Advanced Futures and Promises in C++

Tamino Dauth and Martin Sulzmann

Karlsruhe University of Applied Sciences

C++ User Group Karlsruhe

Agenda

- Futures and Promises explained (Scala)
- Futures and Promises in C++
- Motivation for Advanced Futures and Promises
- Core Language for Futures and Promises
- Advanced Futures and Promises
- Sharing Futures and Promises
- Upcoming Features
- Future Work
- References

Futures and Promises

Motivation

```
val hotel = getHotel()

...

// Do something with hotel
```

- External server call
- Result is not available immediately
- We do not want to block the main thread

Futures and Promises

Manual Solution

```
val hotel, mutex, condition_variable
spawn {
  mutex.lock()
  hotel = getHotel()
  mutex.unlock()
  condition_variable.notify()
}
...
mutex.lock()
condition_variable.wait(m, { hotel.isSet })
// Do something with hotel
mutex.unlock()
```

We need a concurrency abstraction!

Futures and Promises

Scala Futures

```scala
val f = Future { getHotel() }
// ...
f onComplete { hotel => bookHotel(hotel) }
// ...
f onComplete { hotel => informFriends(hotel) }
```

- Future (read-only) value
- Non-blocking computation
- Non-blocking callbacks
- Multiple reads

Futures and Promises

Scala Promises

```
val p = Promise[Hotel]
// ...
p.future onComplete { hotel => bookHotel(hotel) }
// ...
p trySuccess HotelA             ||             p trySuccess HotelB
```

- Promise (write-once) value
- Convertible into futures
- Tried by many

Futures and Promises are Composable

Extended Example: Holiday Planning in Scala

```scala
val switzerland = Future { getHotelSwitzerland() }
// ...
val usa = Future { getHotelUSA() }
// ...
val hotel = switzerland fallbackTo usa
// ...
hotel onComplete { hotel => bookHotel(hotel) }
hotel onComplete { hotel => informFriends(hotel) }
```

- Book hotel either in Switzerland or USA
- Give preference to Switzerland
- Non-blocking compositions with combinators (fallbackTo)

Futures and Promises

Results in Scala

```
val booking = Future { ... }

booking onComplete {
  hotel match {
    case Success(hotel) => bookHotel(hotel)
    case Failure(e) => println("Error: " + e)
  }
}
```

- Try[T] stores the result
- Value on success
- Exception on failure

Futures and Promises

Executors in Scala

```
val threadPool = Executors.newFixedThreadPool(5)
val ex = ExecutionContext.fromExecutorService(threadPool)

hotel onComplete { hotel => bookHotel(hotel) } (ex)
```

- Specify which threads execute callbacks
- Can be specified per callback
- Limit the number of threads
- Default executor

Futures and Promises

Summary

- Way of asynchronous programming
- Non-blocking
- Composable
- Successful or failed results
- Executors may limit concurrency
- Scala provides an advanced library

Futures and Promises in C++

Existing Libraries

- C++17: Rather limited support
- Boost.Thread: Supports callbacks and executors
- Folly: Supports callbacks, executors and combinators
- Other libraries like Qt, POCO etc.: Even more limited

Futures and Promises in C++

C++17

- async(...): Creates a future
- promise<T>: Write once semantics
- future<T>:
  - Read once semantics with `f.get()`
  - No callbacks
  - Movable only
- shared_future<T>:
  - Multiple read semantics with `f.get()`
  - No callbacks
  - Can be copied

Extended Example: Holiday Planning in C++17

```
promise<Hotel> p;
auto switzerland = async([&p] { p.set_value(getHotelSwitzerland()); });
auto usa = async([&p] { p.set_value(getHotelUSA()); });
async([&p] {
  auto hotel = p.get_future().get();
  bookHotel(hotel);
  informFriends(hotel);
}).get();
```

- Hard to understand
- Requires three asynchronous tasks
- No preference of Switzerland

Futures and Promises in C++

C++17

- No callbacks
- No combinators
- No executors
- Only blocking access to results

Futures and Promises in C++

Boost.Thread

- C++17 features
- Callbacks with `then`
- Combinators: `when_any`, `when_all`
- Executors: thread pool etc.
- Non-blocking access to results
- Still missing combinators

