From Eager Futures/Promises to Lazy Continuations

Evolving an Actor Library Based on Lessons Learned from Large-Scale Deployments

Benjamin Hindman
@benh

CppCon 2021

prologue

- past life at UC Berkeley, Twitter, Mesosphere/D2iQ
- currently research at UC Berkeley and reboot.dev
- big thanks to Nikita (@FolMing), Zakhar (@onelxj), and Artur (@ArthurBandaryk)

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

chapters

key challenges we’ve faced
building distributed systems with good
performance and correctness

key challenges we’ve faced
building distributed systems with good
performance and correctness

challenges building systems

challenges building systems

(1) you have to wait

(2) you have state

challenges building systems

(1) you have to wait

(2) you have state

motivating example

```
std::string text = "...";
text = SpellCheck(text);
text = GrammarCheck(text);
```

function composition is fundamental

```
GrammarCheck(SpellCheck("..."))
```

motivating example

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

motivating example

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

motivating example

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

you have to wait!

whether you end up writing code that is *blocking* or *non-blocking* doesn't change the underlying issue … you have to wait!

possible solutions

possible solutions

just wait …

possible solutions

~~just wait …~~

possible solutions

~~just wait …~~

use threads …

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

use coroutines …

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

use coroutines … *circa 2009*

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

use coroutines … *circa 2009*

use a different language (e.g., Erlang) …

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

use coroutines … *circa 2009*

use a different language (e.g., Erlang) … *or bring Erlang to C++!*

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

use coroutines … *circa 2009*

use a different language (e.g., Erlang) … *or bring Erlang to C++!*

use callbacks …

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

use coroutines … *circa 2009*

use a different language (e.g., Erlang) … *or bring Erlang to C++!*

use callbacks … *more on this later*

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

use coroutines … *circa 2009*

use a different language (e.g., Erlang) … *or bring Erlang to C++!*

use callbacks … *more on this later*

use futures/promises …

possible solutions

~~just wait …~~

use threads … *too expensive, not conducive to correctness*

use coroutines … *circa 2009*

use a different language (e.g., Erlang) … *or bring Erlang to C++!*

use callbacks … *more on this later*

use futures/promises … ✅

futures/promises: “buffered channel”

```
Channel<std::string> channel;
```

futures/promises: “buffered channel”

```
Channel<std::string> channel;
```

thread x

```
Channel::Reader<std::string> reader = channel.Reader();
```

futures/promises: “buffered channel”

```
Channel<std::string> channel;
```

thread x

```
Channel::Reader<std::string> reader = channel.Reader();
reader.Read(); // Blocks!
```

futures/promises: “buffered channel”

```
Channel<std::string> channel;
```

thread x

```
Channel::Reader<std::string> reader = channel.Reader();
reader.Read(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
```

futures/promises: “buffered channel”

```
Channel<std::string> channel;
```

thread x

```
Channel::Reader<std::string> reader = channel.Reader();
reader.Read(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
```

futures/promises: “buffered channel”

```
Channel<std::string> channel;
```

thread x

```
Channel::Reader<std::string> reader = channel.Reader();
reader.Read(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Channel<std::string> channel;
```

thread x

```
Channel::Reader<std::string> reader = channel.Reader();
reader.Read(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Channel::Reader<std::string> reader = channel.Reader();
reader.Read(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Channel::Reader<std::string> reader = channel.Reader();
reader.Read();  // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Future<std::string> future = promise.Future();
reader.Read(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Future<std::string> future = promise.Future();
reader.Read(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Future<std::string> future = promise.Future();
future.Get(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Future<std::string> future = promise.Future();
future.Get(); // Blocks!
```

thread y

```
Channel::Writer<std::string> writer = channel.Writer();
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Future<std::string> future = promise.Future();
future.Get(); // Blocks!
```

thread y

```
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Future<std::string> future = promise.Future();
future.Get(); // Blocks!
```

thread y

```
writer.Write("..."); // Non-blocking!
writer.Close();
```

futures/promises: “buffered channel”

```
Promise<std::string> promise;
```

thread x

```
Future<std::string> future = promise.Future();
future.Get(); // Blocks!
```

thread y

```
promise.Set("..."); // Non-blocking!
```

futures/promises

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

futures/promises

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com" , body);
  return response.body;
}
```

futures/promises

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com" , body);
  return response.body;
}
```

futures/promises

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com" , body);
  return response.body;
}
```

futures/promises

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  auto response = http::Post("https://www.online-spellcheck.com" , body);
  return future;
}
```

futures/promises

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return future;
}
```

futures/promises

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

futures/promises

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

futures/promises failures

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [promise = std::move(promise)](auto&& response) {
        if (response.code != 200) promise.Fail(response.code);
        else promise.Set(response.body);
      });
  return future;
}
```

futures/promises and function composition

```
std::string text = "...";
text = SpellCheck(text);
text = GrammarCheck(text);
```

