolution
Reply-to:	Kirk Shoop <kirk.shoop@gmail.com> Ville Voutilainen <ville.voutilainen@gmail.com>

Contents

1 Introduction

This paper describes concepts that would be used to create and cleanup an object within an *async-scope* that will contain all *async-function*s composed into that *async-scope*. These *async-function*s have access to a non-owning handle to the *async-resource* that is safe to use. These *async-function*s can be running on any execution context. An *async-resource* object has only one concern, which is to open before any nested *async-function*s start and to close after any nested *async-function* complete. In order to be useful within other asynchronous scopes, the object must not have any blocking functions.

An *async-resource* can be thought of as an async-RAII object.

1.1 What is an *async-resource*?

An *async-resource* is an object with state that is valid for all the *async-function*s that are nested within the *async_resource* expression.

Examples include:

- thread
- thread-pool
- io-pool
- buffer-pool
- mutex
- file
- socket
- *async-scope*

2 Motivation

It is becoming apparent that all the sender/receiver features are language features being implemented in library.

Sender/Receiver itself is the implementation of an *async-function*. It supports values and errors and cancellation. It also requires manual memory management in implementations because *async-resource*s do not fit inside any single block in the language.

A major precept of [P2300R6] is structured concurrency. The `let_value()` algorithm provides stable storage for values produced by the input *async-function*. What is missing is a way to attach an object to a sender expression such that the object is opened before nested *async-function*s start and is closed after nested *async-function*s complete. This is commonly done in async programs using `std::shared_ptr` to implement ad-hoc garbage collection. Using garbage collection for this purpose removes structure from the code, because the shared ownership is unstructured and allows objects to escape the original scope in which they were created.

The C++ language has a set of rules that are applied in a code-block to describe when construction and destruction occur, and when values are valid. The language implements those rules.

This paper describes how to implement rules for the construction and destruction of async objects in the library. This paper describes structured construction and destruction of objects in terms of *async-function*s. The `use_resources()` algorithm described in this paper is a library implementation of an async code-block containing one or more local variables. The `use_resources()` algorithm is somewhat analogous to the `using` keyword in some languages.

3 Design

3.1 What is the rational for this design?

The rationale for this design is that it unifies and generalizes asynchronous construction and destruction, making the construction adaptable via sender algorithms. Its success cases are handled by what follows an `open()` *async-function*, its failure cases are handled as results of the `run()` *async-function*. The success case isn't run at all if `open()` fails, quite like what follows RAII initialization isn't performed if the RAII initialization fails.

Furthermore, asynchronous resources are acquired only when needed by asynchronous work, and that acquisition can itself be asynchronous. As a practical example, consider a thread pool that has a static amount of threads in it. With this approach, the threads can be spun up when needed by asynchronous work, and no sooner - and the threads are spun up asynchronously, without blocking, but the "success" case, i.e. the code that uses the threads, is run after the threads have been spun up.

The communication between `open()` and `run()` on an asynchronous resource is implicit, as is the communication between `close()` and `run()` on an asynchronous resource. The rationale for this choice is that it more closely models how language-level scopes work - you don't need to 'connect' the success code to a preceding RAII initialization, the success code just follows the RAII initialization once the initialization is complete. Likewise, there's no need to 'connect' `close()` to a completion of `run()`, that happens implicitly, quite like destruction implicitly follows exiting a scope.

3.2 What are the requirements for an *async-resource*?

async construction

Some objects have *async-function*s to establish a connection, open a file, etc..

The design must allow for *async-function*s to be used during construction - without blocking any threads (this means that C++ constructors are unable to meet this requirement)

async destruction

Some objects have *async-function*s to teardown a connection, flush a file, etc..

The design must allow for *async-function*s to be used during destruction - without blocking any threads (this means that C++ destructors are unable to meet this requirement)

structured and correct-by-construction

These are derived from the rules in the language.

The object will not be available until it has been constructed. The object will not be available until the object is contained in an *async-function*. Failures of *async-construction* and *async-destruction* will complete to the containing *async-function* with the error. The object will always complete cleanup before completing to the containing *async-function*. Acquiring an object is a no-fail *async-function*.

composition

Multiple object will be available at the same time without nesting. Composition will support concurrent *async-construction* of multiple objects. Composition will support concurrent *async-destruction* of multiple objects. Dependencies between objects will be expressed by nesting. Concurrency between objects will be expressed by algorithms like `when_all` and `use_resources()`.

3.3 What is the concept that an *async-resource* must satisfy?

There are two options for defining the *async-resource* concept that are described here. Either one will satisfy the requirements.