Futures and Promises in C++

Extended Example: Holiday Planning in Boost.Thread

```
promise<Hotel> p;
auto switzerland = async([&p] { p.set_value(getHotelSwitzerland()); });
auto usa = async([&p] { p.set_value(getHotelUSA()); });
auto hotel = p.get_future();
hotel.then(bookHotel).then(informFriends).get(); // Callbacks!
```

- Callbacks
- Still hard to understand
- Still no preference of Switzerland

Futures and Promises in C++

Folly

- Boost.Thread features
- More combinators: `collectN`, `collectAny`, `collectAnyWithoutException` etc.
- Default executors
- Supports `Try<T>` for results

Futures and Promises in C++

Extended Example: Holiday Planning in Folly

```
auto switzerland = async(&getHotelSwitzerland);
auto usa = async(&getHotelUSA);
auto hotel = collectAnyWithoutException(move(switzerland), move(usa));
move(hotel).then(bookHotel).then(informFriends);
```

- Combinator collectAnyWithoutException
- Still no preference of Switzerland
- Futures become invalid after calls (move semantics)

Motivation

Issues with Folly

- No systematic design
- No distinction between core (onComplete) and derived features (fallbackTo)
- Missing derived features (fallbackTo)
- Futures become invalid after calls (move semantics)
- Only one callback per future (move semantics)

Our Work

Core Language

- Systematic design of futures and promises
- Distinction between core and derived features
- See our papers "Futures and Promises in Haskell and Scala" and "Advanced Futures and Promises in C++"

C++ Implementation of the Core Language

- Implementation based on MVar (container with one element)
- Provides missing derived features such as fallbackTo
- Futures stay valid after calls (no move semantics)

Core Language

Core Features

$p = \text{new}$	Create
get p	Blocking access
onComplete p h	Non-blocking callback
tryComplete p v	Try to set value

- Systematic design of futures and promises
- Unify futures and promises
- Allow different implementations (STM, MVar and CAS)
- Derived features (fallbackTo) are available for all implementations

Core Language Semantics

Promise State

Either a value (p_v) or a list of callbacks (p_{hs})

new

Creates a new promise with an empty list of callbacks (simplified rewrite rules):

$$p = \text{new} \Rightarrow p_{[]}$$

Core Language Semantics

get

Blocks until there is a result and returns it:

- Returns the result immediately:

$$r = \text{get } p_v \Rightarrow r = v$$

- Blocks until there is a result:

$$r = \text{get } p_{hs} \Rightarrow \text{wait for } p_v$$

Core Language Semantics

onComplete

Registers a callback:

- Executes the callback concurrently if there is a result:

$$\text{onComplete } p_v \ h \Rightarrow [h(v)]$$

- Adds the callback if there is no result:

$$\text{onComplete } p_{hs} \ h \Rightarrow p_{h:hs}$$

Core Language Semantics

tryComplete

Sets the result only once:

- No effect if there is already a result:

$$\text{tryComplete } p_v \ v' \Rightarrow p_v$$

- Sets the result and executes all callbacks concurrently:

$$\text{tryComplete } p_{hs} \ v \Rightarrow p_v \mid [h_1(v)], \ldots, [h_n(v)]$$

- Returns whether successful or not

Core Language Semantics

Summary

- Core language for futures and promises
- Changing the promise state must be thread-safe
- Possible implementations: Software Transactional Memory, MVar, atomic compare-and-swap
- Reference implementations in Scala in Haskell (our short paper "Futures and Promises in Haskell and Scala")

Open Question

How to implement the Core Language in C++?

