Advanced Futures and Promises in C++

Tamino Dauth[a] and Martin Sulzmann[a]

a Karlsruhe University of Applied Sciences

Abstract Futures and promises are a high-level concurrency construct to aid the user in writing scalable and correct asynchronous programs. We review the state of the art in C++ where we focus on C++17 and two expressive libraries, Boost.Thread and Folly. There is no agreement on the operations that shall be supported, we encounter subtle differences in their use and point out potentially buggy implementations. We identify a small set of core features based on which *advanced* future and promises can be easily implemented and possibly extended. Advanced futures and promises combine the best of ideas found in Folly and Scala. Our results are also applicable to other languages besides C++.

ACM CCS 2012

- *General and reference → Computing standards, RFCs and guidelines;*

Keywords concurrency, futures, promises, C++

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Parallel and multi-core programming

```
std::future<double> exchangeRate = std::async([] ()
  {
    return currentValue(USD, EUR);
  }
);

std::future<double> purchase = std::async([exchangeRate = std::move(exchangeRate)]
  () mutable
  {
    return buy(amount, exchangeRate.get());
  }
);
```

Listing 1 Currency Exchange with C++17 Futures

1 Introduction

Futures and promises [1, 4] are a high-level concurrency construct to support asynchronous programming. A future can be viewed as a placeholder for a computation that will eventually become a available. A promise is a form of future where the result can be explicitly provided by the programmer. Our interest is to investigate futures and promises in the context of C++ where they have been added to the latest language extension, referred to as C++17 [13].

C++17 Futures Listing 1 gives an implementation in C++17 of a standard example to change Euros into US dollars. We assume a function `currentValue` to determine the exchange rate between the two currencies. As we do not want to block the main thread, we carry out the computation asynchronously.

In C++17 this is achieved via `std::async`. This function takes as an argument a parameterless function and the run-time ensures that the function is executed asynchronously. In C++, we need to wrap the call `currentValue(USD, EUR)` into a lambda function. Brackets [] capture variables used within a lambda and parentheses () mention the arguments. There are no captured variables and the function is parameterless. Hence, the empty bracket [] and empty parentheses ().

The result is a value of type `std::future<double>`. Values of such type represent a computation whose result may still be undetermined. We can query the result via the get method. This method will block until the result becomes available. We say the future is *completed*. There are two possible outcomes. (1) Completion is successful in which case get yields the result of the computation. (2) Completion fails in which case an exception is raised. Based on the exchange rate, we then purchase US dollars, with the help of another auxiliary function `buy`. As we call `exchangeRate.get()`, variable `exchangeRate` needs to be captured. By default C++ futures are not shareable and therefore cannot be copied. Hence, we perform a move instead. The actual amount can be queried asynchronously via the `purchase` future. Unfortunately, futures as found in C++17 are rather limited.

First, C++17 does not provide any executors like thread pools. By default it is up to the implementation if `std::async()` spawns a new thread or not. See [13]. The program-

```
 1 boost::basic_thread_pool ex;
 2 boost::future<double> exchangeRate = boost::async(ex, [] ()
 3   {
 4     return currentValue(USD, EUR);
 5   }
 6 );
 7
 8 boost::future<double> purchase = exchangeRate
 9   .then([] (boost::future<double> f)
10   {
11     return buy(amount, f.get());
12   }
13 );
```

■ **Listing 2** Currency Exchange with Boost.Thread

mer might pass the launch policy manually to std::async() to specify whether it should always spawn a new thread or let the result be evaluated lazily by a requesting thread. The creation of one thread per std::async() call might be slow for many asynchronous calls since every thread has its own stack. Therefore, the thread creation and context switches are slower than for lightweight tasks which are also missing in C++17.

Second, C++17 lacks operations to compose futures. The user is required to write lots of boilerplate code. This is error prone and obfuscates the program logic. In our example, the purchase depends on the exchange rate. As there is no support to (sequentially) compose futures in C++17, we manually query the exchange range via get (line 11). This also requires us to 'move' the exchangeRate future within the lambda body.