3.3.1 run(), open(), and close()

This option uses three new CPOs in two concepts that describe the lifetime of an *async-resource*.

The `open()` and `close()` *async-functions* do no work and never complete with an error. The `open()` and `close()` *async-functions* provide access to signals from the internal states of the `run()` *async-function* before it completes.

This option depends only on [P2300R6]

3.3.1.1 async_resource Concept:

An *async-resource* stores the state used to open and run a resource.

```
/// @brief the async-resource concept definition
template<class _T>
concept async_resource =
  requires (const _T& __t_clv, _T& __t_lv){
    open_t{}(__t_clv);
    run_t{}(__t_lv);
  };

using open_t = /*implementation-defined/*;
/// @brief the open() cpo provides a sender that will complete with a async-resource-token.
/// @details The async-resource-token will be valid until the sender provided
/// by close() is started.
/// The sender provided by open() will complete after the sender provided by
/// run() has completed any async-operation needed to open the resource.
/// The sender provided by open() will not fail.
/// @param async-resource&
/// @returns sender<resource-token>
inline static constexpr open_t open{};

using run_t = /*implementation-defined/*;
/// @brief the run() cpo provides a sender-of-void.
/// @details The sender provided by run() will start any async-operation
/// needed to open the resource and when all those operations complete
/// then run() will complete the sender provided by open().
/// The sender provided by run() will complete after the sender provided
/// by close() is started and all the async-operation needed to close
/// the async-resource complete and the sender provided by close() is completed.
/// @param async-resource&
/// @returns sender<>
inline static constexpr run_t run{};
```

3.3.1.2 async_resource_token Concept:

An *async-resource-token* is a non-owning handle to the resource that is provided after the resource has been opened.

The token must be used to close the resource once the resource has been opened.

```
/// @brief the async-resource-token concept definition
template<class _T>
concept async_resource_token =
  requires (const _T& __t_clv){
    close_t{}(__t_clv);
```

```
  };

using close_t = /*implementation-defined/*;
/// @brief the close() cpo provides a sender-of-void.
/// @details The sender provided by close() will trigger the sender provided
/// by run() to begin any async-operation needed to close the resource and
/// will complete when all the async-operation complete.
/// The sender provided by close() will not fail.
/// @param async-resource-token&
/// @returns sender<>
inline static constexpr close_t close{};
```

3.3.2 run() -> *sequence-sender*

This option uses one new CPO in one concept that describes the lifetime of an *async-resource*.

This option depends on a paper that adds *sequence-sender* on top of [P2300R6]

```
/// @brief the async-resource concept definition
template<class _T>
concept async_resource =
  requires (_T& __t){
    run_t{}(__t);
  };

using run_t = /*implementation-defined/*;
/// @brief the run() cpo provides a sequence-sender-of-token.
/// @details The sequence-sender provided by run() will produce
/// a run-operation. When the run-operation is started, it will start
/// any async-operation that are needed to open the async-resource.
/// After all those async-operation complete, the run-operation
/// will produce an async-resource-token as the only item in the
/// sequence.
/// After the sender-expression for the async-resource-token item
/// completes, the run-operation will start any async-operation
/// that are needed to close the async-resource.
/// After all those async-operation complete, the run-operation
/// will complete.
/// @param async-resource&
/// @returns sequence-sender<async-resource-token>
inline static constexpr run_t run{};
```

3.4 How do these CPOs compose to provide an async resource?

3.4.1 run(), open(), and close()

The `open()` *async-function* and the `run()` *async-function* are invoked concurrently.

After both of the open and run operations are started, `run()` invokes any *async-function* that is needed to initialize the *async-resource*.

After all those *async-operation* complete, then `run()` signals to `open()` which then will complete with the *async-resource-token*.

`run()` will complete after the following steps:

— the runtime has entered the `main()` function (requires a signal from the runtime)

— any *async-operation* needed to open the *async-resource* has completed

at this point, the *async-resource* lifetime begins

— `open()` completes with the *async-resource-token*

— a stop condition is encountered

— a `stop_token`, provided by the environment that invoked `open()`, is in the `stop_requested()` state

OR

— the `close()` *async-function* has been invoked

OR

— the runtime has exited the `main()` function (this requires a signal from the runtime)

at this point, the *async-resource* lifetime ends

— any *async-operation* needed to close the *async-resource* have completed

— `close()` completes

3.4.2 run() -> *sequence-sender*

The *sequence-sender* returned from `run()` produces a *run-operation*.

After the *run-operation* is started, it starts any *async-operation* that are needed to initialize the *async-resource*.

After all those *async-operation* complete, the *run-operation* will emit the *async-resource-token* as the only item in the sequence.