futures/promises and function composition

```
std::string text = "...";
text = SpellCheck(text).Get();
text = GrammarCheck(text).Get();
```

futures/promises and function composition

```
std::string text = "...";
text = SpellCheck(text).Get(); // Blocks!
text = GrammarCheck(text).Get(); // Blocks!
```

futures/promises and function composition

```
std::string SpellAndGrammarCheck(std::string text) {
  text = SpellCheck(text).Get(); // Blocks!
  return GrammarCheck(text).Get(); // Blocks!
}
```

futures/promises and function composition

```
std::string SpellAndGrammarCheck(std::string text) {
  text = SpellCheck(text).Get();  // Blocks!
  return GrammarCheck(text).Get();  // Blocks!
}
```

futures/promises and function composition

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  text = SpellCheck(text).Get();  // Blocks!
  return GrammarCheck(text).Get();  // Blocks!
}
```

futures/promises and function composition

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  text = SpellCheck(text).Get(); // Blocks!
  return GrammarCheck(text).Get();  // Blocks!
}
```

futures/promises and function composition

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
  return GrammarCheck(text).Get();  // Blocks!
}
```

futures/promises and function composition

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
  return GrammarCheck(text).Get(); // Blocks!
}
```

futures/promises and function composition

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
        .Then([](auto&& text) {
          return GrammarCheck(text);
        });
}
```

futures/promises and function composition

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

futures/promises and function composition

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      .Then([](auto&& text) {
        return GrammarCheck(text); // Can be a Future<T> or T.
      });
}
```

challenges building systems

(1) **you have to wait** ✅

(2) you have state

challenges building systems

(1) you have to wait ✅

(2) **you have state**

executing code with futures/promises

- only need a single thread to execute code with futures/promises because you never block (i.e., a single threaded “event loop”)

executing code with futures/promises

- only need a single thread to execute code with futures/promises because you never block (i.e., a single threaded “event loop”)
- however, code may not be executed atomically because it may be interleaved with other code when it has to wait (executing other code while you have to wait is the whole point of all this!)

executing code with futures/promises

- only need a single thread to execute code with futures/promises because you never block (i.e., a single threaded “event loop”)
- however, code may not be executed atomically because it may be interleaved with other code when it has to wait (executing other code while you have to wait is the whole point of all this!)
- many call this “concurrency” vs parallelism because it only gives you the illusion of parallelism since you’re not running anything simultaneously (i.e., on multiple CPUs)

executing code with futures/promises

- only need a single thread to execute code with futures/promises because you never block (i.e., a single threaded “event loop”)
- however, code may not be executed atomically because it may be interleaved with other code when it has to wait (executing other code while you have to wait is the whole point of all this!)
- many call this “concurrency” vs parallelism because it only gives you the illusion of parallelism since you’re not running anything simultaneously (i.e., on multiple CPUs)
- but you still have all the synchronization problems from parallelism!

executing code with futures/promises

- only need a single thread to execute code with futures/promises because you never block (i.e., a single threaded “event loop”)
- however, code may not be executed atomically because it may be interleaved with other code when it has to wait (executing other code while you have to wait is the whole point of all this!)
- many call this “concurrency” vs parallelism because it only gives you the illusion of parallelism since you’re not running anything simultaneously (i.e., on multiple CPUs)
- but you still have all the synchronization problems from parallelism!
- can execute in parallel by using a thread pool instead of a single thread

executing code with futures/promises

- only need a single thread to execute code with futures/promises because you never block (i.e., a single threaded "event loop")
- however, code may not be executed atomically because it may be interleaved with other code when it has to wait (executing other code while you have to wait is the whole point of all this!)
- many call this "concurrency" vs parallelism because it only gives you the illusion of parallelism since you're not running anything simultaneously (i.e., on multiple CPUs)
- but **you** still **have** all the **synchronization problems** from parallelism!
- can execute in parallel by using a thread pool instead of a single thread

possible solutions

1963: mutexes, semaphores

1973: actors

1974: monitors

1978: communicating sequential processes (CSP)

1987: statecharts

possible solutions *with threads*

1963: mutexes, semaphores

1973: actors

1974: monitors

1978: communicating sequential processes (CSP)

1987: statecharts

possible solutions *without threads*

1963: mutexes, semaphores

1973: actors

1974: monitors

1978: communicating sequential processes (CSP)

1987: statecharts

without threads?

abstractions without threads *encapsulate* the execution model/semantics and synchronization of state actors *encapsulate*

execution, synchronization, state

more encapsulation ⇒ higher-level abstraction

- (usually) easier to reason about
- (usually) easier to run on more hardware/platforms
- (usually) easier to optimize

actors

- local *mutable* state
- queue of incoming “messages”
- receive and handle “messages” one at a time
- sending “messages” to other actors is non-blocking (no waiting!)
- same programming model whether local or distributed

