A Modest Executor Proposal

with homage to Swift

Document number: P1055

Date: 2018-04-26

Audience: Library Evolution Working Group

Reply-to: Kirk Shoop kirkshoop{at}fb[dotcom]; Eric Niebler eniebler{at}fb[dotcom]; Lee Howes lwh{at}fb[dotcom]

I. Table of Contents

II. Introduction

An abstraction to associate work with a target that will execute that work is an essential primitive that is required to write higher level libraries that deal with parallelism and concurrency.
There have been several attempts to define this facility with a focus on parallelism, but nothing has yet been accepted. There are indications that the current proposal for Executor is not a complete unification - one is that the NetworkingTS is including a separate `io_context` for concurrency and has devised a way to parameterize the async mechanism used on a per-function-call granularity, with the default being to use callbacks.

The intent of this paper is to express concerns about the complexity of the current executor and parallelism designs, and offer constructive suggestions for streamlining the current proposals and building on them into something more flexible. Following that, the paper will explore a subsumption hierarchy of Concepts and layered Implementations that allow separate implementations of the Concepts to maximize parallelism and/or provide efficient concurrency.

The approach taken in this paper is to list the goals for an Executor and describe a subsumption hierarchy of concepts that will have many implementations. Each implementation will address some goals. Once there are concepts and implementations of those concepts this paper will describe methods to compose the implementations.

Review of Iterator concepts

- An *Iterator* is intended to hide the mechanism of navigating a set of values so that algorithms and containers can be independent of each other. The *Iterator* concepts model a subsumption hierarchy of the operations on a pointer to accomplish this.

Foreshadowing of Executor concepts

- An *Executor* is intended to hide the mechanism of where, how and when some unit of work is performed. **where** may mean that the work is transmitted to an external processor and that the result is read back from the external processor. **how** may mean that the type of work is constrained in some way (SIMD, GPU). **when** may mean that the work is queued to a scheduler that controls when the work actually occurs. The concepts for executor will need to be a subsumption hierarchy of the additive features; error, done, value, sequence, cancellation, time-ordered.

Concerns:
There is a section in this paper called Addressing Concerns. Readers are encouraged to skip there at any point that they have concerns that they hope would be addressed.

P1053 - Future-proofing the Executor Continuation concept:
P1053 is an excellent choice to read right before this paper. In that paper a path is followed from callable to a fully generic continuation with composition.

Naming in this paper:
The purpose of names in this paper changes according to context. Sometimes names are used to connect with existing naming and sometimes names are used to distinguish from existing naming and then there are sometimes when the names are intended as proposals for names. It is expected that the names used in this paper will be ignored until naming is explicitly discussed. Appendices: Naming

III. Motivation and Scope

Starting with the **goals for an *Executor* Concept** is intended to bring the readers and authors of this paper to common ground at the start. There may be additional goals that were missed and

those should be added when the authors are made aware and subsequently addressed in the following sections.

1. Goals for an *Executor* Concept

1.1 Batchable

Batching is used to control the tradeoffs between the cost of each **transition** of a unit of work and the **granularity** of each unit of work.

1.2 Heterogenous

Heterogenous execution allows a unit of work compiled for a different instruction set to be transitioned, executed and the result transitioned back. Batching is important here because often the transition cost is high, so it is desirable to have fewer and larger units of work.

1.3 Orderable

The Order in which work units are invoked is often essential. In some common cases ordering information must be specified when submitting the work. It is also common to combine more than one ordering rule to determine the order of execution. The common composition of ordering rules for a particular scope in a concurrent algorithm library is: *`Sequential & Time & FIFO @ a particular Time T`*.

1.3.1 LIFO

The most recent submitted work is invoked first. Sometimes doing this will maximize locality.

1.3.2 FIFO

The oldest submitted work is invoked first. Some workloads require this guarantee because they deal with sequenced data.

1.3.3 Priority

Priority is important to support because some work (e.g. user initiated) is more important than other work.

1.3.4 Time

Time is essential to many forms of concurrency. Timeouts, polling-intervals and animations are some of the many uses of time-ordered execution.

1.3.5 Sequential

Each invocation of submitted work must be complete before the next is invoked.

1.4 Controllable

1.4.1 Targetable

The selection of an *Executor* allows work units that are submitted to it to be targeted to a specific execution resource or a specific set of resources.

1.4.2 Deferrable

Work is deferred until the Continuation is applied.

Eager constructs like `std::async` and `std::future` are designed to create a race between the result and the continuation and then have to add complexity and overhead to resolve the race. This creates non-deterministic context for calls to the continuation.

Deferred work is much less complex and much more efficient.

1.4.3 Cancelable

Allow the client to signal that the work should not be run and the result should not be delivered. Cancellation requires a back channel and complicates lifetime and therefore adds overhead. Back-channels like cancellation also require races to be resolved between the signals traveling in opposite directions.

1.5 Continuable

When an *Executor* supports non-blocking submission of work units, signals from the work units are needed.

1.5.1 Result Signal

The result of the work unit must be delivered once the work is complete.

1.5.2 Error Signal

When the invocation or transitions or ordering of the work fails, an error must be delivered instead of a result. if an unhandled exception from the work unit reaches the *Executor* it will be translated into an error signal.

1.5.3 Done Signal

Done is signaled when the result is not available and no error occurred. This is important when producing side effects.

1.5.4 Cancel Signal

Cancel is signaled when the consumer wishes to stop the producer or ignore the output from the producer. This is required when cancellation support is introduced (cancellation is not an error).

Cancel vs. Done:
Cancel and Done signals are only distinguished by the direction of travel. Done goes from producer to consumer. Cancel goes from consumer to producer.

1.5.5 Stopping Signal

Stopping is signaled when the handle passed to Starting is invalidated. This signal represents some of the lifetime complexity of a back-channel. Stopping is always the last signal when it exists.

1.5.6 Starting Signal

When cancellation is supported, Starting is signaled by the *`submit()`* method to pass a handle to the producer that can be used to cancel the producer (or ignore the producer result if the cancel signal loses the race with the result signal)

1.6 Layerable

1.6.1 Concepts

Subsumption hierarchies of concepts allow capabilities to be added without increasing the complexity of simpler things.

1.6.2 Types

Layered types and implementations allow an Ordering *Executor* to delegate the invocation of the work to another *Executor*

1.7 Usable

Ease-of-use applies to both implementors and users. This is less about the shape of the Concepts and more about the tools in the library provided to build and consume new types that model the Concepts. The complexity of building a model of the *Range* Concept is reduced by the tools in the range library.

1.7.1 Implementor

Implementors have tools that allow simple construction of new *Executor* types so that they focus on the behavior more than the shape.

1.7.2 Consumer

Consumers have tools that allow usage that is more varied and natural to each use case than the shape of the Concepts natively provides.

1.8 Composable

Composability allows implementations of the Concepts to compose to achieve usage patterns that have not been conceived of yet. Users will need to extend and add support for features that are not included in the standard.

Composability involves creating functions in the pattern *`Executor(Executor)`* and *`Continuation(Continuation)`* that adapt one implementation into another while composing some additional functionality.

Composability involves creating functions that compose sets of adaptors in the pattern of *`Executor operator|(Executor, Executor(Executor))`* and *`Executor pipe(Executor, Executor(Executor)...)`*.

1.9 Minimal

Nothing should exist on the *Executor* Concepts that could be added externally in a library on top of the Concept. This is similar to the principal that nothing should be added to the language that could be added to the library.

Even names should be selected to be most general and the types in the library named differently to be natural for each application of the Concepts.

2. Goals not met by the Executor proposal

> **Concerns:**
> There is a section in this paper called Addressing Concerns. Readers are encouraged to skip there at any point that they have concerns that they hope would be addressed.

> **Impact:**
> There is a section in this paper called IV. Impact On the Standard that will recommend specific changes for C++20.

2.1 Orderable

The only support for ordering in A Unified Executors Proposal for C++ and Executors Design Document are the *properties for bulk execution guarantees*, otherwise ordering is unspecified.

Even if there were more properties and *`_execute()`* methods for ordering semantics, it is unclear how one would compose them to ask for: *`Sequential & Time & FIFO @ a particular Time T`*.

Where work is targeted is a separate concern from the order in which it is processed on that target.

It is tempting to split these separate concerns into separate concepts. This is what the NetworkingTS *`io_context`* does. The *`io_context`* owns a queue and the work in that queue can be consumed by one or more *Executors* consuming work from that queue by calling

run(). However, the *io_context* is not minimal and not sufficient, it is unclear how one would ask for: *Sequential & Time & FIFO @ a particular Time T*.

2.2 Layerable

The requires mechanism provides layering of implementations. The requires mechanism implicitly creates Concepts for every permutation of the *execution* methods, with an implementation returned by a call to *requires()* supporting one permutation of the Concepts.