The *run-operation*, will complete after the following steps:

— the runtime has entered the `main()` function (this requires a signal from the runtime)

— any *async-operation* needed to open the *async-resource* has completed

at this point, the *async-resource* lifetime begins

— the *async-resource-token* item is emitted

— a stop condition is encountered

— a `stop_token`, provided by the environment of the *open-operation*, is in the `stop_requested()` state

OR

— the *token-operation*, produced by the sender expression for the *async-resource-token* item, has completed

OR

— the runtime has exited the `main()` function (this requires a signal from the runtime)

at this point, the *async-resource* lifetime ends

— any *async-operation* needed to close the *async-resource* have completed

3.5 How do you use an *async-resource*?

Here is a basic example of composing resources using this pattern:

Table 1: basic example

run(), open(), and close()

```
int main() {
  exec::static_thread_pool ctx{1};
  exec::counting_scope context;
  auto use = ex::when_all(
      exec::open(ctx),
      exec::open(context)) |
    ex::let_value([&](
      ex::scheduler auto sch,
      exec::async_scope auto scope){
        // async-resource lifetime begins

        sender auto begin =
          ex::schedule(sch);

        sender auto printVoid =
          ex::then(begin,
            []()noexcept { printf("void\n"); })

        exec::spawn(scope, printVoid);

        // async-resource lifetime ends
        // when close starts
        return ex::when_all(
          exec::close(sch), exec::close(scope))
      });
  ex::sync_wait(ex::when_all(
    use,
    exec::run(ctx),
    exec::run(context)));
}
```

run() -> *sequence-sender*

```
int main() {
  exec::static_thread_pool ctx{1};
  exec::counting_scope context;
  auto use = ex::zip(
      exec::run(ctx),
      exec::run(context)) |
    ex::let_value_each([&](
      ex::scheduler auto sch,
      exec::async_scope auto scope){
        // async-resource lifetime begins

        sender auto begin =
          ex::schedule(sch);

        sender auto printVoid =
          ex::then(begin,
            []()noexcept { printf("void\n"); });

        exec::spawn(scope, printVoid);

        // async-resource lifetime ends
        // when printVoid completes
        return printVoid;
      });
  ex::sync_wait(use);
}
```

This pattern correctly scopes the use of the *async-resource* and composes the open, run, and close *async-operations* correctly.

It is possible to compose multiple *async-resource*s into the same block or expression.

Table 2: multiple *async-resource* composition example

run(), open(), and close()

```
stop_source stp;
static_thread_pool ctx{1};
async_allocator aa;
counting_scope as;
async_socket askt;
split spl;

auto use = when_all(
  open(stp), open(ctx), open(aa),
  open(as), open(askt), open(spl))
  | let_value([](
    stop_token stop, auto sched, auto alloc,
    auto scope, auto sock, auto splt){
    auto env = make_env(empty_env{},
      with(get_stop_token, stop),
      with(get_scheduler, sched),
      with(get_async_allocator, alloc),
      with(get_async_scope, scope));
    auto [input, output] = splt;
    auto producer = produce(input,
      async_read_some(sock, MAX_DATA_SIZE));
    for (int i = 0; i < 4; ++i) {
      spawn(scope,
        with_env(env,
          on(sched, consume(output))));
    }
    auto close = when_all(
      close(stop), close(sched), close(alloc),
      close(scope), close(sock), close(splt));
    return finally(
      with_env(env,
        when_all(producer,
          nest(consume(output)))),
      close);
  });

std::this_thread::sync_wait(
  when_all(
    use,
    run(stp), run(ctx), run(aa),
    run(as), run(askt), run(spl)));
```

run() -> *sequence-sender*

```
stop_source stp;
static_thread_pool ctx{1};
async_allocator aa;
counting_scope as;
async_socket askt;
split spl;

auto use = zip(
  run(stp), run(ctx), run(aa),
  run(as), run(askt), run(spl))
  | let_value_each([](
    stop_token stop, auto sched, auto alloc,
    auto scope, auto sock, auto splt){
    auto env = make_env(empty_env{},
      with(get_stop_token, stop),
      with(get_scheduler, sched),
      with(get_async_allocator, alloc),
      with(get_async_scope, scope));
    auto [input, output] = splt;
    auto producer = produce(input,
      async_read_some(sock, MAX_DATA_SIZE));
    for (int i = 0; i < 4; ++i) {
      spawn(scope,
        with_env(env,
          on(sched, consume(output))));
    }
    return with_env(env,
      when_all(producer,
        nest(consume(output))));
  });

std::this_thread::sync_wait(use);
```