Boost.Thread Futures In Listing 2, we find a recast of the currency exchange example in terms of Boost.Thread [7]. Boost.Thread provides executors such as a basic thread pool. This clearly improves the performance compared to C++17 futures. Another improvement is the presence of the then method via which we can feed the result of one future to another future. Thus, we avoid a second async call. This improves the readability of the code and has a positive impact on performance as the number of async calls is reduced.

Folly Futures Folly [3] is another powerful C++ library that also supports futures and promises. Listing 3 implements our running example in Folly. Like Boost.Thread, Folly supports executors which are provided via the Wangle library. The via function corresponds to async. There are two further improvements compared to the Boost.Thread version.

First, Folly's then takes the result of the future whereas Boost.Thread's then expects the future itself. Thus, the call to get can be avoided which then leads to some easier to read program text. If the future on future then is applied has failed, the entire future fails. A second improvement is the filter method which acts like a guard and takes as an argument a predicate. If the predicate holds, the future will be passed through. Otherwise, the resulting future fails. Hence, the entire purchase fails as well.

```
folly::Future<double> exchangeRate = folly::via(wangle::getCPUExecutor().get(), [] ()
  {
    return currentValue(USD, EUR);
  }
);

folly::Future<double> purchase = exchangeRate
  .filter([] (double v) { return v > 1.5; })
  .then([] (double v)
    {
      return buy(amount, v);
    }
  );
```

Listing 3 Currency Exchange with Folly

```
adv::Future<double> exchangeRate = adv::async(wangle::getCPUExecutor().get(), [] ()
  {
    return currentValue(USD, EUR);
  }
);

adv::Future<double> purchaseUSDtoEUR = ...
adv::Future<double> purchaseUSDtoCHF = ...
adv::Future<double> purchase = purchaseUSDtoEUR.orElse(purchaseUSDtoCHF);

purchase.onComplete([] (Try<double> v) {
     if(v.hasValue()) {
          cout << "hooray " << v.get();
     }
});
```

Listing 4 Currency Exchange with Advanced Futures

Earlier Works We summarize. Futures and promises in C++17 are rather limited as the above example shows. There are a number of proposals, e.g. consider [6, 8], that suggest to add further functionality such as `then` etc. None of these proposals seem to have been followed up and there seems to be little hope that the situation will change much in future revisions of the C++ language standard. This is a rather unsatisfactory situation for C++ users and the quest for alternatives is necessary.

Boost.Thread [7] and Folly [3] are powerful libraries with some support for futures and promises. As there is no common agreement on the naming conventions and the set of operations that are be available on futures and promises, it is hard to assess the (often subtle) differences among libraries for C++ users.

Advanced Futures and Promises We propose *advanced* futures and promises, a set of powerful operations on futures and promises that combines ideas found in Folly and Scala's futures and promises library (Scala FP) [5]. Listing 4 shows a refinement of our running example making use of advanced futures and promises. We would like to purchase either Euro or Swiss Francs. The details are omitted for brevity. We wish to give preference to the Euro currency exchange. This is achieved via the `orElse` method.

If purchaseUSDtoEUR succeeds purchase is bound to this future. If purchaseUSDtoEUR fails but purchaseUSDtoCHF succeeds, then purchase is bound purchaseUSDtoCHF. If both fail, then purchase is bound to the first failing case purchaseUSDtoEUR. Via the non-blocking onCompletion method we can try if a result is available and then finally access the value via the blocking get method.

Advanced futures offer various functions to select among a collection of futures with respect to a specific criteria. We have seen orElse. We further find first and firstSucc for selection of the first (successful) future and variants firstN and firstNSucc for selection of the first *n* (successful) futures. Neither Folly nor Scala FP offer this functionality.

Advanced futures and promises follow a library-based approach. We show that based on a small set of core features operations such as orElse can be implemented with little effort. This is useful for experimentation as we expect that new features will be added over time.