Working from the type-erasure definition here: A Unified Executors Proposal for C++ - 1.5.1 the uber-concept might look like this:

```
struct Executor {
  template <class Property>
  executor require(Property) const;

  template <class Property>
  executor query(Property) const;

  template<class Function>
  void execute(Function&& f) const;

  template<class Function>
  std::experimental::future<result_of_t<decay_t<Function>()>>
  twoway_execute(Function&& f) const

  template<class Function, class Future>
  std::experimental::future<result_of_t<decay_t<Function>()>>
  then_execute(Function&& f, Future&& fut) const;

  template<class Function, class SharedFactory>
  void bulk_execute(Function&& f, size_t n, SharedFactory&& sf) const;

  template<class Function, class Shape, class Future, class RF, class SF>
  std::experimental::future<result_of_t<decay_t<Function>()>>
  bulk_then_execute(Function&& f, Shape s, Future&& fut, RF&& rf, SF&& sf) const;

  template<class Function, class ResultFactory, class SharedFactory>
  std::experimental::future<result_of_t<decay_t<ResultFactory>()>>
  bulk_twoway_execute(Function&& f, size_t n, ResultFactory&& rf, SharedFactory&& sf) const;
};
```

Due to the nature of the require mechanism, there are additional concepts implied, though none are defined, for all the permutations of *execute* supported by the type.

The type-erasure definition also demonstrates that the proposal is user extensible on the set of Properties, but has no way for the user to extend the set of *_execute* methods for type-erasure. This means that adding full support for ordering by time would be precluded until it was added to a future standard.

> **require extension mechanism:**
> The require/prefer/query extension mechanism is a novel method for composition. It would be great to see it as a library for composing arbitrary types which could be applied to executors for those that want that style of composition.

The existing permutations of oneway, twoway, then, single, bulk and expected additional permutations for time & priority are concerning.

2.3 Composable

The mechanism of using requires to add/remove *`execute`* methods from an *Executor* prevents user-added functionality from composing the same way as the built-in functionality.

The mechanism of require, followed by *`.execute()`* invocation requires a lot of hand wiring and extra verbosity to compose operations together.

The *`execute`* overloads already look a lot like a future, yet some also return a future. This also complicates composition of execution work and promise work.

2.4 Controllable

Deferral is supported but cancel is not, nor is there a plan for how cancel would be added to ensure that it is a strict super-set of the current proposal.

> **Future cancellation:**
> The P1055 paper makes an attempt at defining cancellation for future that is suitable for use by executors, but through a very specific mechanism rather than a general mechanism. This is another consequence of defining executor and future separately as types and providing neither with concepts.

2.5 Minimal

The inclusion of the requires mechanism and the set of *`execute`* functions is not minimal and yet is equally insufficient to support common usage (e.g. *`Sequential & Time & FIFO @ a particular Time T`*).

The overlap of the underlying pattern of *function that takes a continuation* with a future that supports *`.then()`* indicates that the types being added to the library for executors and futures are obscuring the underlying Concepts that they share.

3. Satisfying these goals

One way to satisfy additional goals is to add complexity to what exists. Another way to satisfy additional goals is to find a core set of concepts upon which all the functionality can be built. The following takes the later approach.

> **Concerns:**
> There is a section in this paper called Addressing Concerns. Readers are encouraged to skip there at any point that they have concerns that they hope would be addressed.

3.1 Before and After

3.1.1 *std::async()* example

This example of *std::async()* from the Executors Design Document - 5.1

```
template<class Executor, class Future, class... Args>
execution::executor_future_t<Executor, auto>
async(const Executor& exec, Function&& f, Args&&... args) {
  // bind together f with its arguments
  auto g = bind(forward<Function>(f), forward<Args>(args)...);

  // introduce single-agent, two-way execution requirements
  auto new_exec = execution::require(exec, execution::single, execution::twoway);

  // implement with execution function twoway_execute
  return new_exec.twoway_execute(g);
}
```

The changes below involve switching from placing execution algorithms as members on the executor, exposed through the requires mechanism, to making external algorithms that compose with Executor concepts.

```
template<class Executor, class Future, class... Args>
auto
async(const Executor& exec, Function&& f, Args&&... args) {
  // bind together f with its arguments
  auto g = bind(forward<Function>(f), forward<Args>(args)...);

  // implement with execution twoway_execute
  return exec | twoway_execute(g) | eager_incorrigible_race_with_continuation();
}
```

> *eager_incorrigible_race_with_continuation()* follows the lineage of making dangerous or slow things look ugly. This one in particular makes something that would be dangerous (a race between the producer of a value and the consumer setting a continuation to receive the value) become safe by adding overhead and making it slow (heap allocation and lock-free code to resolve the race).

3.1.2 *inline_executor* example

This example of *inline_executor* from the Executors Design Document - 6.

```
struct inline_executor {
  . . .
  const inline_executor& context() const noexcept {
    return *this;
  }
  inline_executor require(execution::always_blocking) const noexcept
  {
    return *this;
  }
  template<class Function>
```

```
  void execute(Function&& f) const noexcept {
    std::forward<Function>(f)();
  }
};
```

The changes below reveal the beginnings of the concepts for an Executor. Rather than requiring certain behavior of an executor, composition would be used to layer the desired behavior across executors. More details to follow and don't get very attached to them as shown here..

```
struct inline_executor {
  . . .
  template<class Continuation>
  void execute(Continuation&& c) const noexcept {
    try {
      std::forward<Continuation>(c).work(*this);
    } catch(...) {
      std::forward<Continuation>(c).error(std::current_exception());
    }
  }
};
```

3.1.3 *executor* type-erasure example

This definition of the type-erased executor from A Unified Executors Proposal for C++

```
template <class... SupportableProperties>
class executor
{
public:
  . . .
  // executor operations:

  template <class Property>
  executor require(Property) const;

  template <class Property>
  executor query(Property) const;

  template<class Function>
  void execute(Function&& f) const;

  template<class Function>
  std::experimental::future<result_of_t<decay_t<Function>()>>
  twoway_execute(Function&& f) const

  template<class Function, class Future>
  std::experimental::future<result_of_t<decay_t<Function>()>>
  then_execute(Function&& f, Future&& fut) const;

  template<class Function, class SharedFactory>
  void bulk_execute(Function&& f, size_t n, SharedFactory&& sf) const;

  template<class Function, class Shape, class Future, class RF, class SF>
  std::experimental::future<result_of_t<decay_t<Function>()>>
  bulk_then_execute(Function&& f, Shape s, Future&& fut, RF&& rf, SF&& sf) const;

  template<class Function, class ResultFactory, class SharedFactory>
  std::experimental::future<result_of_t<decay_t<ResultFactory>()>>
```

```
  bulk_twoway_execute(Function&& f, size_t n, ResultFactory&& rf, SharedFactory&& sf) const;

  . . .
};
```

The changes below reveal more about the concepts for an Executor. Remember, don't get very attached to them..

```
template<class E>
class executor;

template<class E>
class continuation
{
public:
  . . .
  // continuation operations:

  void work(executor<E>& exec);
  void error(E e) noexcept;

  . . .
};

template<class E>
class executor
{
public:
  . . .
  // executor operations:

  void execute(continuation<E> c);

  . . .
};
```

3.2 A Future that is an Executor

> **P1053 - Future-proofing the Executor Continuation concept:**
> P1053 is an excellent choice to read right before this section. In that paper a path is followed from callable to a fully generic continuation with composition.

Here is a Future that supports Continuation and defers the start of the work until the Continuation is attached.

```
struct Promise
{
  template<class T>
  void value(T&& t);

  template<class E>
  void error(E&& e);
};

struct Future
{
```

```
  template<class Promise>
  void submit(Promise);
};
```

Static Polymorphism:
Leaving *T*, *E* and *Promise* arguments to the functions unspecified, until they are called, allows very interesting compositions. One is that overloads are supported for each named function. With libraries that allow a *Callable* type to be built from a set of lambdas (these already exist to support other forms of visitation), it is not complicated to create these overload sets. Another is that *exception_ptr* is not required to be the only type used to represent errors as a value. *std::error_code* will work and any other error value type will work. with an overload set for *error()*, many error types can be supported by the same *Promise* implementation.

Deferred vs. Eager:
Deferred work is efficient and simple, Eager work is inefficient and complex. A Deferred *Future* looks like *Future{[](auto p){p.value(42);}}.submit(Promise{[](auto v){}});* godbolt. By deferring the work until *submit()* is called the *Promise* passed to submit can be directly passed to the work lambda which will directly call *value()* with *42*. this completely eliminates the race between the value and the continuation.

The only difference in the *Future* above and the *Executor* shape described earlier is the naming and additional flexibility on the types of the function parameters.