“actors” in C++

many actor libraries are based on low-level message-passing “send/receive”

```
struct MyActor : public Actor {
  void Receive(ActorId sender, Message message, void* arguments) override {
    switch (message) {
      case MESSAGE_FOO_REQUEST:
        auto* request = (FooRequest*) arguments;
        ...
        Send(sender, MESSAGE_FOO_RESPONSE, response);
        break;
      case MESSAGE_BAR_REQUEST:
      ...
    }
  }
};
```

“actors” in C++

many actor libraries are based on low-level message-passing “send/receive”

```
struct MyActor : public Actor {
  void Receive(ActorId sender, Message message, void* arguments) override {
    switch (message) {
      case MESSAGE_FOO_REQUEST:
        auto* request = (FooRequest*) arguments;
        ...
        Send(sender, MESSAGE_FOO_RESPONSE, response);
        break;
      case MESSAGE_BAR_REQUEST:
      ...
    }
  }
};
```

“actors” in C++

many actor libraries are based on low-level message-passing “send/receive”

```
struct MyActor : public Actor {
  void Receive(ActorId sender, Message message, void* arguments) override {
    switch (message) {
      case MESSAGE_FOO_REQUEST:
        auto* request = (FooRequest*) arguments;
        ...
        Send(sender, MESSAGE_FOO_RESPONSE, response);
        break;
      case MESSAGE_BAR_REQUEST:
      ...
    }
  }
};
```

“actors” in C++

many actor libraries are based on low-level message-passing “send/receive”

```
struct MyActor : public Actor {
  void Receive(ActorId sender, Message message, void* arguments) override {
    switch (message) {
      case MESSAGE_FOO_REQUEST:
        auto* request = (FooRequest*) arguments;
        ...
        Send(sender, MESSAGE_FOO_RESPONSE, response);
        break;
      case MESSAGE_BAR_REQUEST:
      ...
    }
  }
};
```

“actors” in C++

many actor libraries are based on low-level message-passing “send/receive”

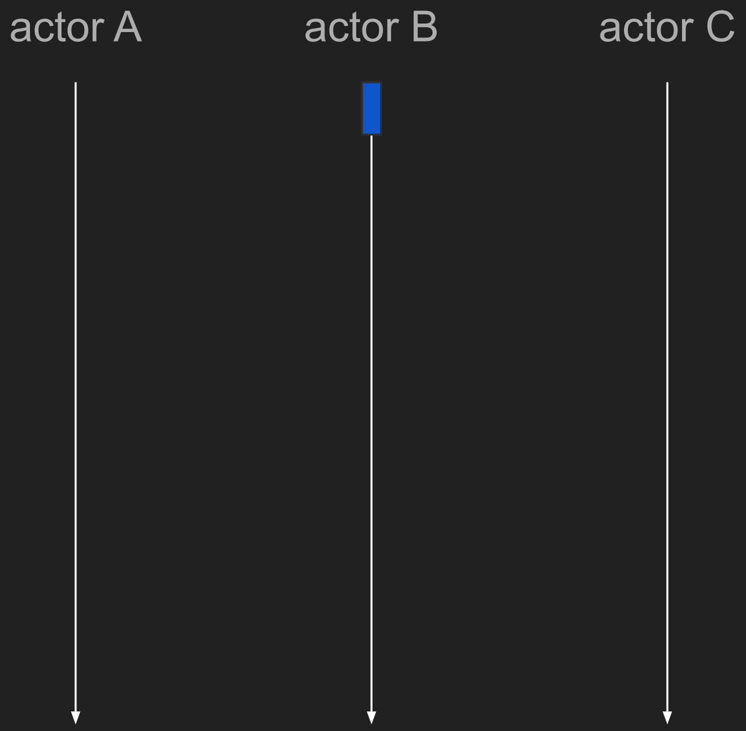
```
struct MyActor : public Actor {
  void Receive(ActorId sender, Message message, void* arguments) override {
    switch (message) {
      case MESSAGE_FOO_REQUEST:
        auto* request = (FooRequest*) arguments;
        ...
        Send(sender, MESSAGE_FOO_RESPONSE, response);
        break;
      case MESSAGE_BAR_REQUEST:
      ...
    }
  }
};
```