Contributions and Outline In summary, our contributions are:

- We introduce *advanced* C++ futures and promises (Section 2) and discuss its semantics and implementation (Section 3).
- We compare ourselves against related approaches such as Boost.Thread (Section 4.1), Folly (Section 4.2) and Scala (Section 4.4).
- We compare the performance of some of our advanced operations against similar operations found in Boost.Thread and Folly (Section 5).
- We discuss how shared futures can be supported (Section 6).

The implementation and benchmarks can be accessed via

https://github.com/tdauth/cpp-futures-promises

We conclude in Section 7.

2 Interface Declarations

We specify the interface of advanced futures and promises. We start off by defining some utility classes to deal with exceptional behavior and customization of the executor via a a thread pool. Then, we provide class declarations and function prototypes for advanced futures and promises. We assume that all declarations are in the name space adv.

2.1 Utility Classes

Listing 5 shows the declaration of the class template adv::Try<T>. This class template resembles the ‘maybe‘/‘option‘ data type found in functional languages. Thus, we can represent the result of a future, either nothing, a concrete value or an exception. We find methods for each of these three cases.

```
template<typename T>
class Try
```

```
{
  public:
    Try();
    Try(T &&v);
    Try(std::exception_ptr &&e);

    T get();
    bool hasValue();
    bool hasException();
};
```

Listing 5 Try class template

Listing 6 shows the declaration of the class Executor to control the scheduling of the execution of asynchronous calls.

```
class Executor
{
  public:
    template<typename Func>
    void submit(Func &&f);
};
```

Listing 6 Executor class

2.2 Advanced Futures

Listing 7 shows the class declaration and associated standalone functions to access and manipulate advanced futures. We distinguish between core and derived functionality. Methods and functions marked by (D) signal derived functionality.

```
template<typename T>
class Future
{
  public:
    T get();

    template<typename Func>
    void onComplete(Func &&f); // (D)

    bool isReady();

    template<typename Func>
    Future<T> guard(Func &&f); // (D)

    template<typename Func, typename S>
    Future<S> then(Func &&f);

    Future<T> orElse(Future<T> &&other); // (D)

    Future<T> first(Future<T> &&other); // (D)

    Future<T> firstSucc(Future<T> &&other); // (D)

    Future();
    Future(Future<T> &&other);
    Future(const Future<T> &other) = delete;
    Future<T>& operator=(const Future<T> &other) = delete;
};

template<typename Func>
Future<T> async(Executor *ex, Func &&f);

template<typename T>
Future<std::vector<std::pair<std::size_t, Future<T>>>>
firstN(std::vector<Future<T>>, std::size_t n); // (D)

template<typename T>
Future<std::vector<std::pair<<std::size_t, T>>>
firstNSucc(std::vector<Future<T>>, std::size_t n); // (D)
```

Listing 7 Advanced Future Interface

The get method blocks until the future on which it is called is completed. Once complete, get either yields a value or throws an exception. All other methods and functions are non-blocking. Method onComplete registers a callback function. The callback function is expected to process arguments of type adv::Try<T> and is called once the future is completed. Via isReady we can check if the computation is still in progress or the future is completed. The guard method takes a predicate and applies the predicate on the successfully completed future. This future will be simply passed through. In all other cases, the resulting future fails. Method then is similar to onCompletion but creates a new future of type adv::Future<S>. Methods orElse, first, firstSucc, and functions firstN and firstNSucc will be discussed in detail shortly.

2.3 Advanced Promises

Listing 8 shows the declaration of the class template adv::Promise<T> that specifies advanced promises.

```
template<typename T>
class Promise
{
  public:
    Future<T> future();

    bool tryComplete(Try<T> &&v);
    bool trySuccess(T &&v); // (D)
    template<typename Exception>
    bool tryFailure(Exception &&e); // (D)

    void tryCompleteWith(Future<T> &&f); // (D)
    void trySuccessWith(Future<T> &&f); // (D)
    void tryFailureWith(Future<T> &&f); // (D)

    Promise();
    Promise(Promise<T> &&other);
    Promise(const Promise<T> &other) = delete;
    Promise<T>& operator=(const Promise<T> &other) = delete;
};
```

Listing 8 Extended promise class template

Via the list of try methods we can attempt to complete, succeed or fail a promise. Promises have write-once semantics and the Boolean return value indicates if the try was successful. The With variants are non-blocking and take as arguments a future.