Given this definition of a Future, it is interesting to ask whether the *Executor* Concept needs a separate definition. One of the algorithms that would compose an *Executor* with a *Future* would be *Future via(Executor, Future)* here is what that might look like if the *Future* just defined was used to implement the *Executor*.

```
auto FooExecutor = makeFuture([](auto out){ // submit
  std::thread t{[out](){
    // an executor always passes an executor as the value.
    // usually the executor passes itself so that
    // more work can be scheduled on the same executor
    //
    // this executor ignores any nested work.
    out.value(makeFuture());
  }};
  t.detach();
});

// via returns a new future that will
// schedule all calls from 'in' to
// value and error, via 'exec'.
template<class FutureExec, class FutureIn>
auto via(FutureExec exec, FutureIn in) {
  return makeFuture([exec, in](auto out){ // submit
    // create a promise that will submit each
    // call to 'exec' and then pass the parameter
    // to 'out'.
    // makePromise captures 'out' and passes it
    // to each lambda as the first parameter
    auto viap = makePromise(out,
```

```
    [exec](auto out, auto v){ // value
      // capture 'v' and 'out' in a Promise
      // that will forward 'v' to out.value
      auto execvp = makePromise(out,
      [v](auto out, auto exec){ // value
        out.value(v);
      });
      // submit 'execvp' to be called by the executor
      exec.submit(execvp);
    },
    [exec](auto out, auto e){ // error
      // capture 'e' and 'out' in a Promise
      // that will forward 'e' to out.error
      auto execep = makePromise(out,
      [e](auto out, auto exec){ // value
        out.error(e);
      }));
      // submit 'execep' to be called by the executor
      exec.submit(execep)
    });
    // time to start 'in' and pass the promise
    in.submit(viap);
  });
}

// submit will return immediately and the lambda passed
// to it will be called later from the std::thread
via(FooExecutor{}, makeFuture([](auto out){ //submit
  out.value(42);
})).submit(makePromise([](auto v){}));
```

Static Functions:
It is important to call out that every call to value, error and submit are straight, static function calls, only the FooExecutor{}.submit() would add overhead to start and run the work on the thread.

Thread Ownership:
The *`detach()`* is correct here. lifetime ownership is transferred to the function passed to the thread. the promise passed to submit will be called to signal that the thread is ending and this signal can be used by the implementor of *`value()`* & *`error()`* to wait for work to complete before continuing or exiting an arbitrary C++ scope.

3.3 Batching

Batching is about controlling the balance between the granularity of the work and the cost of the transitions that submit the work and deliver the result of the work.

The granularity is controlled by how the work is composed and this does not need to be expressed in the shape of the *Executor* Concepts. Work composition can be done by smashing continuations together using functions like *`continuation(continuation)`* or by smashing executors together using functions like *`executor(executor)`*. with overloads for specific types, like *`bulk_executor(bulk_executor)`* the function can leverage private functions to move internals from the parameter and construct a new *`bulk_executor`* that composes two

operations or can use private functions to add the new operation to the parameter and then move that same instance to the result.

> in the following code, presume that *bulk_future* defers the transition of the work until a continuation is attached. this allows multiple *bulk_future* to be combined and sent together.

```
template<class Target, class Function, class SharedFactory>
bulk_future bulk_execute(Target&& t, Function&& f, size_t n, SharedFactory&& sf) {
  auto work = makeBulkWork(t, f, n, sf);
  return makeBulkFuture(work);
}

bulk_future smash(bulk_future lhs, bulk_future rhs) {
  // combine the work from both lhs and rhs into one bulk_future
  return makeBulkFuture(lhs.work(), rhs.work());
}
```

The transition cost is a property of an *Executor* implementation and does not need to be expressed in the shape of the *Executor* Concept.

3.4 Heterogenous

The work for heterogenous computation is often constrained in the operations that it can perform. what this means is that all of the functions on the Concepts run on the CPU.

The structure of heterogenous *Executors* follows a pattern.

- CPU submits compatible work to device
- device does work
- CPU delivers result from device via continuation

The implementation decides whether the CPU blocks waiting for the work or blocks doing the work or returns after submitting the work and continues based off some internal signal.

The function *simd_future simd_execute(. . .)* fits this pattern and these *simd_future*s can be composed before the final continuation is attached and the deferred work is started.

> **Batching:**
> The number of operations performed on the device per submission is determined by how the work is composed prior to submission. It is expected that operations will steal the work from adjacent operations and submit them all at once. See 3.3 Batching

3.5 Ordering

Ordering is in play when there are queues of non-blocking work. In these cases *submit()* pushes into the queue and potentially returns before the work is even started.

Sequential, LIFO & FIFO do not surface on the Concepts.

Priority could appear on the Concepts, or since it is a finite construct, it could be abstracted by a Factory.

```
auto priority = makePriorityFactory(getSystemPool());
auto highexec = priority.makeExecutor(HIGH_PRIORITY);
auto normalexec = priority.makeExecutor(NORMAL_PRIORITY);

highexec | execute(work1) | then([normalexec](auto& highexec){
  return normalexec | twoway_execute(work2);
}) | then([](auto work2result){});
```

The sticky ordering primitive is Time. Time is a construct used to describe the physical world. As such it requires explicit support in the concepts. This is one of those cases where the library will be able to hide an aspect of the Concepts for the many uses that do not care to specify ordering in time.

```
pool = getSystemPool();

pool.execute(pool.now(), work1); // ewww

pool | execute(work1); // same thing as the `.execute()` - no muss no fuss
pool | execute_at(pool.now() + 1d, work1); // still nice
pool | execute_after(1d, work1); // same as the call to execute_at
```

Adding time affects the *Future*, but not the *Promise*.

```
template<class TimePoint>
struct TimeFuture
{
  TimePoint now();

  template<class Promise>
  void submit(TimePoint, Promise);
};
```

3.6 Minimal

The *Future* in 3.2 is minimal but not sufficient for Time ordering. The TimeFuture is sufficient and minimal until cancellation and support for value-sequences and back-pressure are supported. Concepts that support these additions are all defined in Appendices: Naming and Appendices: Time. These are structured as a subsumption hierarchy similar to *Iterators*. The subsumption hierarchy is made explicit in the Appendices: Concepts

4. Addressing Concerns

- **Does this require starting over on executors?**
  No! most of the work to define the behavior and functionality of the types in the Executor proposal are required after concepts are applied. The concepts will change composition and layering and user extensibility.

- **Does every goal have to be achieved right now?**
  No! as long as there is a clear plan from the proposal now to the full set of concepts and library tools needed to achieve the goals later.

- **Why object now?**
  Other priorities have taken precedence until now. Much that is in this paper has been expressed in conversations with committee members before, but until now has not had a formal write up.

- **What about the additional composition overhead?**
  Based on previous experience with similar libraries, the runtime overhead of these compositions will be zero. The actual implementation will affect this statement as will type-erasure, but type-erasure will be explicitly controlled by the user and the implementation will be under the control of the components being composed. P1053 has some great links to *godbolt* that demonstrate how the compositions are optimized out.

- **Why not have fewer concepts with more capabilities?** Everything can be done with Many, FlowMany & TimeFlowManyDeferred. cancellation and time are both things that have to be exposed in the concepts and the choice is between many/fewer concepts and compile-time/runtime detection (though static tools could also be built to enforce these runtime restrictions). This balance can be shifted but those shifts have large impact on the libraries. Taking smaller steps from the minimal concepts up to the full featured concepts allows incremental and purely additive changes to the libraries and the capabilities.

IV. Impact On the Standard

> **Rigor**
> This section will have too much hand waving. There is code similar to this on github now. There will more exact implementations of this on github later.

1. concepts for C++20

One minimum change needed for C++20 would be to move the expression of executor into a subsumption hierarchy of concepts. The require mechanism and usage could remain, but not be the fundamental expression for purposes of composition and usage.

Algorithms should take Concepts as parameters and not call requires()/prefers().

bulk, twoway and then execute overloads should be made into operators that can be composed.

The minimum concepts to support oneway, twoway, bulk and then_execute are the following (they are also sufficient to support `std::future` - where work starts at construction, and *task* - where work starts at submission).

```
struct None
{
  template<class E>
  void error(E&& e) noexcept;
```

```
  void done();
};

struct Deferred
{
  template<class None>
  void submit(None);
};

struct Single
{
  template<class T>
  void value(T&& t);

  template<class E>
  void error(E&& e) noexcept;

  void done();
};

struct SingleDeferred
{
  template<class Single>
  void submit(Single);
};
```

> For conversation and context for these names, please consult Appendices: Naming

2. expression composition for C++20

Also minimum for C++20 is expression composition. This involves the definition of *Adaptor* to chain from producer to consumer and *Lifter* to chain from consumer to producer.

```
SingleDeferred Adapter(SingleDeferred);

Single Lifter(Single);
```

Adaptor and *Lifter* functions are often produced by factory functions that capture some parameters to define the transformation that will be applied by the *Adapter* or *Lifter* returned.

```
template<class... AN>
auto Foo(AN&&... an) {
  //
  // An adapting operator like Foo returns a function
  // that takes a SingleDeferred and returns a SingleDeferred.
  //
  return [<capture an>](auto singleDeferred){
    return makeFooSingleDeferred(<captured an>, singleDeferred);
  };
}

template<class... AN>
auto Bar(AN&&... an) {
  //
  // A lifting operator like Bar returns a function
  // that takes a Single and returns a Single.
```

```
  // this Lifter function can be composed using
  //
  return [<capture an>](auto single){
    return makeBarSingle(<captured an>, single);
  };
}
```