“actors” in C++

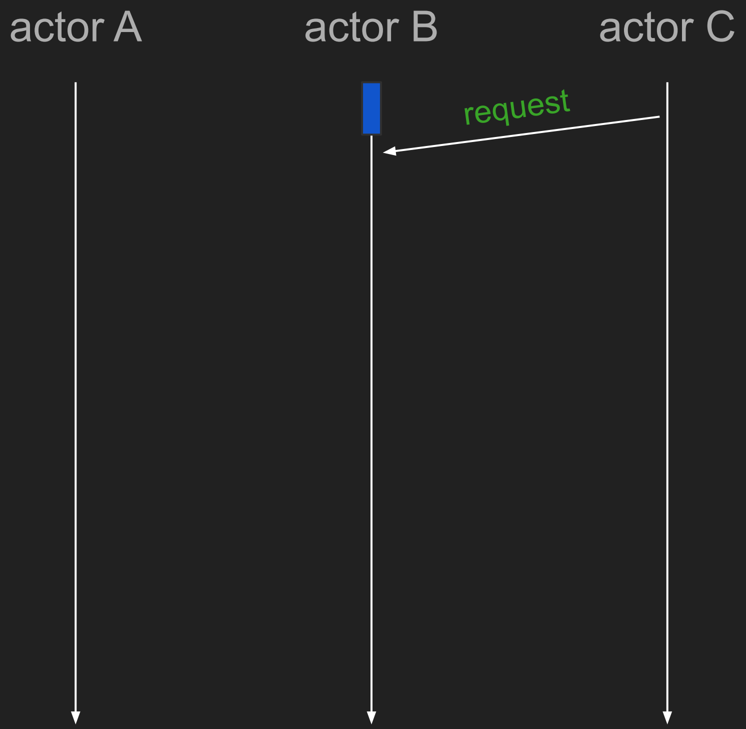
many actor libraries are based on low-level message-passing “send/receive”

```
struct MyActor : public Actor {
  void Receive(ActorId sender, Message message, void* arguments) override {
    switch (message) {
      case MESSAGE_FOO_REQUEST:
        auto* request = (FooRequest*) arguments;
        ...
        Send(sender, MESSAGE_FOO_RESPONSE, response);
        break;
      case MESSAGE_BAR_REQUEST:
      ...
    }
  }
};
```

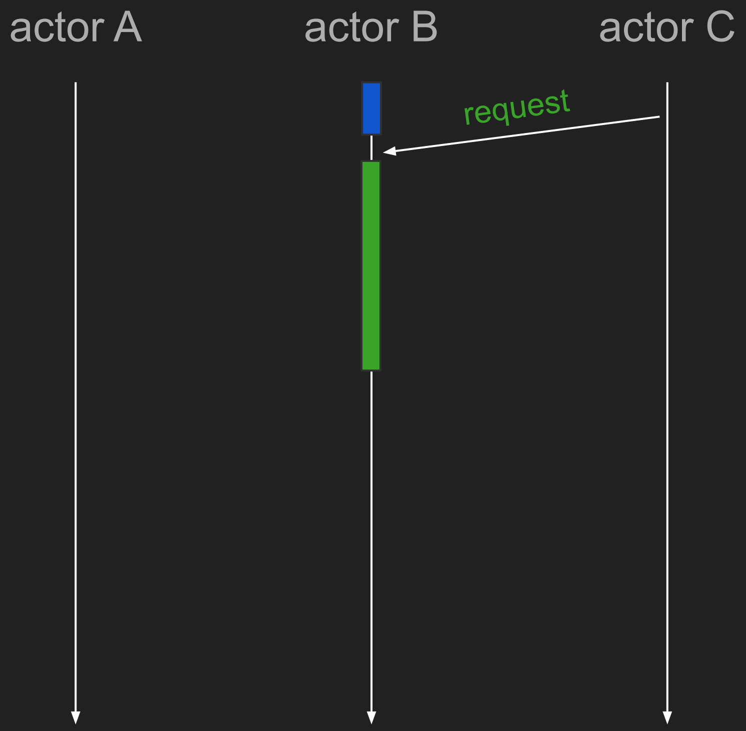
actors (visualized)



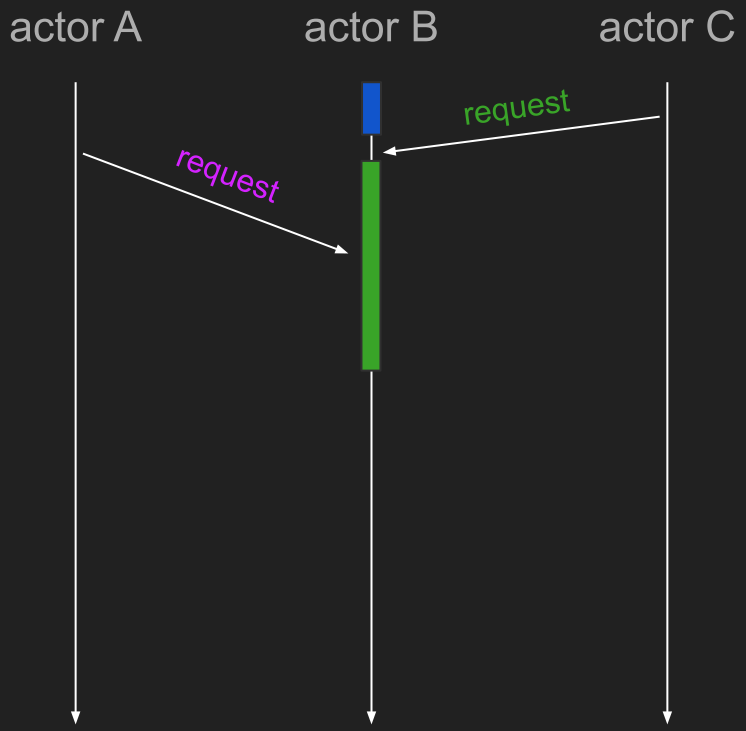
actors (visualized)