Both of these options fall into a pattern as well. This pattern can be placed in an algorithm. The `use_resources` algorithm changes the above example to look like:

```
std::this_thread::sync_wait(
  use_resources([](
      auto stop, auto sched,
      auto alloc, auto scope,
```

```
    auto sock, auto split){
      auto [input, output] = split;
      auto env = make_env(empty_env{},
        with(get_stop_token, stop),
        with(get_scheduler, sched),
        with(get_async_allocator, alloc),
        with(get_async_scope, scope));
      auto producer = produce(input, async_read_some(sock, MAX_DATA_SIZE));
      for (int i = 0; i < 4; ++i) {
        spawn(scope, with_env(env, on(sched, consume(output))));
      }
      return with_env(env, when_all(producer, nest(consume(output))));
    },
    make_deferred<stop_source>(),
    make_deferred<static_thread_pool>(1),
    make_deferred<async_allocator>(),
    make_deferred<counting_scope>(),
    make_deferred<async_socket>(),
    make_deferred<split>()));
```

4 Why this design?

There have been many, many design options explored for the `async_scope`. We had a few variations of a single object with methods, then two objects with methods. It was at this point that a pattern began to form across `stop_source`/`stop_token`, *execution-context*/`scheduler`, *async-scope*/*async-scope-token*.

The patterns model RAII for objects used by *async-function*s.

In C++, RAII works by attaching the constructor and destructor to a block of code in a function. This pattern uses `run()` to represent the block that the object is contained in. The `run()` *async-function* satisfies the structure requirement (only nested *async-function*s use the object) and satisfies the correct-by-construction requirement (the object is not available until the `run()` *async-function* is started).

4.1 run(), open(), and close()

The run/open/close option uses an *async-function* to store the object (`run()`), another *async-function* to access the object (`open()`), and a final *async-function* to stop the object (`close()`). `open()` is not a constructor and `close()` is not a destructor. `open()` and `close()` are signals. `open()` signals that the object is ready to use and `close()` signals that the object is no longer needed. Any errors encountered in the object cause the `run()` *async-function* to complete with the error, but only after completing any cleanup *async-function*s needed.

4.1.1 The open cpo

`open` does not perform a task, its completion is a signal that `run` has successfully constructed the resource.

Existing resources, like `run_loop` and `stop_source` have a method that returns a token. This does not provide for any asynchronous operations that are required before a token is valid.

`open` is an operation that provides the token only after it is valid.

`open` completes when the token is valid. All operations using the token must be nested within the `run` operation (yes, it is the run operation that owns the resource, not the open operation).

The receiver passed to the `open` operation is used to query services as needed (allocator, scheduler, stop-token, etc..)

4.1.2 The run cpo

`open` may start before the resource is constructed and completes when the token is valid. `run` starts before the resources is constructed and completes after the token is closed. The `run` operation represents the entire resource. The `run` operation includes construction, open, resource usage, and close. `run` is the owner of the resource, `open` is the token accessor, `close` is the signal to stop the resource.

`open` cannot represent the resource because it will complete before the resources reaches the closed state.

`close` cannot represent the resource because it cannot begin until after open has completed.

4.1.3 The close cpo

`close` does not perform a task, its invocation is a signal that requests that the resource safely destruct.

`close` is used to start any operations that stop the resource and invalidate the token. After the `close` operation completes the `run` operation runs the destructor of the resource and completes.

4.1.4 Composition

The `open` and `close` cpos are not the only way to compose the token into a sender expression.

The benefit provided by the `open` and `close` operations is that a `when_all` of multiple `open`s and a `when_all` of multiple `close`s can be used to access multiple tokens without nesting each token inside the prev.

4.1.5 Structure

The `run`, `open`, and `close` operations provide the token in a structured manner. The token is not available until the `run` operation has started and the `open` operation has completed. The `run` will not complete until the `close` operation is started. This structure makes using the resource correct-by-construction. There is no resource until the `run` and `open` operations are started. The `run` operation will not complete until the `close` operation completes.

Ordering of constructors and destructors is expressed by nesting resources explicitly. Using `when_all` to compose resources concurrently requires that the resources are independent because there is no token to the resource available until the `when_all` completes.

4.2 run() -> *sequence-sender*

The run/sequence-sender option uses an *async-function* to store the object (`run()`). The sequence produces one item that provides access to the object once it is ready to use. When the item has been consumed, `run()` will cleanup the object and complete. Any errors encountered in the object cause the `run()` *async-function* to complete with the error, but only after completing any cleanup *async-function*s needed.

4.2.1 The run cpo

`run` starts before the resource is constructed and completes after all nested *async-function*s have completed and the object has finished any cleanup *async-function*. The `run` operation represents the entire resource. The `run` operation includes construction, resource usage, and destruction. `run` is the owner of the resource.