An *Adaptor* function can be composed using
SingleDeferred operator|(SingleDeferred, Adaptor);
or a function like
template<class... AdapterN> SingleDeferred pipe(SingleDeferred, AdaptorN... adapterN);
With usage like this

```
http.get('http://localhost') |
  adapt::filter([](auto r){ return r.status != 200;}) |
  adapt::map([](auto r){ throw http_exception{r}; }) |
  adapt::submit();
// or
adapt::pipe(http.get('http://localhost'),
  adapt::filter([](auto r){ return r.status != 200;}),
  adapt::map([](auto r){ throw http_exception{r}; })).
  submit();
```

A *Lifter* function can be composed using
Single operator|(Single, Lifter);
or a function like
template<class... LifterN> Single pipe(Single, LifterN... lifterN);
With usage like this

```
http.get('http://localhost') |
  adapt::submit(
    lift::filter([](auto r){ return r.status != 200;}) |
    lift::map([](auto r){ throw http_exception{r}; }));
// or
http.get('http://localhost').
  submit(lift::pipe(
    lift::filter([](auto r){ return r.status != 200;}),
    lift::map([](auto r){ throw http_exception{r}; })));
```

It is possible to implement *pipe()* tersely in terms of *operator|* using C++17 fold expressions

```
template <class In, class Operator>
auto operator|(In&& in, Operator op) -> decltype(op(in)) {
  return op(in);
}

template<class T, class... FN>
auto pipe(T t, FN... fn) {
    return (t | ... | fn);
}
```

> It is trivial to turn *Lifters* into *Adaptors*. It is quite challenging and would probably introduce overhead, to turn *Adaptors* into *Lifters*.

3. library support for C++20

3.1 execute and twoway_execute

oneway and twoway execution are supported by *`std::execute(F)`*

```
template<class F>
auto execute(F&& f) {
  return [f](auto exec) {
    if constexpr (std::is_void_v<std::result_of_t<std::decay_t<F>>>) {
      // f returns void, return deferred{}
      return std::deferred{[f](auto out){
        exec.submit(single{out, [f](auto out, auto exec){
            f();
            out.done();
          }});
        }};
    } else {
      // f returns value, return single_deferred{}
      return std::single_deferred{[f](auto out){
        exec.submit(single{out, [f](auto out, auto exec){
          auto r = f();
          out.value(r);
        }});
      }};
    }
  };
}

auto f = std::new_thread() | std::execute([](){ return 42; });
// run f twice - each on a new thread
auto i = std::this_thread::get(f);
auto ii = std::this_thread::get(f);

auto v = std::new_thread() | std::execute([](){ /* side-effects */ });
// run v twice - each on a new thread
std::this_thread::get(v);
std::this_thread::get(v);
```

3.2 then_execute

This construct can exist and usage prioritizes the composition of executors.

```
template<class F, class In>
auto then_execute(F&& f, In&& in) {
  return std::executor{[f, in](auto exec){
    return in | std::via(exec) | std::map(f);
  }};
}

auto exp84 = std::new_thread() |
  std::then_execute([](int i){ return i * 2; }, std::just(42));
// run exp84 twice - each on a new thread
```

```
auto i = std::this_thread::get(exp84);
auto ii = std::this_thread::get(exp84);
```

This expresses the same operations as above and usage prioritizes the composition of data.

```
auto exp84 = std::just(42) |
  std::via(std::new_thread()) | std::map([](int i){ return i * 2; });
// run exp84 twice - each on a new thread
auto i = std::this_thread::get(exp84);
auto ii = std::this_thread::get(exp84);
```

3.3 bulk_execute

This example demonstrates how chained work can be smashed together into one transition

```
template<class Function, class SharedFactory>
auto bulk_execute(Function&& f, size_t n, SharedFactory&& sf) {
  return [f, n, sf](auto exec) {
    if constexpr (is_bulk_executor_v<decltype(exec)>) {
      // private function to accrete work to be batched together
      return exec.add(f, n, sf);
    } else {
      return bulk_executor{exec, f, n, sf};
    }
  };
}
```

This example shows a different structure for the bulk algorithm that is data focused and using parameters that are more general (there is only one piece of state, that can contain all the implementation details, and a selector function to resolve the final result value from the internal state)

```
template<class F, class Shape, class IF, class RS>
auto bulk(
  F&& func,
  Shape s,
  IF&& initFunc,
  RS&& selector) {
    return [func, s, initFunc, selector](auto in){
      return std::single_deferred{[in, func, s, initFunc, selector](auto out) mutable {
        in.submit(std::single{out,
          [func, s, initFunc, selector](auto out, auto input){
            auto [target, acc] = initFunc(input);
            std::bulk_driver(target, acc, func, s);
            auto result = selector(std::move(acc));
            out.value(std::move(result));
          }
        });
      }};
    };
  }
```

See the compiler output on godbolt

3.4 just

```
template<class V>
auto just(V&& v) {
  return std::single_deferred{[v](auto out){
    try {
      out.value(v);
    } catch (...) {
      out.error(std::current_exception());
    }
  }};
}

auto i = std::this_thread::get(std::just(42));
```

3.5 map

```
template<class F>
auto map(F&& f) {
  return [f](auto in) {
    if constexpr (std::is_single_deferred_v<decltype(in)>) {
      // 'in' is a singleDeferred, create a single{} and submit that to 'in'
      if constexpr (std::is_void_v<std::result_of_t<std::decay_t<F>>>) {
        // f returns void, return deferred{}
        return std::deferred{[f, in](auto out){
          in.submit(single{out,
            [f](auto out, auto v){
              f(v);
              out.done();
            }});
          }
        };
      } else {
        // f returns value, return single_deferred{}
        return std::single_deferred{[f, in](auto out){
          in.submit(single{out,
            [f](auto out, auto v){ out.value(f(v)); }});
          }
        };
      }
    } else {
      // 'in' is a deferred, create a none{} and submit that to 'in'
      if constexpr (std::is_void_v<std::result_of_t<std::decay_t<F>>>) {
        // f returns void, return deferred{}
        return std::deferred{[f, in](auto out){
          in.submit(none{out,
            [f](auto out){
              f();
              out.done();
            }});
        }};
      } else {
        // f returns value, return single_deferred{}
        return std::single_deferred{[f, in](auto out){
          in.submit(none{out,
            [f](auto out){ out.value(f()); }})
        }};
      }
    }
  };
}
```

```
auto i = std::this_thread::get(std::just(42) | std::map([](int i){ return i * 2; }));
```

3.6 concat

```
template<class Executor>
auto concat(){
  return [](auto in){
    return std::single_deferred{[in](auto out){
      in.submit(single{out, [](auto out, auto v){
        v.submit(out);
      }});
    }};
  };
}

// concat allows nested operations to be resolved
std::just(21) |
  std::map([](int i){return std::just(i * 2);}) |
  std::concat() |
  std::submit();
```

3.7 via

```
template<class Executor>
auto via(Executor exec) {
  // return SingleDeferred(SingleDeferred) function for composition
  return [exec](auto singleDeferred) {
  // create the single_deferred{} that will proxy calls from
  // 'singleDeferred' through the 'exec'
  return std::single_deferred{[exec, singleDeferred](auto out) {
    // create a single that will submit each
    // call to 'exec' and then pass the parameter
    // to 'out'.
    // single{} captures 'out' and passes it
    // to each lambda as the first parameter
    auto viap = single{out,
      [exec](auto out, auto v){ // value
        // capture 'v' and 'out' in a single
        // that will forward 'v' to out.value
        auto execvp = single{out,
          [v](auto out, auto exec){ // value
            out.value(v);
          }
        };
        // submit 'execvp' to be called by the executor
        exec.submit(execvp);
      },
      [exec](auto out, auto e){ // error
        // capture 'e' and 'out' in a single
        // that will forward 'e' to out.error
        auto execep = single{out,
          [e](auto out, auto exec){ // value
            out.error(e);
          }
        };
        // submit 'execep' to be called by the executor
        exec.submit(execep)
      }
```

```
    };
    // time to start 'singleDeferred' and pass the single
    singleDeferred.submit(viap);
    }};
  };
}
```

3.8 on

```
template<class Executor>
auto on(Executor exec) {
  // return SingleDeferred(SingleDeferred) function for composition
  return [exec](auto singleDeferred) {
    // create the single_deferred{} that will proxy calls to
    // 'singleDeferred' through the 'exec'
    return std::single_deferred{[exec, singleDeferred](auto out) {
      // capture 'v' and 'out' in a single
      // that will forward 'v' to out.value
      auto execs = single{out,
        [singleDeferred](auto out, auto exec){ // value
          // time to start 'singleDeferred' and pass the single
          singleDeferred.submit(out);
        }
      };
      // submit 'execs' to be called by the executor
      exec.submit(execs);
    }};
  };
}
```

