**Doc. No.:** WG21/P0904
**Date:** 2018-02-11
**Authors:** Lee Howes lwh@fb.com,
Andrii Grynenko andrii@fb.com,
Jay Feldblum yfeldblum@fb.com
**Reply-to:** Lee Howes
**E-mail:** lwh@fb.com
**Audience:** SG1

# P0904 - A strawman Future API

## Motivation

In **P0783** we discussed the abstract idea of separating continuable futures from those that are not continuable. The basic idea is that a future returned from an API should not expose that API's execution context to the caller without care, and that the standard means for returning a future from an asynchronous API should lean towards not exposing the execution context. Any continuations chained on a future returned from such an API should explicitly be associated with some execution context owned by the caller, and that this control should be colocated with the future in code.

At Facebook we have implemented this concept in the open source folly library as `folly::SemiFuture`, which does not support continuations, and the earlier existing future type `folly::Future`, which does. We've had good feedback from across the company on this basic design and numerous libraries are in the process of converting their `Future`-returning code to return `SemiFuture` to add this layer of safety. Note that in folly, for consistency with earlier executor modifications, the `via` customization point is implemented as a method on `SemiFuture` and `Future`.

## Summary

This paper aims to strengthen some of these ideas, and to start to tie futures together with executors as proposed in **P0443**, to understand how synchronization can work and to look at the interaction with bulk execution.

Executors add bulk execution and greedy continuation capabilities - the ability to use then_execute to have the executor wait directly on the future. In both cases we wish to be able to expose this functionality such that we can benefit from it on the interfaces of futures, but without loss of efficiency. This paper aims to make the link to those executor interfaces clear, to help us better understand what interfaces we really need in the executors to implement futures.

Finally this paper aims to start to solidify the forward progress delegation requirements for the future APIs, to make sure we expose appropriate interfaces for executors and to be confident that we can deal with execution agents that offer different forward progress guarantees in a safe manner.

# Existing Executor Concepts

We rely on a set of type requirements from **P0443**, which we loosely describe here as concepts, without using correct concept syntax, to emphasise that they describe sets of types satisfying some basic set of requirements. In each case we simply note the set of functions on the type that matter for our purposes and any other information that we see as important to communicate.

Executor is a simplified version of the concept taken from the executors paper. In practice we have a set of these types exposing different capabilities and some can be converted to others. The necessary operations are summarised here.

The fundamental primitive we need to implement futures efficiently is a one-way execute. This is fundamental because in the absence of more information, futures will dispatch work to the executor lazily when the dependencies are satisfied. The executor in this case will support the `OneWayExecutor` requirements:

```
concept OneWayExecutor {
      template<class F> void execute(F&& f);
};
```

An executor may also follow the `TwoWayExecutor` requirement:

```
concept TwoWayExecutor {
      template<...>
      ReturnFutureType<T> twoway_execute(F&& f);
};
```

Where the returned future is satisfied by the return value of `F`. We can convert a one-way execution into a two-way execution trivially. We can convert a two-way execution trivially into a one-way if the returned future is *Continuable* (described below).

An executor may support continuations directly such that it has a `then_execute` operation that creates a dependency between function `f` and some future `fut`:

```
concept ThenExecutor {
      template<...>
      ReturnFutureType<T> then_execute(F&& f, InputFutureType<T2>&& fut);
};
```

We can use an executor that supports this operation as its basic operation if `InputFutureType` matches our source future.

An executor may support bulk operations such that one call to the execute function launches some amount of work, optionally greater than one instance. Bulk operations come in the three variants above and can be used to emulate the single launch forms of the operations by dispatching only one instance. As an example, the `BulkOneWayExecutor`:

```
concept BulkOneWayExecutor {
      template<...>
      void bulk_execute(
            F&& f, executor_shape_t<ExecutorType> s, Synchronizer);
};
```

The shape defines the set of instances dispatched. The synchronizer type is an executor-specific type that enables synchronization of the set of instances.

We note the above because the Future concepts below depend on them. Please read **P0443** for more details.

Finally, we require the expected type defined in **P0323R2**. Although this is exemplary, and could be replaced with some other type that satisfies similar requirements such as `folly::Try` as necessary.