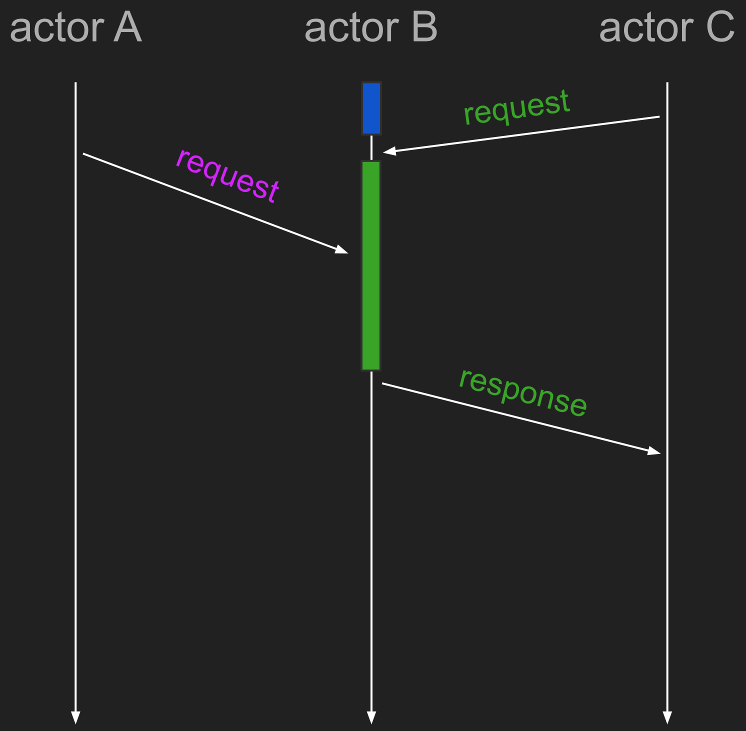
actors (visualized)



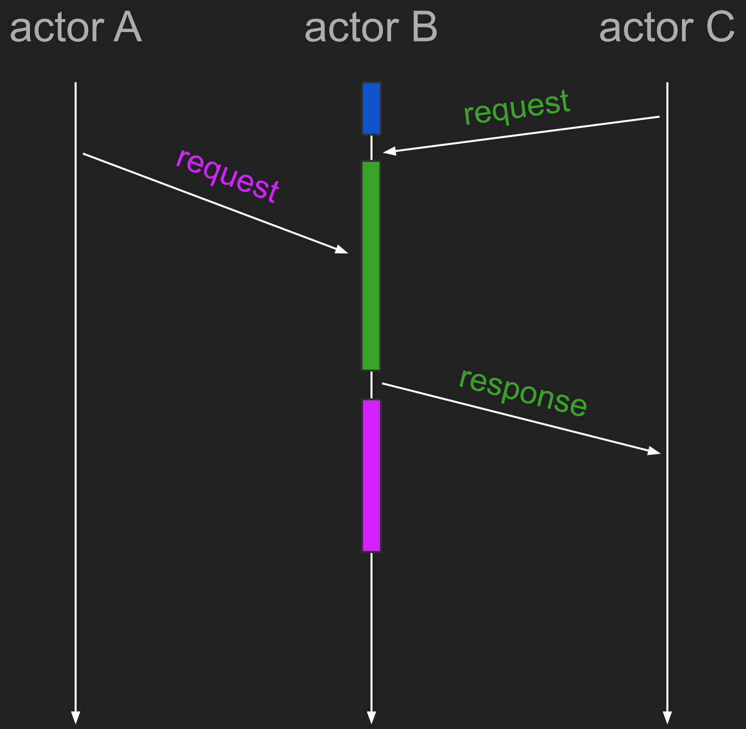
actors (visualized)