3 Semantics and Implementation

Our futures have read-once semantics and we can register one callback per future. So, multiple get calls yield failure. Futures and promises are non-shareable as we assume that operations on them transfer ownership to the callee. The only exception being isReady which can be called multiple times on the same future. Shared futures with multiple callbacks and read-many semantics will be discussed in the later Section 6.

3.1 Core Functionality

Much of the (advanced) functionality can be explained in terms of a small set of core methods. In case of class Future these are get, isReady and then. For class Promise we require method tryComplete. Existing libraries such as Boost.Thread and Folly pretty much support this core functionality already. For example, in case of Folly we only need to add tryComplete. See the below Listing 9.

```
template<typename T>
bool tryComplete(folly::Promise<T> &p, folly::Try<T> &&t)
{
  try
  {
    p.setTry(std::move(t));
  }
  catch (const folly::PromiseAlreadySatisfied &e)
  {
    return false;
  }

  return true;
}
```

Listing 9 Implementation of tryComplete for Folly

Folly's setTry method attempts to set the promise and yields an exception if the promise has already been set. We simply need to catch this promise to derive tryComplete.

3.2 Derived Promises Functionality

Methods trySuccess and tryFailure can easily be mapped to the tryComplete method. The same applies to methods tryCompleteWith, trySuccessWith and tryFailureWith where we in addition make use of the core future method onComplete. See Listing 10 for details.

```
bool trySuccess(T &&v) {
  return tryComplete(Try<T>(std::move(v)));
}
```

```
template<typename Exception>
bool tryFailure(Exception &&e) {
  return tryComplete(Try<T>(std::make_exception_ptr(std::move(e))));
}
void tryCompleteWith(Future<T> &&f) {
  auto ctx = std::make_shared<Future<T>>(std::move(f));

  ctx->onComplete([this, ctx] (Try<T> t) {
    this->tryComplete(std::move(t));
  });
}
void trySuccessWith(Future<T> &&f) {
  auto ctx = std::make_shared<Future<T>>(std::move(f));

  ctx->onComplete([this, ctx] (Try<T> t) {
    if (t.hasValue()) {
      this->tryComplete(std::move(t));
    }
  });
}
void tryFailureWith(Future<T> &&f) {
  auto ctx = std::make_shared<Future<T>>(std::move(f));

  ctx->onComplete([this, ctx] (Try<T> t) {
    if (t.hasException()) {
      this->tryComplete(std::move(t));
    }
  });
}
```

Listing 10 Derived Promises Functionality

For example, consider trySuccessWith. As we transfer ownership to the call site, we move the future f and create a shared pointer ctx. The callback function captures ctx and once completed, guarantees that f's destructor will be called (to reclaim memory space).

3.3 Derived Futures Functionality

Methods onComplete and guard can be straightforwardly expressed in terms of then. We assume an special purpose exception type PREDICATE_NOT_FULFILLED to indicate that the predicate could not be satisfied. See Listing 11.

```
template<typename Func>
void onComplete(Func &&f) {
  return this->then([f=std::move(f)] (Try<T> v) { f(v); });
}
template<typename Func>
Future<T> guard(Func &&f) {
  return this->then([f=std::move(f)] (Try<T> v) {
          auto x = v.get();
          if (!f(x))
                throw PREDICATE_NOT_FULFILLED;
          return x;
          });
}
```