# Future concepts

We split futures into two new concepts: `SemiFuture` and `ContinuableFuture`.

`SemiFuture` represents a future value, but only has the potential to provide access to that value. It is defined defined as a type that has a `via` operation exposed as a customization point that itself takes an r-value of the `SemiFuture` type and an executor and that returns a `ContinuableFuture` of that executor. The return concept of `ContinuableFuture` will be described next.

`SemiFuture` has no associated executor, and there is therefore no `.then` operation on it. `SemiFuture` does not directly permit continuations. Rather, a `SemiFuture` may be converted to a `ContinuableFuture` by attaching an executor, and that `ContinuableFuture` will permit continuations.

```
template<class T>
concept SemiFuture {
      explicit SemiFuture(/* implementation-defined ContinuableFuture */&&);
```

```
        // Move constructible
        SemiFuture(/*self type*/&&);

        // get and get_expected are both destructive.
        // get will throw on exception. get_expected will return either a value
        // or an exception.
        T get() &&;
        ExpectedType<T> get_expected() noexcept &&;

        // Wait is not destructive.
        SemiFuture<T>& wait() noexcept &;
        SemiFuture<T>&& wait() noexcept &&;

        bool is_ready() noexcept;
};
```

A `SemiFuture` can be constructed from some matching `ContinuableFuture` as a means of type erasing the executor for safe return from APIs. This is important because it means a full chained set of futures can be used and then the executor erased for returning from a library. For example something along the lines of:

```
SemiFuture<int> doThings() {
   auto f = doWork();
   Future<int> f2 = f.then(sometask);
   return SemiFuture<int>{std::move(f)};
}
```

The `via` customization point of `SemiFuture` will return a `ContinuableFuture`:

```
template<
        OneWayExecutor Ex,
        SemiFuture<T> ConcreteSemiFuture,
        ContinuableFuture<Ex> CF>
CF via(ConcreteSemiFuture&&, Ex);
```

The precise type of the returned ContinuableFuture from via depends on the exeuctor. It may be a custom future type. The executor type that is part of the future returned by a call to the `via` customization point need not match that passed. A valid extension of this interface would be to require that the Executor passed to via be non-blocking, or be convertible to one that is non-blocking using require operations. This would preclude use of an inline executor but would increase the safety of the API overall.

Calls to `get`, `get_expected` and to `wait` are blocking and support forward progress delegation. If present, an executor associated with the `SemiFuture` (which may have been constructed from a `ContinuableFuture`) may delegate its forward progress to the next executor in the future chain attached with `via(std::move(sf), ex)`. It should not be assumed to be safe to call a blocking future operation from a weaker-than-concurrent agent on an unknown future type. See section on Synchronization below.

`get` will throw if the SemiFuture holds an exception, `get_expected` will return an expected type that wraps either the value or an `exception_ptr`.

We add the ability to enqueue continuations on a future using the `ContinuableFuture` concept. ContinuableFuture has `.then` and `.bulk_then` methods and is always associated with an executor, which we propose exposing explicitly in the type.

```
template<class T, Executor Ex>
concept ContinuableFuture : SemiFuture  {
      using executor_type = Ex;
      using semi_future_type = /* implementation-defined */

      // Move constructor
      ContinuableFuture(/*self type*/&&);

      template<class ReturnT, class F, Executor Ex2>
      ContinuableFuture<ReturnT, Ex> then(F&&);

      template<
            class ReturnT,
            class F,
            Executor Ex2,
            class SharedFactory,
            class ResultFactory>
      ContinuableFuture<ReturnT, Ex2> bulk_then(
            F&& f,
            executor_shape_t<Ex> shape,
            SharedFactory&& s,
            ResultFactory&& r);

      Ex get_executor() noexcept;
      semi_future_type semi() &&;
};
```

A call to `via(std::move(cf), ex)` is allowed to return `std::move(cf)` if the passed executor instance, ex, matches the executor attached to the `ContinuableFuture`.

The matching `SemiFuture` type that can collapse the `ContinuableFuture` is exposed through the `semi_future_type` type export. A `ContinuableFuture` can be converted directly to that type using the `semi()` method for convenience.