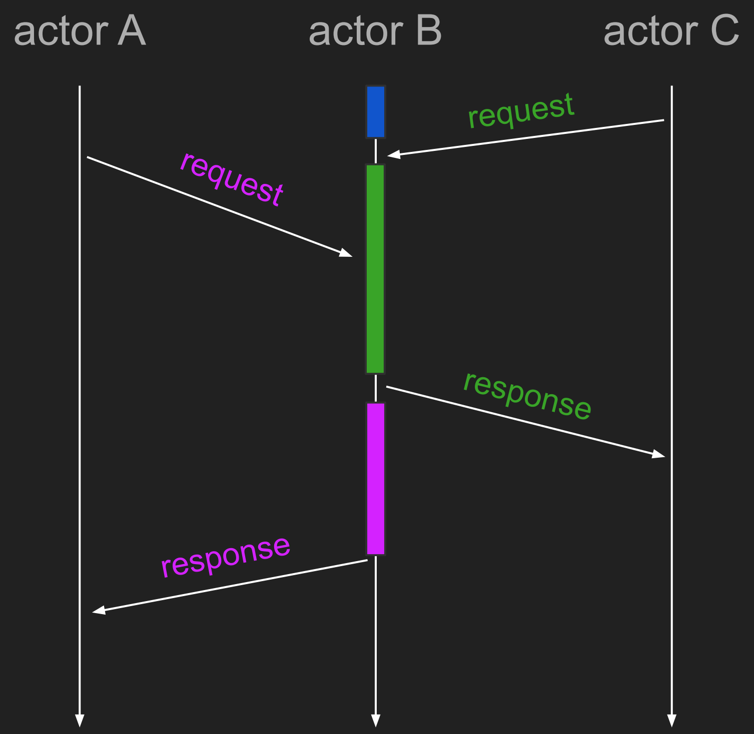
actors (visualized)



actors (visualized)



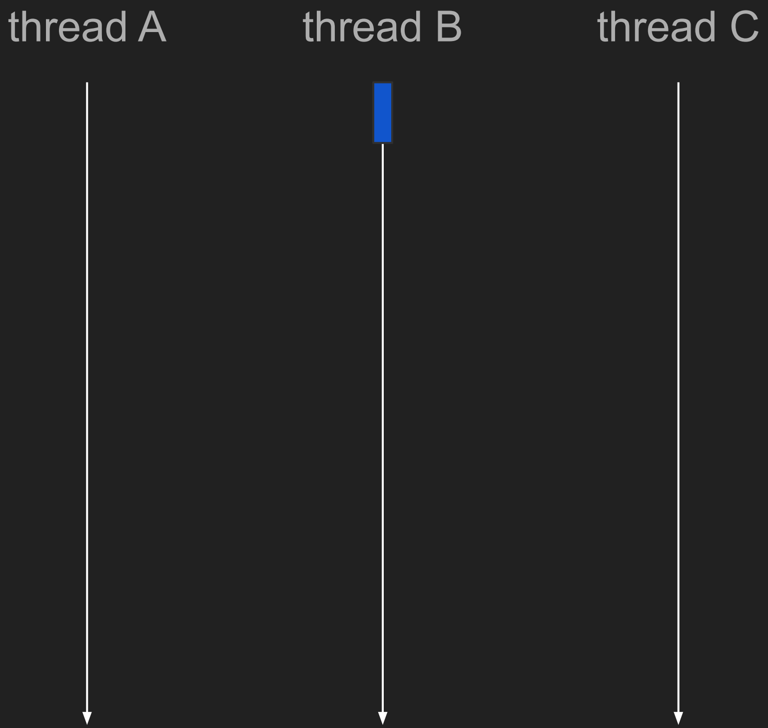
actors (visualized)



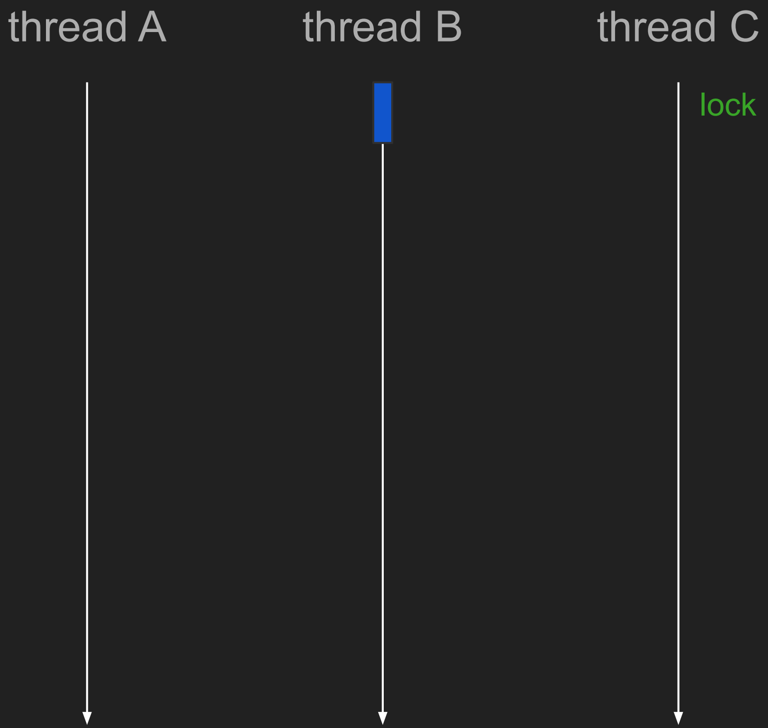
can’t I get the same thing using locks for all my methods? (or mark all my methods in Java as `synchronized`)

NO!