Listing 11 Derived onComplete and guard

Listing 12 provides for implementations of methods orElse, first and firstSucc. Their intended meaning can easily be derived from the few lines of codes required to implement them.

```
Future<T> Future<T>::orElse(Future<T> &&other) {
   return this->then([other = std::move(other)] (Try<T> t) mutable {
      if (t.hasException()) {
         try {
            return other.get();
         } catch (...) {
         }
      }
      return t.get(); // will rethrow if failed
   });
}

template<typename T>
Future<T> Future<T>::first(Future<T> &&other) {
   struct SharedContext {
      SharedContext(Future<T> &&f0, Future<T> &&f1) : f0(std::move(f0)), f1(std::move(f1))
      { }

      Promise<T> p;
      Future<T> f0;
      Future<T> f1;
   };

   auto ctx = std::make_shared<SharedContext>(std::move(*this), std::move(other));
   auto future = ctx->p.future();

   ctx->f0.onComplete([ctx] (Try<T> t) {
      ctx->p.tryComplete(std::move(t));
   });

   ctx->f1.onComplete([ctx] (Try<T> t) {
      ctx->p.tryComplete(std::move(t));
   });

   return future;
}



template<typename T>
Future<T> Future<T>::firstSucc(Future<T> &&other) {
   struct SharedContext {
      SharedContext(Future<T> &&f0, Future<T> &&f1) : f0(std::move(f0)), f1(std::move(f1))
      { }

      Promise<T> p;
      Future<T> f0;
      Future<T> f1;
   };

   auto ctx = std::make_shared<SharedContext>(std::move(*this), std::move(other));
```

```
  auto future = ctx->p.future();

  ctx->f0.onComplete([ctx] (Try<T> t) {
    ctx->p.trySuccess(t.get());
  });

  ctx->f1.onComplete([ctx] (Try<T> t) {
    ctx->p.trySuccess(t.get());
  });

  return future;
}
```

Listing 12 Derived orElse, first and firstSucc

Method orElse gives preference to the future on which the method is applied. If this future fails we check for the argument. If this argument fails, we call t.get() to rethrow the exception of this. Otherwise, we return the value obtained by the other future.

Method first removes this bias and chooses the first completed future. In the implementation, we create a promise and then start a race among the this and other future. Whichever completes first will complete the promise.

Method firstSucc imposes the condition that the first successful future shall be chosen. Instead of tryComplete we employ trySuccess. If both fail, we rely on the following assumption. The promise will be deleted and then the resulting future fails. This is a standard assumption found in C++17 and Folly.

Implementations of derived operations firstN and firstNSucc are more involved. See Listings 13 and 14.

```
template<typename T>
Future<std::vector<std::pair<std::size_t, Try<T>>>> firstN(std::vector<Future<T>> &&c,
      ↪ std::size_t n)
{
  typedef std::vector<std::pair<size_t, Try<T>>> V;

  struct FirstNContext
  {
    /*
     * Reserve enough space for the vector, so emplace_back won't modify the whole vector and
          ↪ stays thread-safe.
     * Folly doesn't do this for folly::collectN which should lead to data races when the vector has a
          ↪ capacity smaller than n.
     * See the following link (section Data races):
          ↪ http://www.cplusplus.com/reference/vector/vector/emplace_back/
     */
    FirstNContext(std::size_t n)
    {
      v.reserve(n);
    }

    V v;
    std::atomic<std::size_t> completed = {0};
    std::atomic<bool> done = {false};
    Promise<V> p;
  };

  auto ctx = std::make_shared<FirstNContext>(n);
  const std::size_t total = c.size();

  if (total < n)
  {
    ctx->p.tryFailure(std::runtime_error("Not enough futures"));
  }
  else
  {
    std::size_t i = 0;

    for (auto it = c.begin(); it != c.end(); ++it, ++i)
    {
      it->onComplete([ctx, n, total, i] (Try<T> t)
      {
        if (!ctx->done)
        {
          auto c = ++ctx->completed;

          if (c <= n)
          {
            /*
             * This is only thread-safe if it does not reallocate the whole vector.
             * Since we allocated enough space, it should never happen and therefore we don't need a mutex
             * to protect it from data races.
             */
            ctx->v.emplace_back(i, std::move(t.get()));

            if (c == n)
            {
              ctx->p.trySuccess(std::move(ctx->v));
              ctx->done = true;
            }
          }
        }
      });
    }
  }

  return ctx->p.future();
}
```