`get`, `get_expected` and `wait` are equally applicable to `ContinuableFuture` and should be supported for any `ContinuableFuture` type with the same semantics as for `SemiFuture`.

The factory parameters of `bulk_then` are equivalent to those of `bulk_two_way_execute` in **P0443**.

`bulk_then` may be delegated to the executor for efficient execution:

- If `Ex` is a `BulkExecutor` and `SharedFactory` and `ResultFactory` are supported by that executor's `bulk_execute` operation, then that may be called to implement `bulk_then` lazily.
- If `Ex` does not satisfy `BulkExecutor` but is convertible to a `BulkExecutor` using require, and the parameters of the result match as above, then `Ex::bulk_execute` may be used to implement `bulk_then` lazily. Note that in this case `Ex` and `Ex2` may be different types.

`then` may be delegated to the executor for greedy evaluation and task-graph creation:

- If `Ex` is a `ThenExecutor` and its then_execute accepts `*this` as its source future, then_execute may be called to implement then greedily.
- If `Ex` does not satisfy `ThenExecutor` but is convertible to a `ThenExecutor` using `require`, and the resulting executor accepts `*this` as a source type, `Ex::then_execute` may be called to implement `then` greedily. Note that in this case `Ex` and `Ex2` may be different types.

`bulk_then` may be delegated to the executor for greedy evaluation and task-graph creation:

- If `Ex` is a `BulkThenExecutor` and its `Ex::bulk_then_execute` accepts `*this` as its source future and the `SharedFactory` and `ResultFactory` parameters are valid, bulk_then_execute may be called to implement bulk_then greedily.
- If `Ex` does not satisfy `BulkThenExecutor` but is convertible to a `BulkThenExecutor` using `require`, the resulting executor's `bulk_then_execute` accepts `*this` as its source future and the `SharedFactory` and `ResultFactory` parameters are valid, `bulk_then_execute` may be called to implement `bulk_then` greedily.. Note that in this case `Ex` and `Ex2` may be different types.

It is implementation-defined for a given Future whether, if `bulk_execute` and `then_execute` are both available, how `bulk_then` will be delegated. Otherwise the execute method will be called on the executor lazily when the future is satisfied.

Valid signatures for continuation function `F` passed to `then` are:

```
expected<ReturnT, exception_ptr>      (expected<T, exception_ptr>&&);
expected<ReturnT, exception_ptr>      (Ex&, expected<T, exception_ptr>&&);
SemiFuture<ReturnT>                   (expected<T, exception_ptr>&&);
SemiFuture<ReturnT>                   (Ex&, expected<T, exception_ptr>&&);
```

Valid signatures for continuation function `F` passed to `bulk_then` are:

```
expected<ReturnT, exception_ptr> (
      expected<T, exception_ptr>&&, ResultFactory&, SharedFactory&);
```

```
expected<ReturnT, exception_ptr> (
      Ex&,
      expected<T, exception_ptr>&&,
      ResultFactory&,
      SharedFactory&);
SemiFuture<ReturnT> (
      expected<T, exception_ptr>&&,
      ResultFactory&,
      SharedFactory&);
SemiFuture<ReturnT> (
      Ex&,
      expected<T, exception_ptr>&&,
      ResultFactory&,
      SharedFactory&);
```

Where `ResultFactory`, `SharedFactory` are constructed and used according to the rules of `bulk_then_execute` in **P0443**.

A continuation that returns `ReturnT`, `ContinuableFuture<ReturnT>` or some other type convertible to either of the known return types would also be supported with the obvious conversions.

Optionally providing the executor to the continuation offers the opportunity to query the executor for information about the system.

Continuations that return futures, that is those of the form:

```
f.then([](T&& t){
      return FutureType<T>(doSomethingTo(std::forward<T>(t)));});
```

are supported. A `ContinuableFuture` will be returned in these cases, such that the resulting expression is semantically equivalent to:

```
f.then([](Ex& ex, T&& t){
      return via(ConcreteSemiFuture<T>(
            doSomethingTo(std::forward<T>(t))), ex);});
```

The future returned by the continuation will if necessary be wrapped into a future that completes on the original future's executor, such that the future returned by the call to `f.then` always completes on `f`'s executor to avoid leaking executors.