3.9 new_thread

```
auto new_thread() {
  return std::single_deferred{[](auto s){
    std::thread t{[s](){
      // TODO:
      // add a trampoline/derecursor for nested
      // work on the same thread and pass that
      // to s.value();
      s.value(std::new_thread());
    }};
    // pass ownership of thread to s
    t.detach();
  }};
}
```

3.10 get

```
template<class T, class SingleDeferred>
T get(SingleDeferred in) {
  std::mutex lock;
  std::condition_variable signal;
  std::variant<std::exception_ptr, T> result;
  in.submit(single{
    [&](T t){result.emplace(t);},
    [&](auto e){result.emplace(std::make_exception_ptr(e));}
```

```
  });
  std::unique_guard<std::mutex> guard{lock};
  signal.wait(guard, [&](){
    auto pep = std::get_if<std::exception_ptr>(&result);
    return !pep || !*pep;
  });
  if (auto pep = std::get_if<std::exception_ptr>(&result)) {
    std::rethrow_exception(*pep);
  }
  return std::get<T>(result);
}
```

3.11 executor

```
template<class SubmitFunction>
class executor : private SubmitFunction;
```

> this function inheritance pattern learned from *Zero-allocation & no type erasure futures - Vittorio Romeo @ ACCU 2018* youtube

3.12 single_deferred

```
template<class SubmitFunction>
class single_deferred : private SubmitFunction;
```

3.13 single

```
template<class... TN>
class single;

template<class T>
class single<std::promise<T>>;

template<class ValueFunction, class ErrorFunction, class DoneFunction>
class single<ValueFunction, ErrorFunction, DoneFunction>;

template<class Data, class DataValueFunction, class DataErrorFunction, class DataDoneFunction>
class single<Data, DataValueFunction, DataErrorFunction, DataDoneFunction>;
```

3.14 none

```
template<class... TN>
class none;

template<>
class none<std::promise<void>>;

template<class ErrorFunction, class DoneFunction>
class none<ErrorFunction, DoneFunction>;
```

```
template<class Data, class DataErrorFunction, class DataDoneFunction>
class none<Data, DataErrorFunction, DataDoneFunction>;
```

3.15 deferred

```
template<class SubmitFunction>
class deferred : private SubmitFunction;
```

3.16 *std::promise<T>* & *std::future<T>*

```
template<class T>
std::future<T> future_from(auto singleDeferred) {
  std::promise<T> p;
  auto result = p.get_future();
  singleDeferred.submit(single{p});
  return result;
}

std::future<void> future_from(auto deferred) {
  std::promise<void> p;
  auto result = p.get_future();
  deferred.submit(none{p});
  return result;
}
```

3.17 task/outcome

type-forgetters for implementations of the concepts

```
template<class E = std::exception_ptr>
class void_outcome {
  void error(E e) noexcept override;
  void done() noexcept override;
};

template<class E = std::exception_ptr>
class void_task {
  void submit(void_outcome<E> out) override;
};

template<class T, class E = std::exception_ptr>
class single_outcome : public void_outcome<E> {
  void value(T t) override;
};

template<class T, class E = std::exception_ptr>
class single_task {
  void submit(single_outcome<T, E> out) override;
};
```

4. concepts for time and cancellation

Pie in the sky! Add two additional concepts to C++20 to support time-ordering and cancellation

```
struct FlowSingle
{
  template<class T>
  void value(T&& t);

  template<class E>
  void error(E&& e) noexcept;

  void done();

  void stopping() noexcept;
      // up reference is invalidated

  template<class NoneRef>
  void starting(NoneRef up);
      // up.error(e); - aborts with consumer error
      // up.done(); - cancels
};

template<class TimePoint = std::chrono::system_clock::time_point>
struct TimeFlowSingleDeferred
{
  TimePoint now() noexcept;

  template<class FlowSingle>
  void submit(TimePoint, FlowSingle);
};
```

V. Acknowledgments

Many people have contributed to the long history of iterative design and application of the ideas in this paper. Some of those are: Lee Howes, Eric Niebler, David Sankel, Sean Parent, Gor Nishanov, Erik Meijer, Ben Christensen, Aaron Lahman, Mark Lawrence, Marc Barbour

VI. Appendices

Naming

Picking names that will survive later additions to these Concepts would be worthwhile. Naming the Concepts differently from the types may also help avoid confusion.

Additional Concepts would support cancellation, time, value sequences, back-pressure, etc..

The following is one approach to naming that is consistent with these additional features

Deferred

The shape of a *Future* that does not produce a result value. (ala *`std::future<void>`*)

```
struct None
{
  template<class E>
  void error(E&& e) noexcept;

  void done();
};

struct Deferred
{
  template<class None>
  void submit(None);
};
```

SingleDeferred

The shape of a *Future* and an *Executor*

```
struct Single
{
  template<class T>
  void value(T&& t);

  template<class E>
  void error(E&& e) noexcept;

  void done();
};

struct SingleDeferred
{
  template<class Single>
  void submit(Single);
};
```

FlowSingleDeferred

The shape of a *Future* and an *Executor* with cancellation

```
struct FlowSingle
{
  template<class T>
  void value(T&& t);

  template<class E>
  void error(E&& e) noexcept;

  void done();

  void stopping() noexcept;
      // up reference is invalidated

  template<class NoneRef>
  void starting(NoneRef up);
      // up.error(e); - aborts with consumer error
      // up.done(); - cancels
};
```

```
struct FlowSingleDeferred
{
  template<class FlowSingle>
  void submit(FlowSingle);
};
```

ManyDeferred

The shape of a set of events w/o cancellation

```
struct Many
{
  template<class T>
  void value(T&& t);

  template<class E>
  void error(E&& e) noexcept;

  void done() noexcept;
};

struct ManyDeferred
{
  template<class Many>
  void submit(Many);
};
```

FlowManyDeferred

The shape of a set of events with cancellation and back-pressure

```
struct FlowMany
{
  template<class T>
  void value(T&& t);

  template<class E>
  void error(E&& e);

  void done() noexcept;

  void stopping() noexcept;
      // up reference is invalidated

  template<class ManyRef>
  void starting(ManyRef up);
      // up.value(rate/count); - controls back-pressure
      // up.error(e); - aborts with consumer error
      // up.done(); - cancels
};

struct FlowManyDeferred
{
  template<class FlowMany>
  void submit(FlowMany);
};
```

Other naming patterns that have been discussed are:

- *None*Result/*Single*Result/*Many*Result
- *None*Result/*One*Result/*Some*Result
- *Void*Result/*Value*Result/*Varied*Result

With some substitutions:

- Result -> Continuation
- Result -> Value
- Result -> Promise
- None -> Empty
- None -> Void
- Single -> Value
- Single -> One
- Many -> Multi
- Many -> Some
- Many -> Various
- Many -> Varied
- Deferred -> Future
- Deferred -> Task

> Future & Promise already have a lot of baggage and actual types in the std. These concepts can be used to implement those types but are not bound to the same baggage that the types have - a distinction in naming for the concepts could be helpful.

Time

Time is a construct used to describe the physical world. As such it requires explicit support in the concepts.

Actually all the previous Deferred concepts can be completely replaced with the following, since they are strict super-sets and the library will provide submit helpers that pass *now()* for the TimePoint and *max()* for the Duration.

TimeDeferred

The shape of a *Future*, that does not produce a result value, with time-ordering

```
template<class TimePoint = std::chrono::system_clock::time_point>
struct TimeDeferred
{
  TimePoint now() noexcept;

  template<class None>
  void submit(TimePoint, None);
};
```

TimeSingleDeferred

The shape of a *Future* and an *Executor* with time-ordering

The TimePoint specifies that the *value()* method is expected to be called as soon as possible after that time.
The *now()* method allows a virtual clock owned by the implementation to control time.

```
template<class TimePoint = std::chrono::system_clock::time_point>
struct TimeSingleDeferred
{
  TimePoint now() noexcept;

  template<class Single>
  void submit(TimePoint, Single);
};
```

TimeFlowSingleDeferred

The shape of a *Future* and an *Executor* with time-ordering and cancellation

The TimePoint specifies that the *value()* method is expected to be called as soon as possible after that time.
The *now()* method allows a virtual clock owned by the implementation to control time.

```
template<class TimePoint = std::chrono::system_clock::time_point>
struct TimeFlowSingleDeferred
{
  TimePoint now() noexcept;

  template<class FlowSingle>
  void submit(TimePoint, FlowSingle);
};
```

TimeManyDeferred

The shape of a *Future* and an *Executor* with time-ordering and cancellation

The TimePoint specifies that the *value()* method is expected to be called as soon as possible after that time and then again at intervals specified by the Duration.
The *now()* method allows a virtual clock owned by the implementation to control time.

```
template<class TimePoint = std::chrono::system_clock::time_point>
struct TimeManyDeferred
{
  TimePoint now() noexcept;

  template<class Duration, class Many>
  void submit(TimePoint, Duration, Many);
};
```

TimeFlowManyDeferred

The shape of a *Future* and an *Executor* with time-ordering and cancellation

The TimePoint specifies that the *value()* method is expected to be called as soon as possible after that time and then again at intervals specified by the Duration.
The *now()* method allows a virtual clock owned by the implementation to control time.

```
template<class TimePoint = std::chrono::system_clock::time_point>
struct TimeFlowManyDeferred
{
  TimePoint now() noexcept;

  template<class Duration, class FlowMany>
  void submit(TimePoint, Duration, FlowMany);
};
```

Concepts

> **Disclaimer:**
> You don't find the concepts by looking at the types in your system. You find them by studying the algorithms.