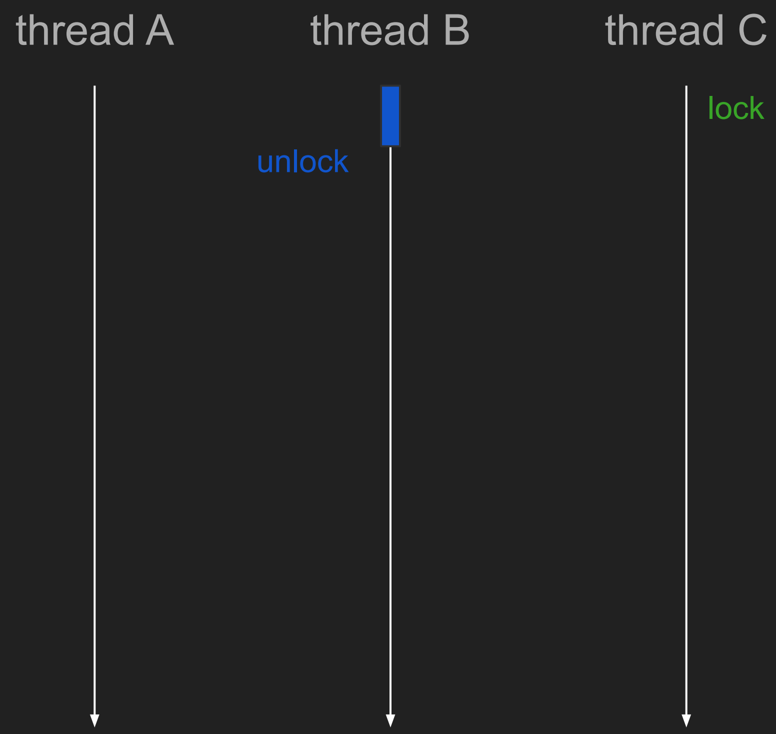
threads (visualized)



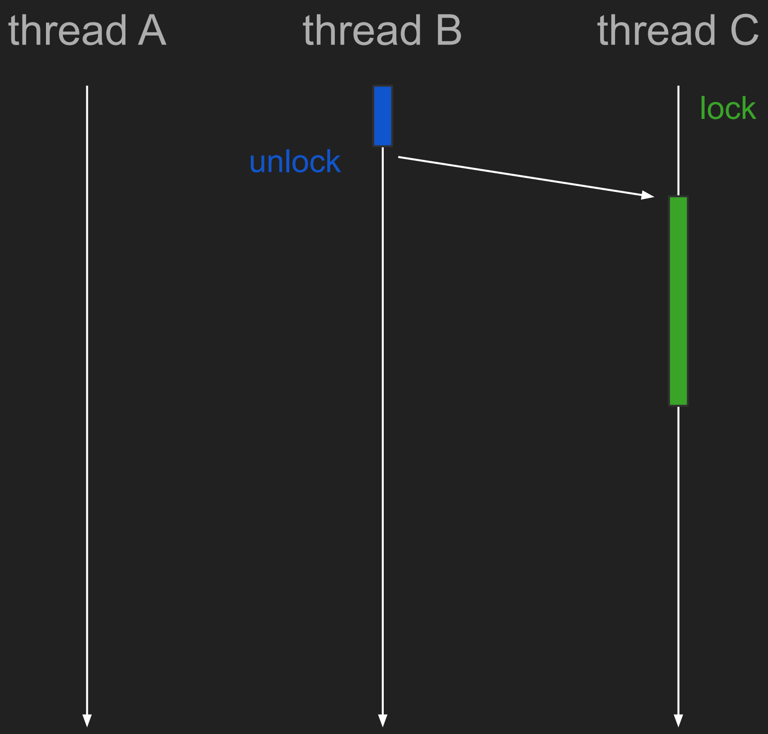
threads (visualized)



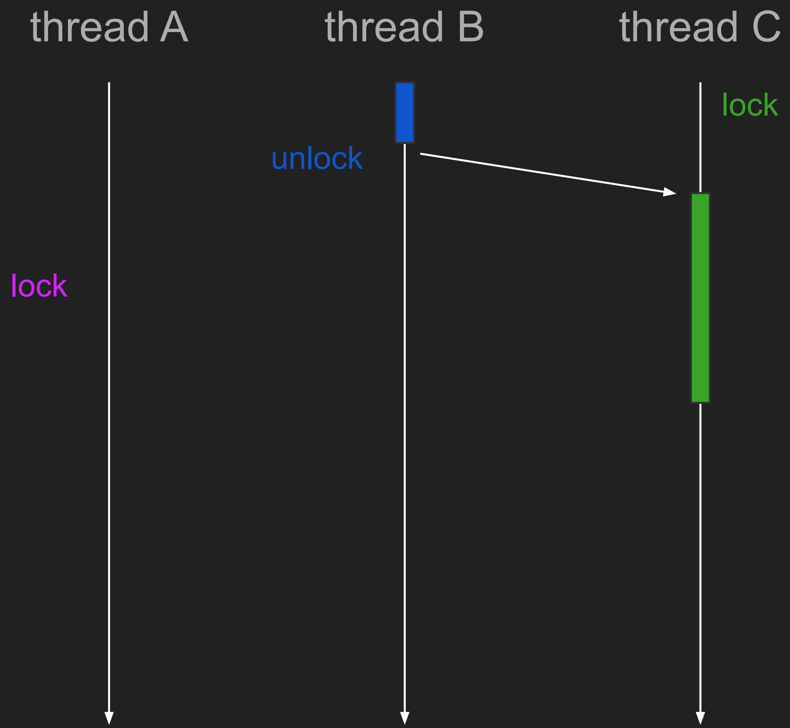
threads (visualized)