Listing 13 Derived firstN

```
template<typename T>
Future<std::vector<std::pair<std::size_t, T>>> firstNSucc(std::vector<Future<T>> &&c, std::size_t n)
{
  typedef std::vector<std::pair<size_t, T>> V;

  struct FirstNSuccContext
  {
    /*
     * Reserve enough space for the vector, so emplace_back won't modify the whole vector and stays thread-safe.
     * Folly doesn't do this for folly::collectN which should lead to data races when the vector has a capacity smaller than n.
     * See the following link (section Data races):
         ↪ http://www.cplusplus.com/reference/vector/vector/emplace_back/
     */
    FirstNSuccContext(std::size_t n)
    {
      v.reserve(n);
    }

    V v;
    std::atomic<std::size_t> succeeded = {0};
    std::atomic<std::size_t> failed = {0};
    std::atomic<bool> done = {false};
    Promise<V> p;
  };

  auto ctx = std::make_shared<FirstNSuccContext>(n);
  const std::size_t total = c.size();

  if (total < n)
  {
    ctx->p.tryFailure(std::runtime_error("Not enough futures"));
  }
  else
  {
    std::size_t i = 0;

    for (auto it = c.begin(); it != c.end(); ++it, ++i)
    {
      it->onComplete([ctx, n, total, i] (Try<T> t)
      {
        if (!ctx->done)
        {
          // ignore exceptions until as many futures failed that n futures cannot be completed
              ↪ successfully anymore
          if (t.hasException())
          {
            auto c = ++ctx->failed;

            /*
             * Since the local variable can never have the counter incremented by more than one,
             * we can check for the exact final value and do only one setException call.
             */
            if (total - c + 1 == n)
            {
              try
              {
                t.get();
              }
              catch (...)
              {
                ctx->p.tryFailure(std::current_exception());
              }

              ctx->done = true;
            }
          }
          else
          {
```

```
67                 auto c = ++ctx->succeeded;
68
69                 if (c <= n)
70                 {
71                    /*
72                     * This is only thread-safe if it does not reallocate the whole vector.
73                     * Since we allocated enough space, it should never happen and therefore we don't
                           ↪ need a mutex
74                     * to protect it from data races.
75                     */
76                    ctx->v.emplace_back(i, std::move(t.get()));
77
78                    if (c == n)
79                    {
80                       ctx->p.trySuccess(std::move(ctx->v));
81                       ctx->done = true;
82                    }
83                 }
84              }
85           }
86        });
87     }
88   }
89
90   return ctx->p.future();
91 }
```

Listing 14 Derived firstNSucc

For brevity, we only consider firstSucc. We return of future of a vector of n pairs. Each component represents one of the successful futures and its index in the incoming vector. If we cannot find n successful futures, the resulting future fails.

The resulting future is represented by a promise `p`. See line 22. We proceed as follows. We first check if there is a sufficient number of futures at all. See lines 28-31. If yes we register via `onComplete` a callback for each future. The callback covers lines 39-86. The atomic (flag) variable `done` signals if promise `p` is already set. If the future fails, we atomically increment the number of failures (line 45) and check if our goal of finding n successful features can still be reached (see line 51-63). Otherwise, we found a successful feature and atomically increment the number of successes (line 67). If we are still short of n (successful) features, we add this feature to the output vector.

4 Comparison

4.1 Boost.Thread

Boost.Thread supports sufficient functionality for our core features but as of now lacks many of the advanced features we propose. For example, Boost.Thread supports `when_any` which at first look seems similar to our first. That is, to collect the first available future. Instead of being defined as a binary operation (method), `when_any` takes as argument a collection of futures. `when_any` waits for the first future complete.

Instead of reporting just the first completed future, when_any simply returns the collection of futures. So, the user needs to go through the list and check for the first available future. This is tedious and somewhat inconsistent as in between other futures, appearing earlier in the collection., may be completed.