# Defer

When working with folly we have found specific cases where we do want some sort of continuation on a `SemiFuture`, but with very specific and strongly-defined semantics. As an example, take a networking library that receives data from the network and wants to deserialize it.

```
SomeComplexType getFromNetwork() {
	SemiFuture<string> data = getData();
	return deserialize(data.get());
}
```

In this case blocking is clearly not what we want. Facebook libraries currently tend to accept an executor on construction and use that to return the data. However, the usual case is that we actually want to deserialize the data in some execution context associated with the caller. That gives us the following:

```
SemiFuture<SomeComplexType> getFromNetwork() {
	SemiFuture<string> data = getData();
	return data.defer([](string&& data){return deserialize(data)};
}
```

This looks like a standard call to then, but note that we do not attach an executor. Instead we can call get on the return value:

```
auto v = get(DrivableExecutor{}, getFromNetwork());
```

Where `DrivableExecutor` is an exemplary executor that provides only delegated forward progress.

In this case, deserialize is going to run during the call to `get`. Defer adds a callback to the `SemiFuture` that delegates its forward progress guarantee to either the caller of `get`, as above, or to the next executor in the chain, as in:

```
auto f = via(getFromNetwork(), e);
```

Note that we have tightly coupled the executor we set with the operation, rather than with the entire network library.

We therefore extend the SemiFuture concept with a defer method:

```
template<class T>
concept SemiFuture {
	explicit SemiFuture(/* implementation-defined ContinuableFuture */&&);

	// Move constructor
	SemiFuture(/* implementation-defined */&&);

	template<class ReturnT>
	SemiFuture<ReturnT> defer(F&&);

	// get and get_expected are both destructive.
	// get will throw on exception. get_expected will return either a value
	// or an exception.
	T get() &&;
```

```
        ExpectedType<T> get_expected() noexcept &&;

        // Wait is not destructive.
        SemiFuture<T>& wait() noexcept &;
        SemiFuture<T>&& wait() noexcept &&;

        bool is_ready() noexcept;
};
```

Valid signatures for continuation function F passed to defer are:

```
        expected<ReturnT, exception_ptr>     (expected<T, exception_ptr>&&);
        SemiFuture<ReturnT>                  (expected<T, exception_ptr>&&);
```

With conversion rules defined as for `.then`.

Callbacks added using calls to `defer` are chained as callbacks added with then, as if through a chain of futures, and hence are satisfied after any previous callbacks and in order of addition. Delegation of forward progress guarantees is transitive such that in code like:

```
{
        auto s = promise.getSemiFuture();
        auto f1 = via(std::move(s), e);
        auto f2 = std::move(f1).then(task1);
        auto f3 = std::move(f2).then(task2);
        auto s2 = ConcreteSemiFuture{f3};
        auto s3 = s2.defer(task3);
        auto result = get(DeferredExecutor{}, std::move(s3));
}
```

Executor `e` may delegate its forward progress to the caller of `get` and all intermediate calls to defer will run inline with the caller of `get`.

# Standardised Future type

We propose that we do include a basic future type, that `std::async` and other core APIs can evolve to return, and that is efficient enough to use as a standard type-erasing wrapper for any types that implement the `Future` or `SemiFuture` concepts.

While other future types may be created through library-specific means, to use the standard future for purposes other than standard APIs (such as `std::async`) the promise provides the means both of creation, and of setting the value. We therefore require a promise type with a void specialization. The promise type can have a value set on it, and will return a `StandardSemiFuture`. This is important because no continuation may be attached to that future, so we will not get direct call-through on the promise setter without explicit control.

```
template<class T>
class StandardPromise {
	public:
	StandardSemiFuture<T> get_future();
	void set_value(T&&);
};

template<>
class StandardPromise<void> {
	public:
	StandardSemiFuture<void> get_future();
	void set_value();
};
```

We have a standard implementation of the `SemiFuture` concept. This may share state with `StandardPromise` and `StandardContinuableFuture`.