4.2.2 Composition

Composition is easily achieved using the `zip()` algorithm and the `let_value_each()` algorithm.

4.2.3 Structure

The `run` *async-function* provides the object in a structured manner. The object is not available until the `run` operation has started. The `run()` *async-function* will not complete until the object is no longer in use. This structure makes using the resource correct-by-construction. There is no resource until the `run()` *async-function* is started. The `run()` *async-function* completes after all nested *async-function*s have completed and the object has finished any cleanup *async-function*.

Ordering of constructors and destructors is expressed by nesting resources explicitly. Using the `zip()` algorithm to compose resources concurrently requires that the resources are independent because there is no token to the resource available until the `zip()` algorithm completes.

5 Algorithms

5.1 make_deferred

The `make_deferred` algorithm packages the constructor arguments for a type `T` and provides `void operator()()` that will construct `T` with the stored arguments when it is invoked.

The `make_deferred` algorithm returns a *deferred-object* that contains storage for `T` and for `ArgN...`.

Before `T` is constructed, the *deferred-object* copies and moves if the stored `ArgN...` supports the operations.

When the *deferred-object* is invoked as a function taking no arguments, `T` is constructed in the reserved storage for `T` using the `ArgN...` stored in the *deferred-object* when it was constructed.

Once `T` is constructed, attempts to copy and move the *deferred-object* will `terminate()`.

Once `T` is constructed in the *deferred-object*, `T` can be accessed with `T& operator->()` and `T& value()` and eagerly destructed with `void reset()`.

```
struct make_deferred_t {
  template<class T, class... ArgN>
  implementation-defined operator()(ArgN&&... argN) const;
};
static inline constexpr make_deferred_t make_deferred{};
```

5.2 use_resources

The `use_resource` algorithm composes multiple *async-resources* into one *async-function* that is returned as a sender.

The `use_resource` algorithm will use the selected option (run-open-close or run-sequence-sender) to apply all the *async-resource-tokens* for the constructed *async-resource*s to the single *body-function*.

When the returned *async-function* is invoked, it will invoke all the deferred *async-resource*s to construct them in its *operation-state* and then it will acquire the *async-resource-token* for each *async-resource* and then invoke the *body-function* once with all the tokens.

```
struct use_resources_t {
  template<class Body, class... AsyncResourcesDeferred>
  implementation-defined operator()(Body&& body, AsyncResourcesDeferred&&... resources) const;
};
static inline constexpr use_resource_t use_resources{};
```

6 Appendices

6.1 Rejected Options:

- `join() -> sender`:
  - `join()` returns a sender that completes after running any pending *async- operation*s followed by running any *async-operation*s needed to close the *async-resource*.

- This option is challenging because it is not correct by construction:
  - Imposes that all users remember to compose `join()` into an `async_scope` and prevent the destructor from running until `join()` completes.
  - Provides no way to run *async-operation*s to open the *async-resource*.
- `run((token)->sender) -> sender`:
  - The sender returned from `run()` will complete after the following steps:
    - *async-operation*s to open the *async-resource*