4.2 Folly

Folly is the most expressive C++ futures and promises library we are aware of. Folly offers much of the functionality we propose but lacks certain features. For example, Folly supports isFilter which corresponds to guard. Folly lacks orElse and firstNSucc but supports variants of our first, firstSucc and firstN

In Folly, first and firstSucc are called collectAny and collectAnyWithoutException. Listing 15 shows their signatures.

```
Future<std::pair<size_t, Try<T>>> collectAny(Collection&& c);
Future<std::pair<size_t, T>> collectAnyWithoutException(Collection&& c);
```

Listing 15 Folly collectAny and collectAnyWithoutException signatures

There are some minor differences. Folly defines them as functions, taking as arguments a collection of futures, whereas we use binary operations represented as methods. Our first yields a failing future if the arguments fail. Folly's collectAny yields always a successful future which contains as the result the first completed future, represented by Try<T>. On the other hand, the behavior of collectAnyWithoutException corresponds to firstSucc. The first successful feature will be reported. If all future fail, the resulting future fails as well.

Folly also supports firstN, called collectN. However, we spotted a potential bug. In the Folly implementation we may need to resize the vector for the collection of completed futures. This possibly leads to a data race. In our implementation, this is fixed by reserving sufficient memory space. See line 15 in Listing 13.

4.3 C++ Others

We have already discussed C++17 and its limited support for futures. Other C++ libraries like Qt [12] and POCO [9] support futures as well. However, Qt lacks support for promises and only provides one default type of an executor and some basic non-blocking functions but no callback support. POCO follows a different pattern with its active object approach. Active results (futures) are returned from active methods but do not allow the use of any non-blocking functions or callbacks. For details, see here [2].

4.4 Scala

Scala FP [5] is another expressive futures and promises library and the main source of inspiration behind advanced futures and promises. We also believe that the design behind Folly's futures and promises is largely based on Scala FP. For example, Scala supports orElse which is called fallbackTo and first is called firstCompletedOf. Scala FP does not directly support firstSucc. Instead, there is find to select the first future that matches a predicated where failed futures are ignored. In Listing 16, we show that Scala FP's find is equivalent to our firstSucc + guard.

```
Future<T> find(Future<T> &&other, Func &&f) {
 return this->guard(f).firstSucc(other.guard(f));
}

Future<T> firstSucc(Future<T> &&other) {
  return this->find(other, [] (Try<T> v) { return true; });
}
```

Listing 16 Scala find = firstSucc + guard

In Scala FP, find is a standalone function, performing the selection among a collection of futures. For convenience, we assume that find is a method. Its definition in terms of firstSucc and guard is straightforward. On each future we apply the guard and then perform the selection via firstSucc. Conversely, firstSucc is a special instance of find applied on the always true predicate.

Scala FP does not support firstN (collectN in Folly), nor firstNSucc. On the other hand, Scala FP supports all of the advanced promises operations which are largely lacking in Folly.

5 Empirical Results

We test the performance of our advanced futures functionality compared to similar functionality found in Boost.Thread and Folly.

Table 1 shows the results of our performance tests. We create a binary tree of method/function calls with the height of 12 on a machine with an Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz processor. Our implementation is a bit slower compared to Folly. We believe this is due to wrapper code to implement our core features in terms of Folly.

Table 1 Performance Results

Combinators	Time (ms)
Folly collectAll	14.13
Folly collect	17.97
Folly collectN	16.80
Folly collectAny	12.43
Folly collectAnyWithoutException	12.01
Boost when_all	95.87
Boost when_any	64.90
collectNWithoutException	16.22
orElse	6.75
first	20.85
firstSucc	21.05

Table 1 Performance Results

firstN	17.42
firstNSucc	17.73

6 Shared Futures

Since C++ does not provide any garbage collection, the memory management has to be done manually. Hence, our futures are not shareable by default and our API assumes ownership transfer that leads to many uses of std::move. The consequence is that the access to a future is limited to a single access. For example, get calls move the result out of the future and therefore it has read-once semantics as discussed earlier. This can be tedious for the user and therefore C++17 offers shared futures. We show that shared futures can be readily support based on our approach by employing Folly's shared promises.