It is safe to construct a `StandardSemiFuture` directly from a value and calling `get` on such a future should always be expected to be ready.

```
template<class T>
class StandardSemiFuture {
	public:
	// StandardSemiFuture may be constructed already complete
	StandardSemiFuture(T);
	StandardSemiFuture(StandardSemiFuture&&);

	// StandardSemiFuture may type erase any ContinuableFuture
	template<Executor Ex, ContinuableFuture<T, Ex> CF>
	StandardSemiFuture(CF&&);

	// Similar to .then but with very specific semantics.
	// Defers work to be boost-blocked on a
	// to-be-attached executor, or at get time.
	template<Callable F, class ReturnT>
	StandardSemiFuture<ReturnT> defer(F&&);

	// get and get_expected are both destructive.
	// get will throw on exception. get_expected will return either a value
	// or an exception.
	T get() &&;
	expected<T, exception_ptr> get_expected() noexcept &&;

	// Wait is not destructive.
	StandardSemiFuture<T>& wait() noexcept &;
	StandardSemiFuture<T>&& wait() noexcept &&;
```

```
      bool is_ready() noexcept;
};
```

Of course, we need a specialization of the via customization point for `StandardSemiFuture`:

```
template<class T, Executor Ex>
/* implementation-defined */ via(StandardSemiFuture<T>&&, Ex);
```

Note that while StandardContinuableFuture is the obvious choice here, the actual future type is dependent on the executor. The executor type may be modified with require operations, and the future type will depend on the combination of the executor type and value type.

The standard version of `ContinuableFuture` is typed on the Executor. A polymorphic executor is a valid option here and could be used as the means to pass a future around libraries that want the continuable future but are happy with type erasing the executor.

```
template<class T, Executor Ex>
class StandardContinuableFuture {
      public:
      using executor_type = Ex;
      using semi_future_type = StandardSemiFuture<T>;

      // Move constructor
      StandardContinuableFuture(StandardContinuableFuture&&);

      template<class ReturnT, class F, Executor Ex2>
      ContinuableFuture<ReturnT, Ex> then(F&&);

      // Will be implemented as:
      // return executor_.then_execute(std::move(*this), std::forward<F>(f))
      // if E has a then_execute method that takes ContinuableFuture <T, E>
      // as a future parameter.
      template<Callable F>
      StandardContinuableFuture <invoke_result_t<F, Args...>, Ex> then(F&& f);

      // Will be implemented as:
      // return executor_.bulk_then_execute(std::move(*this),
      // std::forward<F>(f)) if Ex has a bulk_then_execute method that takes
      // StandardContinuableFuture <T, E> as a future parameter.
      template<Callable F>
      StandardContinuableFuture <invoke_result_t<F>, Ex> bulk_then(F&&);

      // get and get_expected are both destructive.
      // get will throw on exception. get_expected will return either a value
      // or an exception.
      T get() &&;
      expected<T, exception_ptr> get_expected() noexcept &&;
```

```
        // Wait is not destructive.
        StandardContinuableFuture<T, Ex>& wait() noexcept &;
        StandardContinuableFuture<T, Ex>&& wait() noexcept &&;

        bool is_ready() noexcept;

        Ex get_executor() noexcept;

        semi_future_type semi() &&;
};
```

The extension point is valid here too. Note that the type of the executor and future may change based on how the way the executor is defined.

```
template<class T, Executor Ex>
/* implementation-defined */ via(StandardContinuableFuture<T>&&, Ex);
```

A `StandardContinuableFuture` is constructible from any other future type that implements the `ContinuableFuture` concept and shares the same executor.

# Synchronization

Synchronization between futures on potentially different agents is dealt with in two ways:

1. It is always safe to add a callback to a future - any state shared between execution agents must allow calls to `via`, and calls to `.then` and `.bulk_then` to be executed irrespective of where any promise associated with the future is located.
2. The executor implements synchronization on call to `execute` (work enqueue) when the dependencies are satisfied, or earlier during a call to `then_execute` if we are greedily enqueuing. The type of agent on which this is safe is defined by the blocking properties of the `execute` operation.
3. A custom future type can chain by internal magic, or by implementing `then_execute` on an associated Executor type and customising it for the future type.