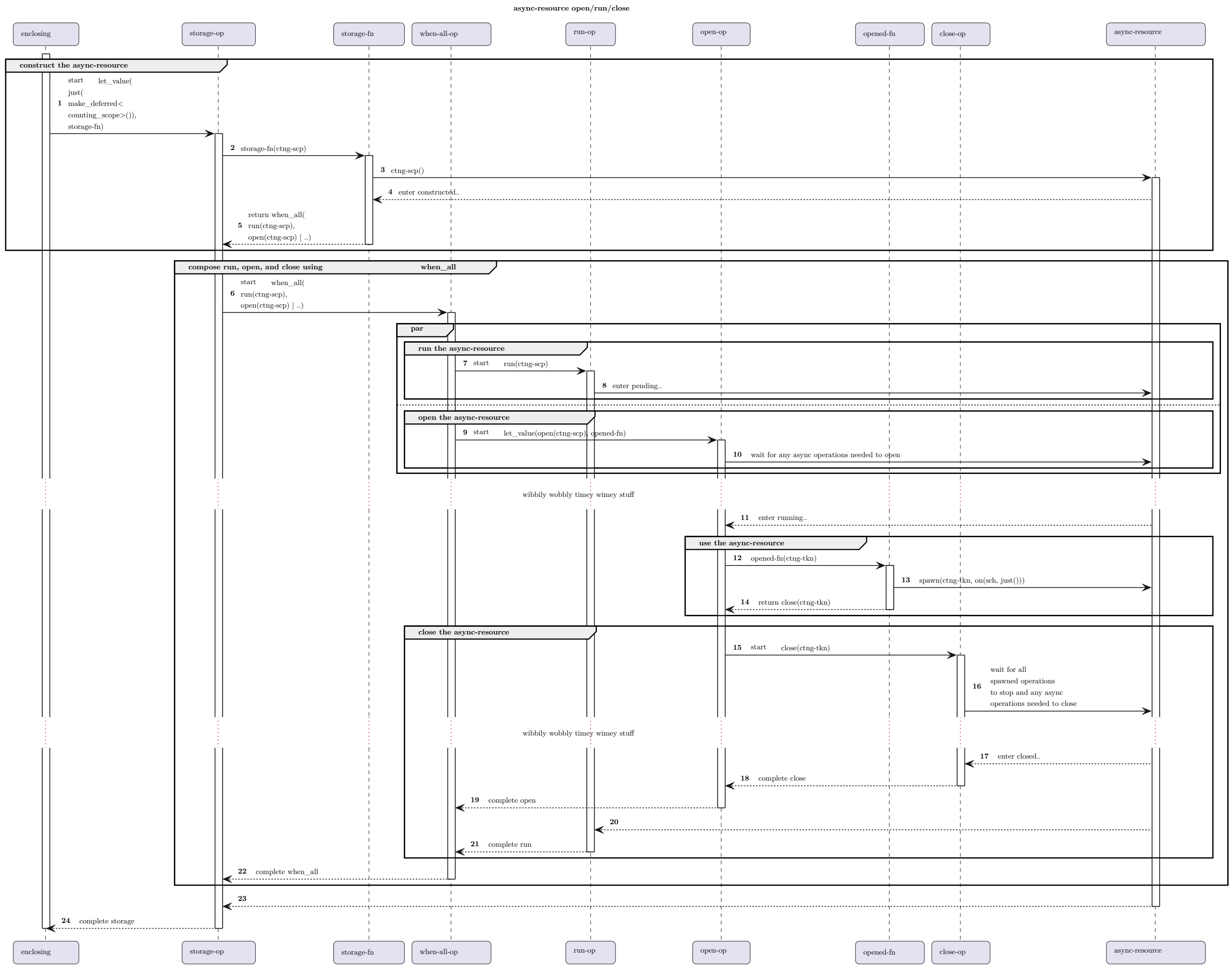
      ** at this point, *async-resource* lifetime begins **
    - an *async-resource-token* is passed to the provided function
    - the sender returned from the provided function

      ** at this point, *async-resource* lifetime ends **
    - any *async-operation*s needed to close the *async-resource*
  - This option scopes the use of the *async-resource* and composes the open and close *async-operation*s correctly.
  - It is hard to compose multiple *async-resource*s into the same block or expression (requires nesting calls to `run()` for each *async-resource*, which also sequences the open and close for each *async-resource*).