Core Language in C++

Abstract Class Executor

```
class Executor {
  public:
    virtual void add(std::function<void()> &&f) = 0;
};

class FollyExecutor : public Executor {
  public:
    explicit FollyExecutor(folly::Executor *ex);
    ...
};
```

- Specifies which threads execute callbacks
- Can adapt Folly's and Boost's executors

Core Language in C++

Class Template Try

```
template <typename T>
class Try {
  public:
  explicit Try(T &&v);
  explicit Try(std::exception_ptr &&e);
  const T& get() const;
  bool hasValue() const;
  bool hasException() const;
};
```

- Stores a future's result
- Similar to Folly's and Scala's type

Core Language in C++

Class Template Core

```
template <typename T>
class Core {
  public:
    explicit Core(Executor *executor);
    virtual const Try<T>& get() = 0;
    virtual void onComplete(std::function<void(const Try<T>&)> &&h) =
        0;
    virtual bool tryComplete(Try<T> &&v) = 0;
};
```

- Allows different implementations
- Referred by shared pointers to allow copying

Core Language in C++

Derived Features

```
template <typename T>
class Future {
   ...
};
template <typename T>
class Promise {
   ...
};
```

- Future and Promise hide a Core implementation
- Use shared pointers to allow copying
- Do not depend on the concrete implementation

Core Language in C++

Summary

- Helper classes Executor and Try
- Implement the class template Core
- Get all derived features for free

Open Question

How to implement Core?

Core Implementation with MVar

MVar in C++

```
template<typename T>
class MVar {
  void put(T &&v);
  T take();
  const T &read();
  bool isEmpty();
};
```

- Inspired by Haskell
- Holds one element
- Allows concurrent access
- Implemented with mutex and condition variables

Core Implementation with MVar

Promise State with MVar

```
using Callback = std::function<void(const Try<T> &)>;
using Callbacks = std::vector<Callback>;
using State = std::variant<Try<T>, Callbacks>;

MVar<State> state;
```

- Either a result value or a list of callbacks
- MVar allows concurrent acess

Core Implementation with MVar

new

```
MVar<State>> state(Callbacks());
MVar<void> signal;
```

- New promise with an empty list of callbacks
- Signal for get

Creates a new promise with an empty list of callbacks:

$$p = \text{new} \Rightarrow p_{[]}$$

Core Implementation with MVar

get

```
const Try<T>& get() {
  signal.read();
  auto s = state->read();
  auto r =
      std::get<Try<T>>(std::move(s));
  return r;
}
```

- signal gets its value in tryComplete
- signal.read() blocks the calling thread

Blocks until there is a result and returns it:

- Returns the result immediately:

$$r = \text{get } p_v \Rightarrow r = v$$

- Blocks until there is a result:

$$r = \text{get } p_{hs} \Rightarrow \text{wait for } p_v$$

Core Implementation with MVar

onComplete

```
void onComplete(Callback &&h) {
  auto s = state->take();
  if (s.index() == 0) {
    state->put(std::move(s));
    executeCallback(std::move(h));
  } else {
    auto hs =
        addCallback(std::move(s),
        std::move(h));
    state->put(std::move(hs));
  }
}
```

Registers a callback:

- Executes the callback concurrently if there is a result:

$$\text{onComplete } p_v \ h \Rightarrow [h(v)]$$

- Adds the callback if there is no result:

$$\text{onComplete } p_{hs} \ h \Rightarrow p_{h:hs}$$

Core Implementation with MVar

tryComplete

```
bool tryComplete(Try<T> &&v) {
  auto s = state->take();
  if (s.index() == 0) {
    state->put(std::move(s));
    return false;
  } else {
    auto hs =
        std::get<Callbacks>(s);
    state->put(std::move(v));
    signal.put();
    executeCallbacks(std::move(hs));
    return true;
  }
}
```

Sets the result only once:

- No effect:

$$\text{tryComplete } p_v \; v' \Rightarrow p_v$$

- Sets the result:

$$\text{tryComplete } p_{hs} \; v \Rightarrow p_v \mid [h_1(v)], \ldots, [h_n(v)]$$

Advanced Futures and Promises

Features

- Systematic design
- Class Core allows different implementations (MVar, STM, CAS, . . .)
- Derived features for free
- Futures and promises do not become invalid (no move semantics)
- Multiple callbacks per future
- Multiple-read semantics for futures

Advanced Futures and Promises

Extended Example: Holiday Planning

```
FollyExecutor ex(getCPUExecutor().get());
auto switzerland = async(&ex, getHotelSwitzerland);
auto usa = async(&ex, getHotelUSA);
auto hotel = switzerland.fallbackTo(usa);
hotel.onComplete(bookHotelAdv);
hotel.onComplete(informFriendsAdv);
```