We introduce the the wrapper class template SharedFuture<T> for (by default non-shareable) futures to become shareable. The idea is to store the result of the non-shareable future in a folly::SharedPromise and therefore SharedFutures can be copied around. The programmer does not need std::move anymore. In addition, multiple callbacks are supported that can run concurrently. For details see Listing 17 below. Every copy gets a folly::SharedPromise shared pointer referring to the same object. The get call creates a non-shareable future from the shared promise, every time it is called. This allows it to be called more than once. It returns only a constant reference to prevent modifications of the result.

```
template<typename T>
class SharedFuture
{
  public:
    SharedFuture(Future<T> &&x) : me(new folly::SharedPromise<T>())
    {
      auto ctx = this->me;
      tryCompleteWith(*ctx, std::move(x._f), [ctx] () {});
    }

    SharedFuture(const SharedFuture<T> &other) : me(other.me)
    {
    }

    const T& get()
    {
      folly::Future<T> f = this->me->getFuture();
      f.wait();

      f.getTry().throwIfFailed();

      return f.value();
    }

  private:
    std::shared_ptr<folly::SharedPromise<T>> me;
};
```

Listing 17 Class template SharedFuture

It is fairly straightforward to provide the same functionality for SharedFutures. Each call will be mapped to the respective call for our standard (non-shareable) futures. The resulting is then again lifted to a shared future via the wrapper class SharedFuture. See Listing 18 where we cover orElse and first. For brevity, we omit firstN and firstNSucc.

```
SharedFuture<T> orElse(SharedFuture<T> other)
{
  return SharedFuture<T>(
    Future<T>(this->me->getFuture())
    .orElse(Future<T>(other.me->getFuture()))._f);
}

SharedFuture<T> first(SharedFuture<T> other)
{
  return SharedFuture<T>(
    Future<T>(this->me->getFuture())
    .first(Future<T>(other.me->getFuture()))._f);
}
```

Listing 18 Advanced Functionality for SharedFuture

7 Conclusion

We have reviewed the state of the art of futures and promises in C++. By combining ideas in C++ libraries such as Folly and ideas found in other languages such as Scala, we propose advanced futures and promises. Advanced futures and promises provide for a rich set of operations which may also help for a wider adoption and use of futures and promises. Our approach is entirely library-based where most features can be implemented based a small set of core features. These core features are readily available in C++.

References

[1] Henry C. Baker Jr. and Carl Hewitt. “The Incremental Garbage Collection of Processes”. In: *SIGPLAN Not.* 12.8 (Aug. 1977), pages 55–59. DOI: 10.1145/872734.806932. URL: http://doi.acm.org/10.1145/872734.806932.

[2] *Class Poco::ActiveResult*. Feb. 21, 2017. URL: https://pocoproject.org/docs/Poco.ActiveResult.html (visited on 2017-07-16).

[3] *Folly: Facebook Open-source Library*. https://github.com/facebook/folly.

[4] Daniel Friedman and David Wise. *The Impact of Applicative Programming on Multiprocessing*. 1976.

[5] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. *Futures and Promises*. http://docs.scala-lang.org/overviews/core/futures.html.

[6] *Improvements to std::future<T> and Related APIs*. Jan. 16, 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf.

[7] Vicente J. Botet Escriba. *Chapter 38. Thread 4.7.2 - 1.64.0*. Apr. 19, 2017. URL: http://www.boost.org/doc/libs/1_64_0/doc/html/thread.html (visited on 2017-07-14).

[8] *More Improvements to std::future<T>*. Jan. 18, 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3865.pdf.

[9] *Multithreading*. July 18, 2010. URL: https://pocoproject.org/slides/130-Threads.pdf (visited on 2017-07-14).

[10] Chris Mysen. *Executors and schedulers, revision 5*. Apr. 10, 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4414.pdf (visited on 2017-07-14).

[11] *Programming Languages — Technical Specification for C++ Extensions for Concurrency*. Oct. 22, 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0159r0.html (visited on 2017-07-14).

[12] *Qt Concurrent 5.9*. May 31, 2017. URL: http://doc.qt.io/qt-5/qtconcurrent-index.html (visited on 2017-07-14).

[13] Richard Smith. *Working Draft, Standard for Programming Language C++*. Mar. 21, 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf (visited on 2017-07-11).