6.2 The order of operations when using an async-resource

6.2.1 One resource

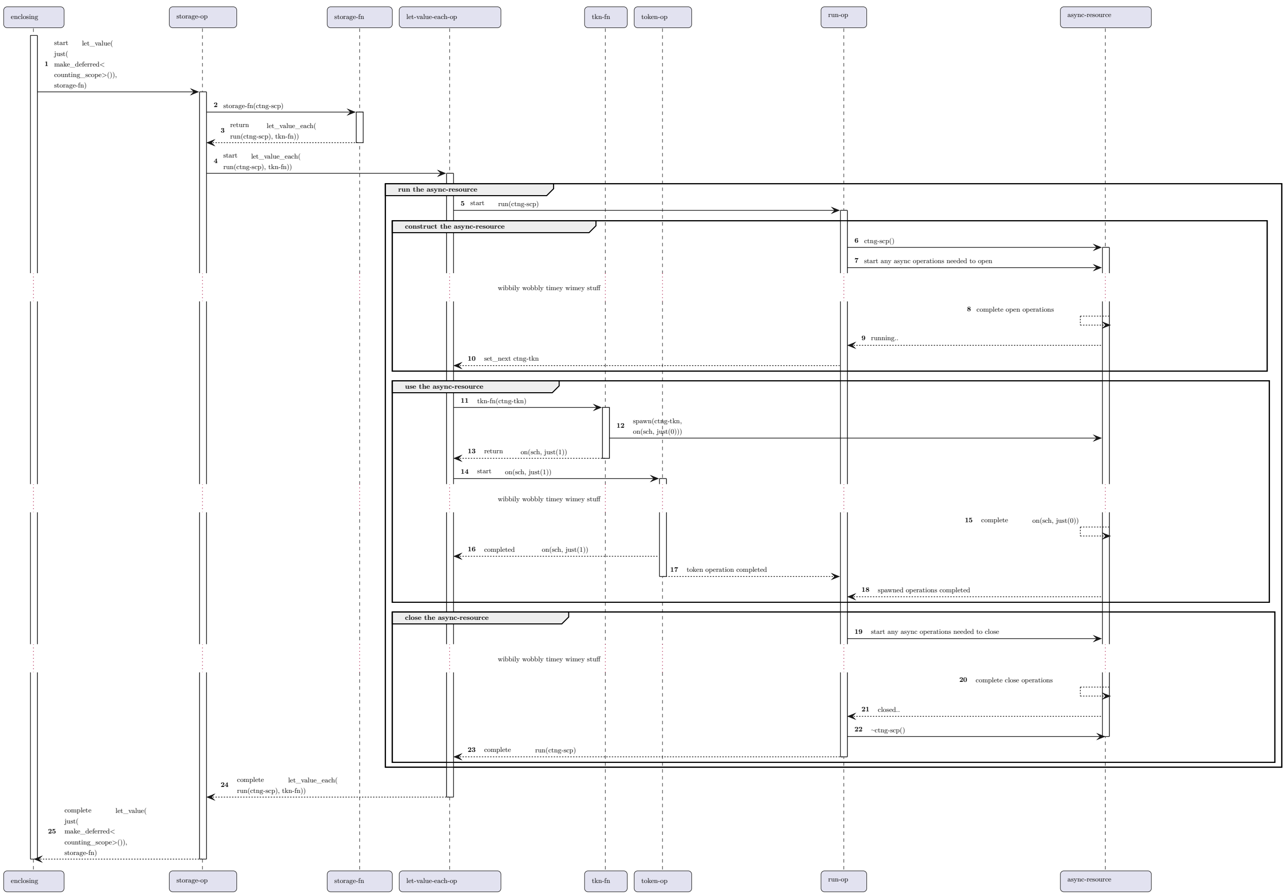
6.2.1.1 run(), open(), and close()