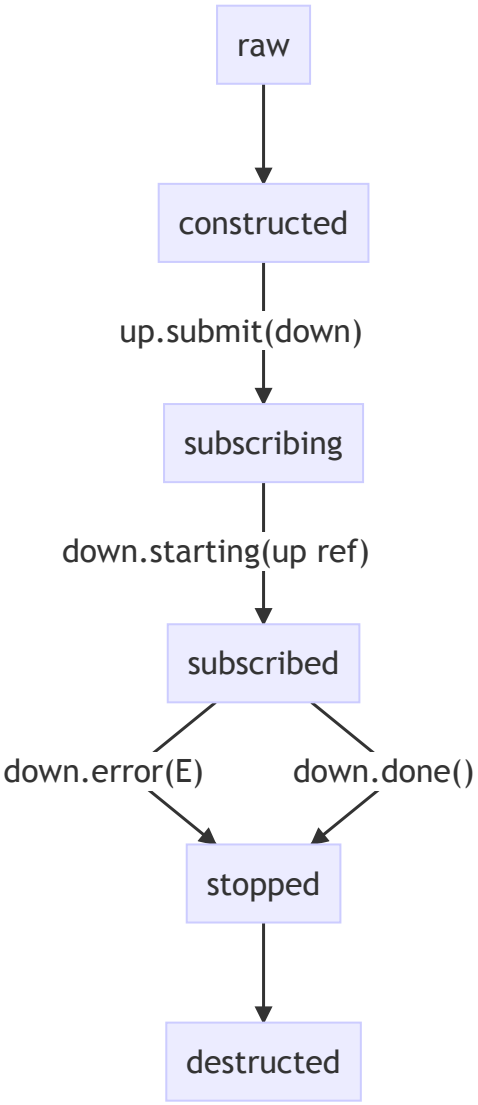
> **Extension Points:**
> The concepts below depend on new extension points that would allow each type to extend how value, error, done, stopping, starting and submit are delivered to that type.

```
template<class S, class E>
concept None =
    requires (S& s, E&& e) {
        {std::set_error(s, (E&&) e)} noexcept;
        {std::set_done(s)} noexcept;
    };
template<class D, class S, class E>
concept Deferred =
    None<S, E> &&
    requires (D& d, S&& s) {
        std::submit(d, (S&&) s);
    };

template<class S, class T, class E>
concept Single =
    None<S, E> && requires (S& s, T&& t) {
        std::set_value(s, (T&&) t); // Semantics: called exactly once.
    };
template<class D, class S, class T, class E>
concept SingleDeferred =
    Single<S, T, E> && Deferred<D, S, E>;

template<class S, class Up, class T, class SE, class UE = SE>
concept FlowSingle =
    Single<S, T, SE> && None<Up, UE> &&
    requires (S& s, Up& up) {
        {std::stopping(s)} noexcept;
        std::starting(s, up);
    };
```

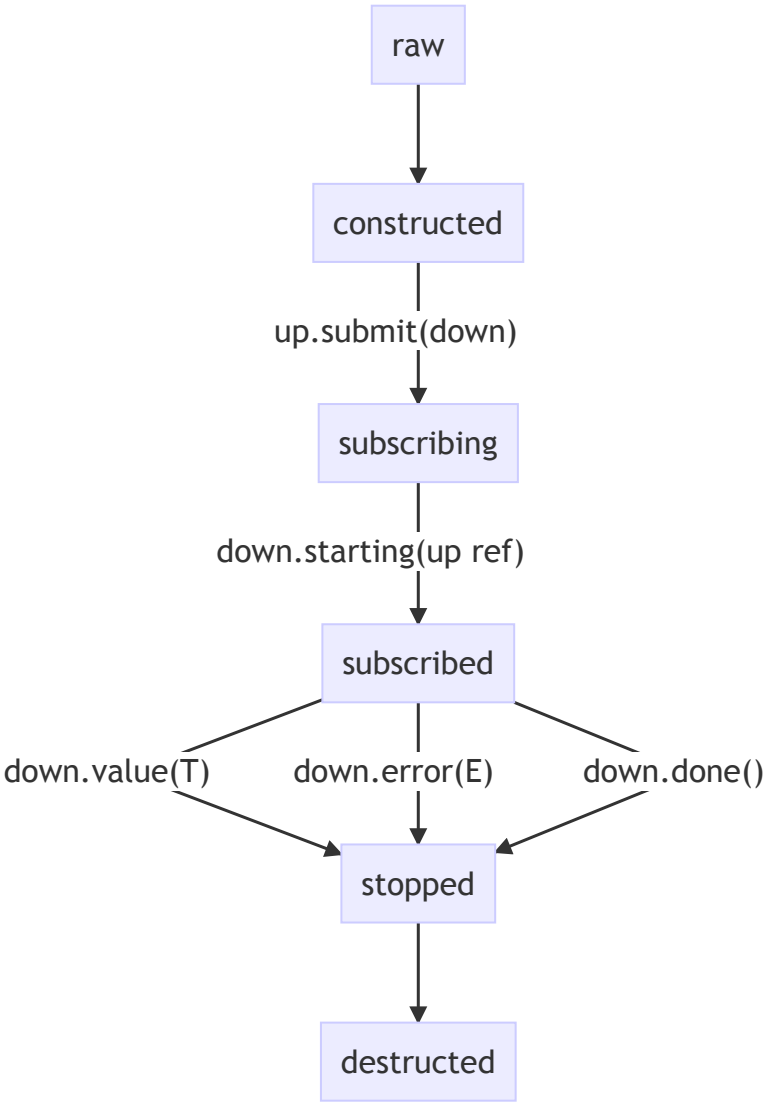
```
template<class D, class S, class Up, class T, class SE, class UE = SE>
concept FlowSingleDeferred =
    FlowSingle<S, Up, T, SE, UE> && Deferred<D, S, SE>;

template<class S, class T, class E>
concept Many =
    Single<S, T, E>; // Semantics: set_value called one or more times.

template<class D, class S, class T, class E>
concept ManyDeferred =
    Many<S, T, E> && Deferred<D, S, E>;

template<class S, class Up, class ST, class SE, class UT = count_t, class UE = SE>
concept FlowMany =
    Many<S, ST, SE> && Many<Up, UT, UE> && FlowSingle<S, Up, ST, SE, UE>;

template<class D, class S, class Up, class ST, class SE, class UT = count_t, class UE = SE>
concept FlowManyDeferred =
    FlowMany<S, Up, ST, SE, UT, UE> && Deferred<D, S, SE>;
```