It is therefore not safe to call `.get()`, `.get_expected()` or `.wait()` from a weaker-than-concurrent execution agent on an unknown future type. Synchronization is made safe by transforming the future using via with a known executor type that is aware of the execution agent and only calling `.get()` on the resulting future.

# Open Questions

## Continuations and exceptions

Should we support pattern-matching continuations or only an expected parameter.

This would mean only supporting:

```
auto f2 = f.then(
      [](expected<T, exception_ptr>&& a){
            /* Do success and exception */});
```

Which could be expanded with more general pattern matching capabilities on the expected type, or on all types, for example:

```
auto f2 = f.then([](Expected<T>&& a){
  a.match(
    [](T&& value){
      // Do success
    },
    [](exception_ptr exception){
      // Do exception
    });
  });
```

Instead of embedding the support in the future model directly with:

```
auto f2 = f.then([](T&& a){
            // Do success
      }).error([](exception_ptr exception) {
            // Do exception
      });
```

or

```
auto f2 = f.then([](T&& a){
            // Do success
      },
      [](exception_ptr exception) {
            // Do exception
      });
```

Our experience at Facebook makes us lean very strongly towards the expected version of error handling, although adding then_value and then_error chaining that are bypassed by the non-matching result state is an extension we considered and defer for later. The big problem with a double-closure approach to error handling is that developers have to deal with two closures that will often share state. This is clumsy and a single expected type makes for a much cleaner model.

## Exception pass-through

In the absence of exception handling, and a function that takes a value not an ExpectedType, do we abort, or do we pass the exception past that function and into the next in the chain, not running that particular continuation at all?

## Delegation of forward progress and executors

If we chain futures:

```
f.then(thing).then(thing).then(thing);
```

and that work is added to the executor lazily when each future in turn completes, then it isn't really obvious how the forward progress delegation works. It is likely that we need some sort of drive functionality on the executors here, to expose an API from which we can provide the execution context that forward progress is delegated to. This could be through a `drive` customization point overloaded for each executor that provides such functionality.

In that case, a call to `get` on the result of the above chain would call `drive(Ex&)` on the executor. Executors would have to be able to drive each other in turn to make this propagate. The best way to do that might be for each stage in the future chain to reference the previous future's executor as well as its own, and then allow a blocking operation to propagate through that chain as far as drive customization points allow.

As one example, work deferral can be implemented as an executor that does nothing until `drive` is called. In folly we do this using a custom executor for work chaining that knows about the previous executor and the callback, then implements the chaining using a state machine. A static thread pool could delegate in that it has a set of threads that tries to perform work, but if the threads are all blocked at the point `get` is called, then work in the queue could be run inline with the caller of `drive` (and in turn of `get`) allowing the total thread set to scale with the number of waiters.

## Removing defer

Most importantly, with a clean definition for forward progress delegation, we can be confident in dropping `.defer()` and relying on a deferred executor type that only executes work delegated to the next executor in the chain.

## Blocking get

There is an inherent problem with any solution that requires that either the future be transformed by an executor to be safe on a given agent, or that a given locally safe synchronization primitive is provided. There is no guarantee that the executor/synchronization primitive is safe for the current agent in the general case.

It may then be that what we actually want to do is define, for every executing agent some executor that provides the appropriate functionality and that, when necessary, will be driven by that agent to make progress. In that situation we can make it safe to call wait methods on arbitrary futures, as the underlying implementation would do something like:

```
Future Future::wait() {
      this->via(get_local_executor()).wait();
}
```

and the future customised by via would be safe to wait on directly (and not transform itself again by comparison of executors).

This depends on a good definition of agent-local storage.

### Removing get completely

A final option I'd like to consider is making SemiFuture purely a potential future with no get or then functionality. Any blocking or continuation behaviour would then require an executor which would handle synchronization problems because get() could always be handled using then. That would make SemiFuture a set of types that may wrap future values and are convertible to ContinuableFutures, which would remove some misuse cases of using a blocking get on a device without support - but calling get against a future with the wrong executor (or wrong synchronization primitive) would still be a failure case so it is unclear how much it really helps.