sequence diagram for async-resource

6.2.1.2 run() -> *sequence-sender*

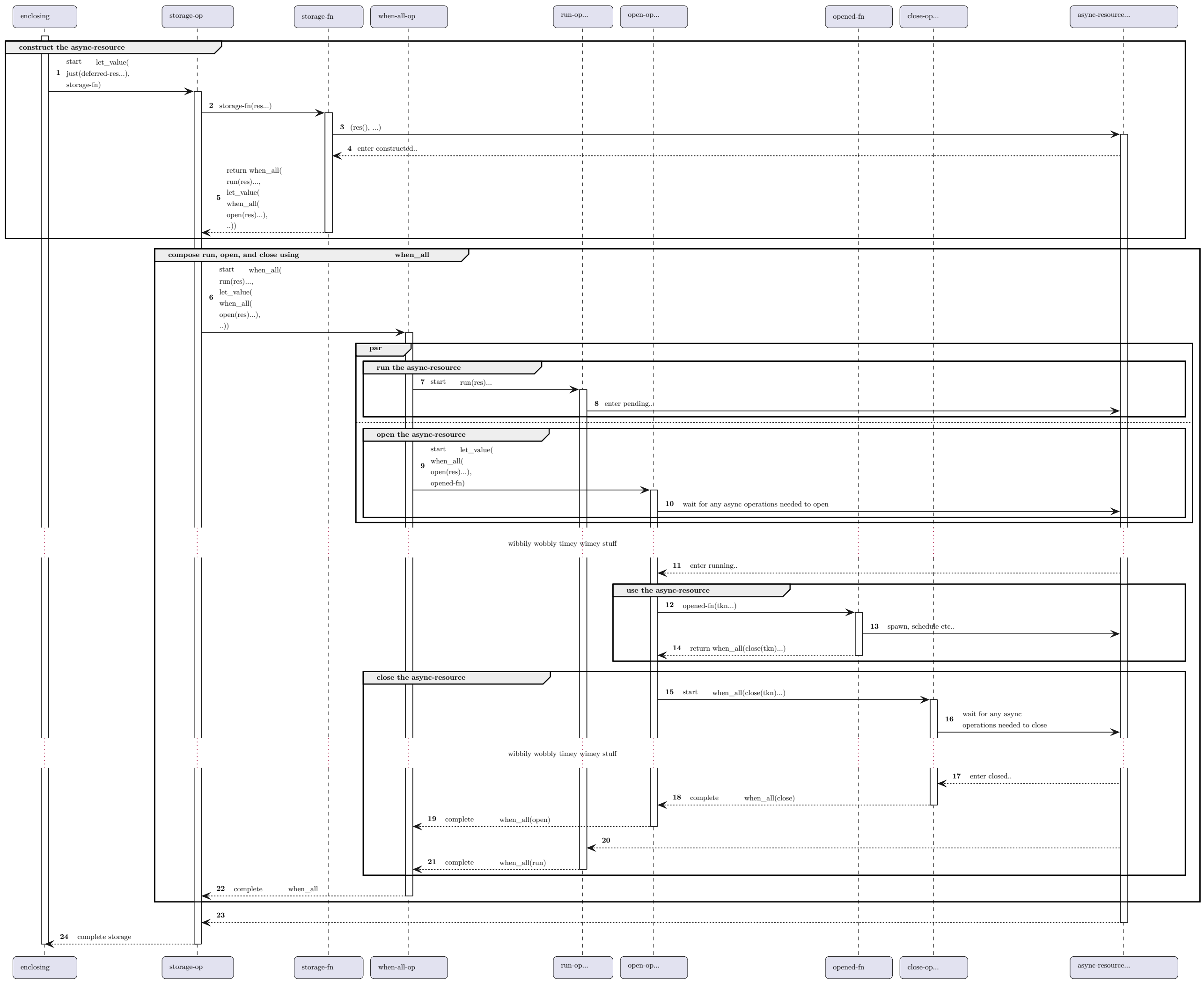
async-resource run() ->sequence-sender



sequence diagram for async-resource

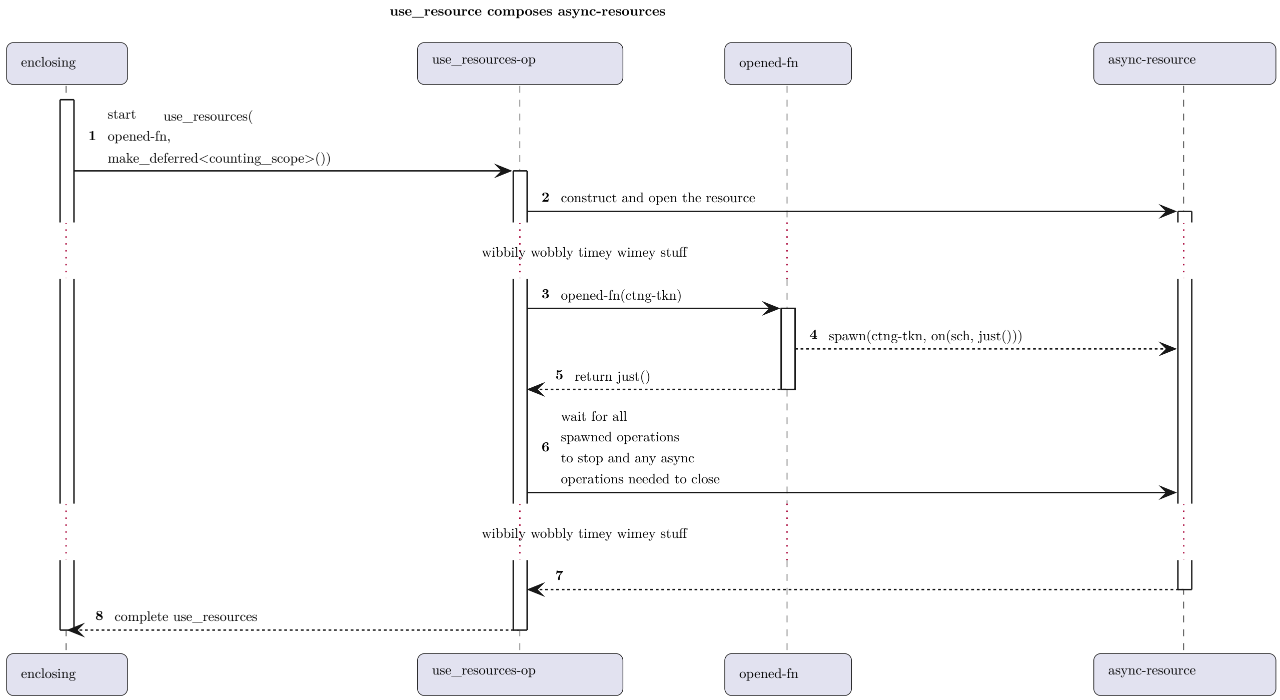
6.2.2 N resources

async-resource open/run/close



sequence diagram for async-resource

6.2.3 use_resource



sequence diagram for use_resource

7 References

[P2300R6] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. 2023-01-19. ‘std::execution‘. https://wg21.link/p2300r6