- Prefers Switzerland
- Multiple callbacks per future
- Futures do not become invalid

Advanced Futures and Promises

Derived Features

async ex f
then p f
thenWith p f
guard p f
fallbackTo p q
first p q
firstSucc p q

firstN $[p_1, \ldots, p_m]$ n
firstNSucc $[p_1, \ldots, p_m]$ n
trySuccess p v
tryFail p e
tryCompleteWith p q
trySuccessWith p q
tryFailWith p q

Even more powerful than Scala's library

Sharing Futures and Promises

No Automatic Garbage Collection

- Manual memory management
- Distinction between shared and non-shared futures and promises
- Move semantics for non-shared futures and promises
- Copy semantics for shared futures and promises

Move Semantics

- Prevent copying
- Invalidate the source object

Sharing Futures and Promises

tryCompleteWith

tryCompleteWith p q =
onComplete q (v => tryComplete p v)

- Derived feature
- Completes p with the result q
- Non-blocking

Sharing Futures and Promises

tryCompleteWith for Folly Implemented Wrong!

```
template <typename T>
void tryCompleteWith(Promise<T> &p, Future<T> &f) {
  f.setCallback_([&p](Try<T> &&t) {
    p.setTry(move(t));
  });
  return p;
}
```

- No guarantees for the lifetime of p
- Only one callback per future
- f becomes invalid

Sharing Futures and Promises

tryCompleteWith for Folly Implemented Safe

```
template <typename T>
void tryCompleteWith(Promise<T> &&p, Future<T> &&f) {
  auto promise = make_shared<Promise<T>>(move(p));
  auto future = make_shared<Future<T>>(move(f));
  future->setCallback_([promise, future](Try<T> &&t) {
    promise->setTry(move(t));
  });
}
```

- Moves promise and future
- Promise and future cannot be used anymore
- Guarantees the lifetime

Sharing Futures and Promises

tryCompleteWith for Advanced Futures and Promises

```
template<typename T>
void tryCompleteWith(Promise<T> p, Future<T> f) {
  f.onComplete([p](const Try<T> &t) mutable {
    p.tryComplete(t);
  });
}
```

- Copies promise and future
- Promise and future can still be used
- Internal shared pointer to Core
- Guarantees the lifetime

Sharing Futures and Promises

C++17 and Boost.Thread

```
future<int> f = async([] () { ... });
shared_future<int> = f.share();
```

No shared promises

Folly

- No shared futures or promises
- Workaround with the help of SharedPromise<T>
- See our paper "Advanced Futures and Promises in C++"

Sharing Futures and Promises

Broken Promises

```
promise<Hotel> p;
future<Hotel> f = p.get_future();
async([p = move(p)] { ... });
...
f.get();
```

- Promise is never completed
- Futures are completed with special exception BrokenPromise
- Same behavior as C++17, Boost.Thread and Folly
- We must keep track of referencing promises

Sharing Futures and Promises

Summary

- Move semantics restrict derived features
- Copying removes restrictions
- C++ provides shared pointers
- Consider broken promises

Upcoming Features

async/await

```
val switzerland = Future { ... }
val usa = Future { ... }
val hotel = async { bookHotels(await(switzerland), await(usa)) }
```

- Syntactic sugar for composition
- Scala, C#, JavaScript etc. support
- Microsoft provides C++ support
- Same as:

```
switzerland.flatMap(s => usa.map(u => bookHotels(s, u)))
```

Upcoming Features

C++20 Proposals

- Add executors
- Separation of SemiFuture and ContinuableFuture
- Only some combinators like onSuccess and onFailure
- async/await only for Coroutines
- Track `https://github.com/executors`
- Track `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/`

Future Work

Implementations of the Core Language

- CAS (std::atomic)
- STM (C++ library)
- ...

Program Transformations

Promise Linking

Benchmarks

Compare to Folly

References

Papers

- Advanced Futures and Promises in C++
- Futures and Promises in Haskell and Scala

GitHub

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