Diagrams

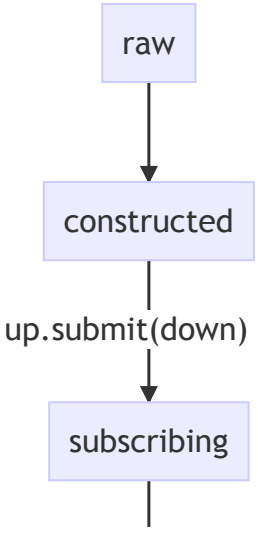
Deferred state transitions diagram

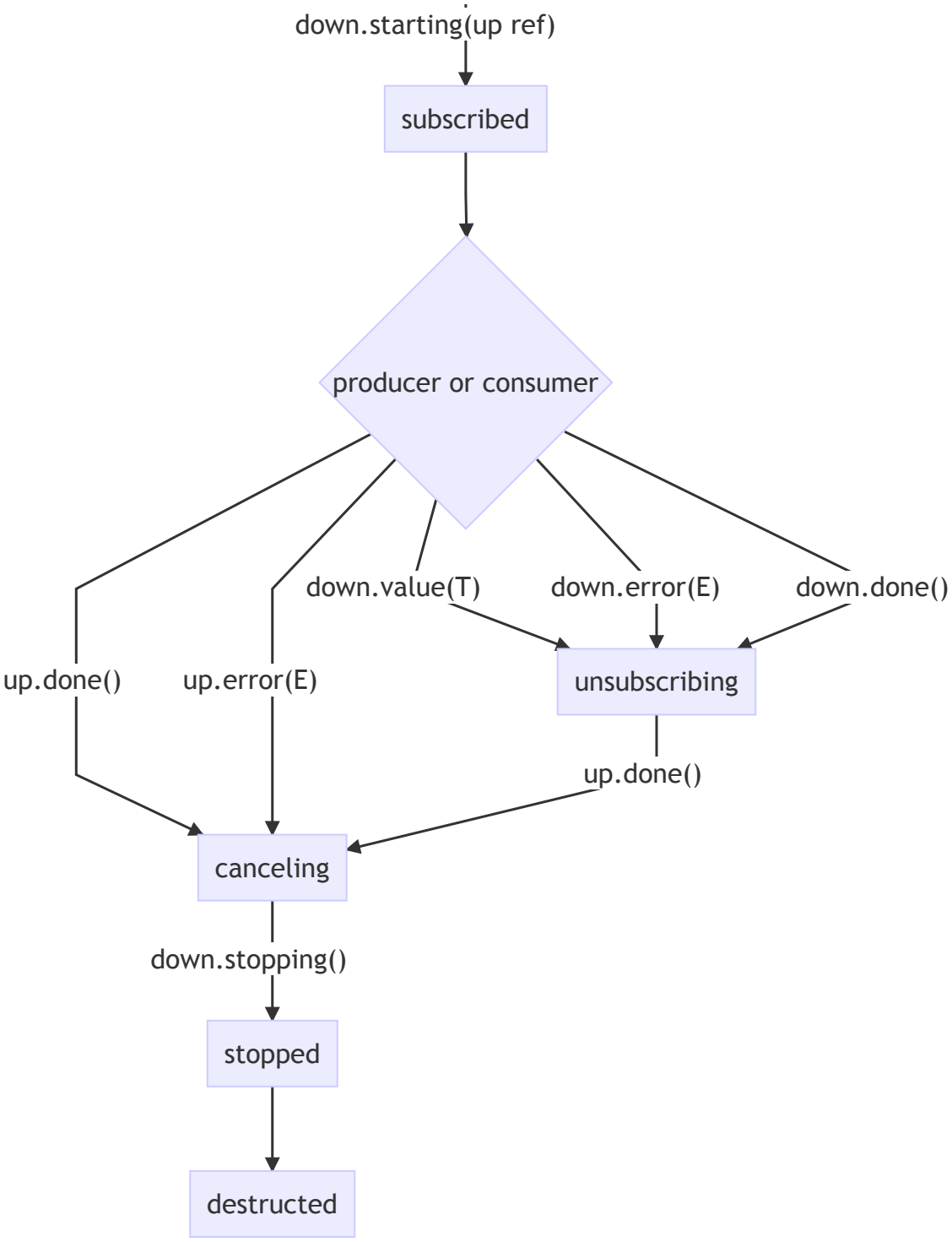


SingleDeferred state transitions diagram

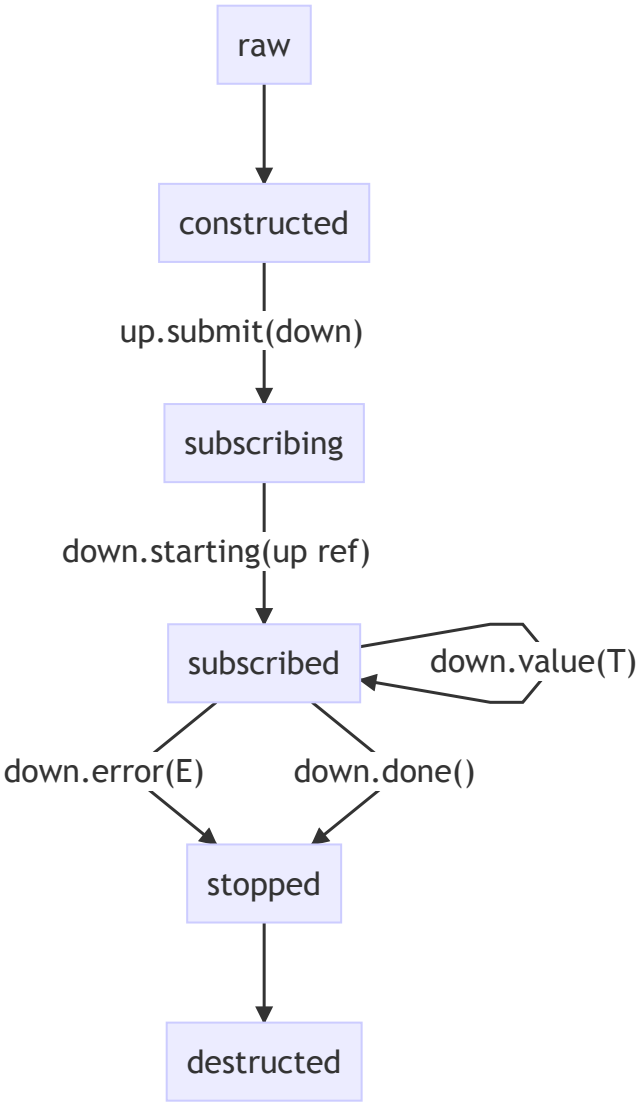


FlowSingleDeferred state transitions diagram

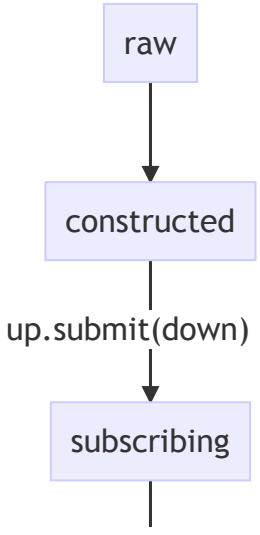


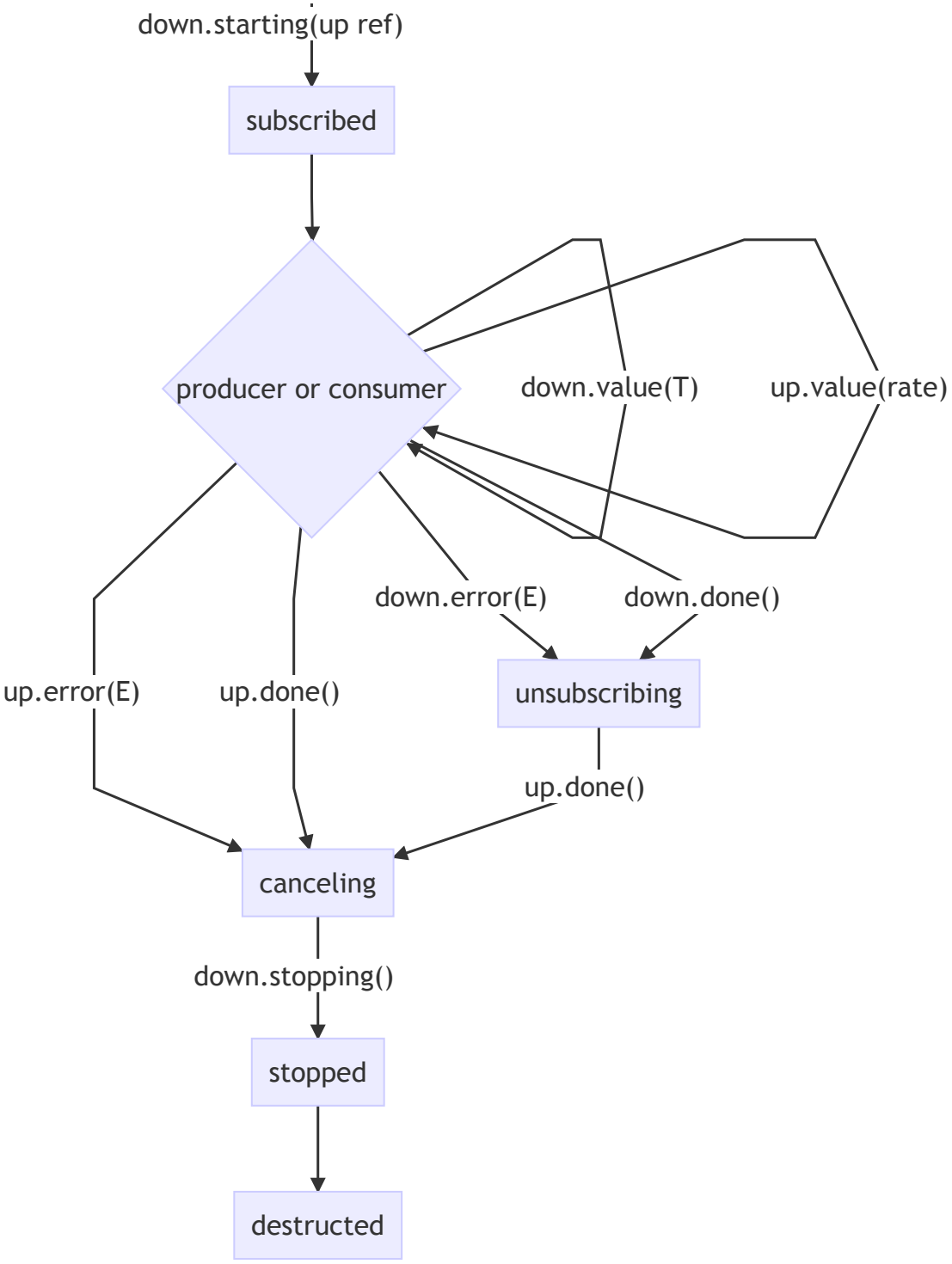


ManyDeferred state transitions diagram

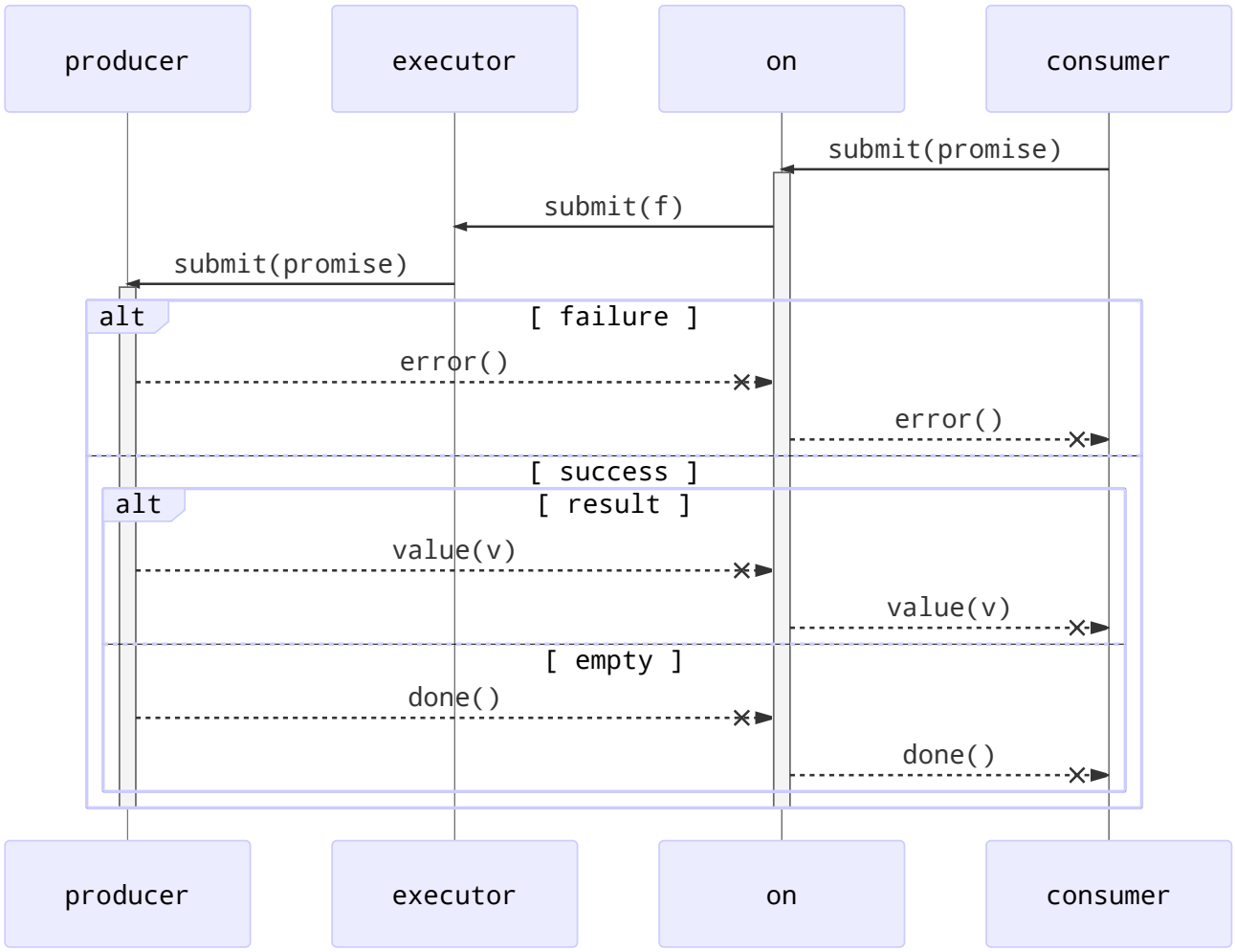


FlowManyDeferred state transitions diagram

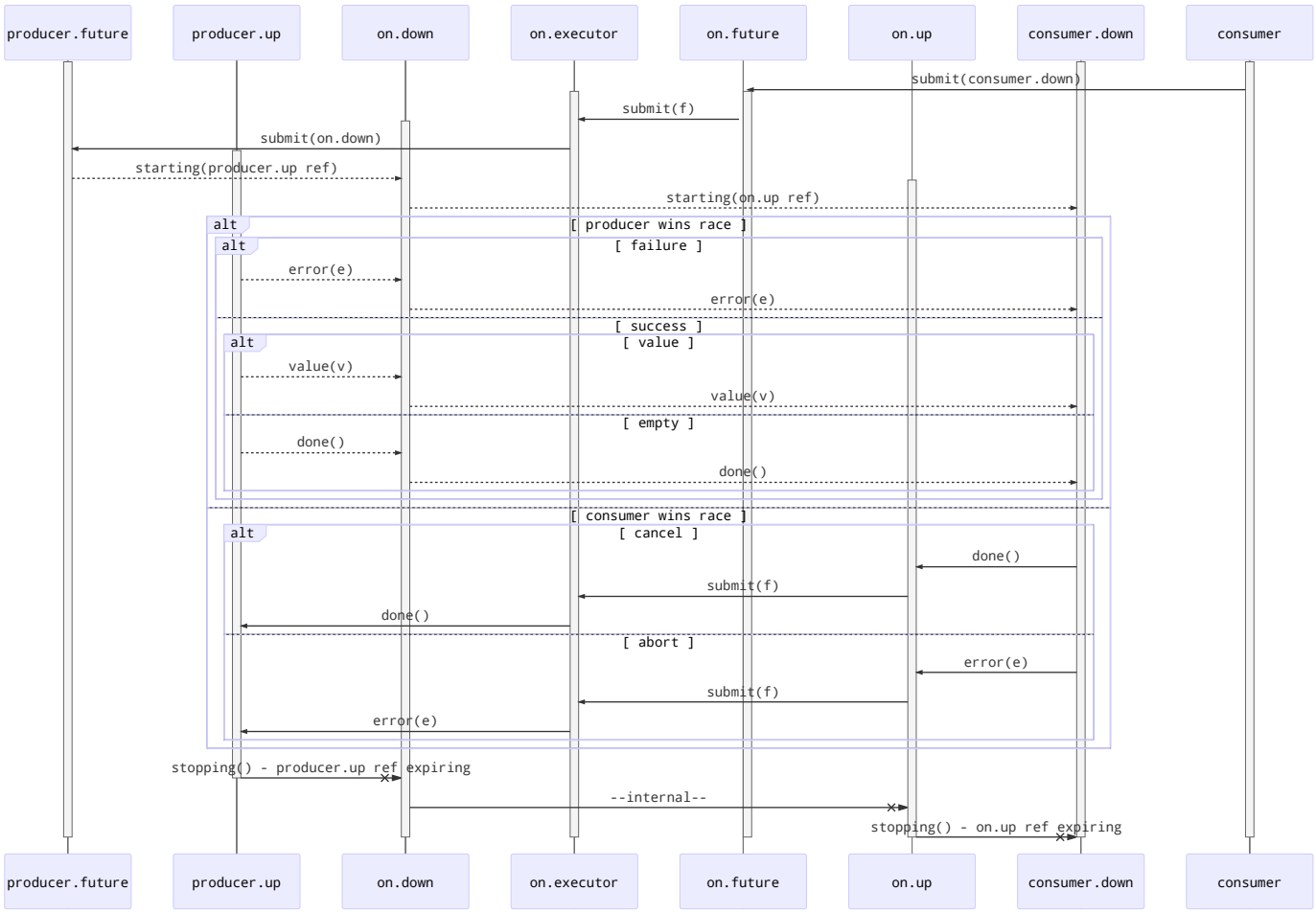




SingleDeferred on(Executor) sequence diagram



FlowSingleDeferred on(Executor) sequence diagram


producer.future
producer.up
on.down
on.executor
on.future
on.up
consumer.down
consumer
submit(consumer.down)
submit(f)
submit(on.down)
starting(producer.up ref)
starting(on.up ref)
alt
[producer wins race]
alt
[failure]
error(e)
error(e)
[success]
alt
[value]
value(v)
value(v)
[empty]
done()
done()
[consumer wins race]
alt
[cancel]
done()
submit(f)
done()
[abort]
error(e)
submit(f)
error(e)
stopping() - producer.up ref expiring
--internal--
stopping() - on.up ref expiring
producer.future
producer.up
on.down
on.executor
on.future
on.up
consumer.down
consumer