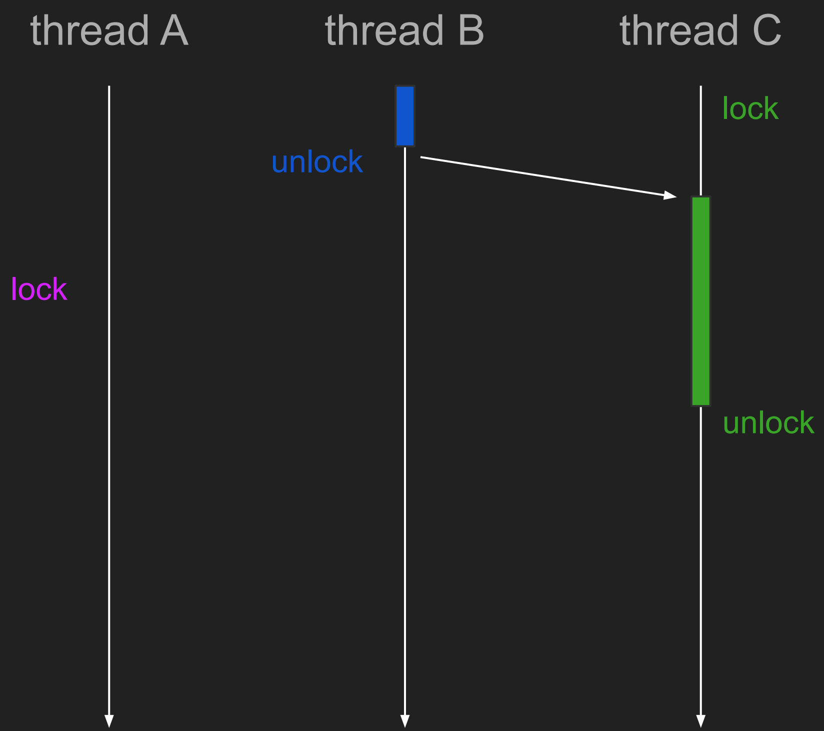
threads (visualized)



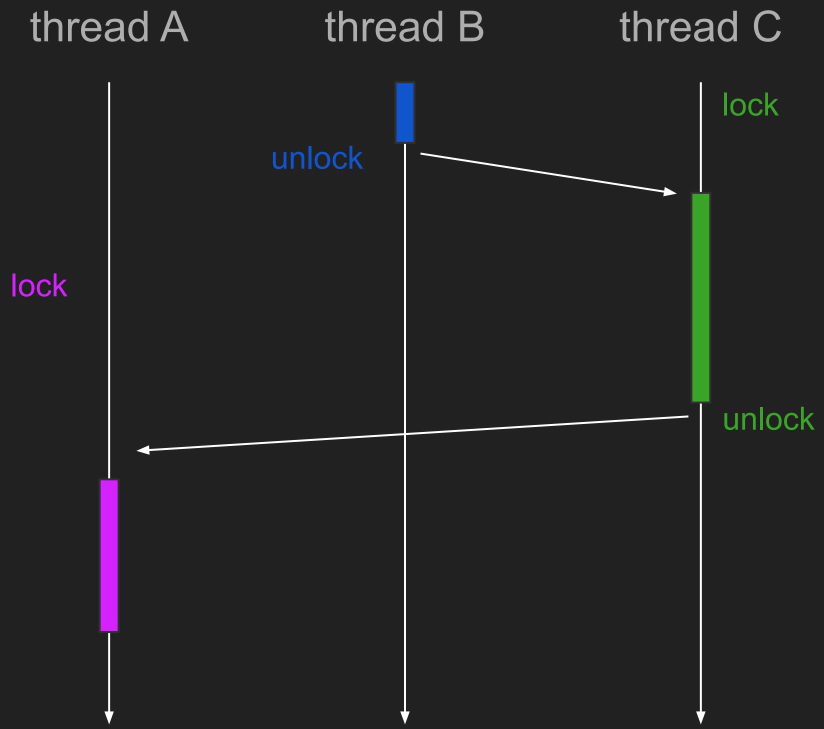
threads (visualized)



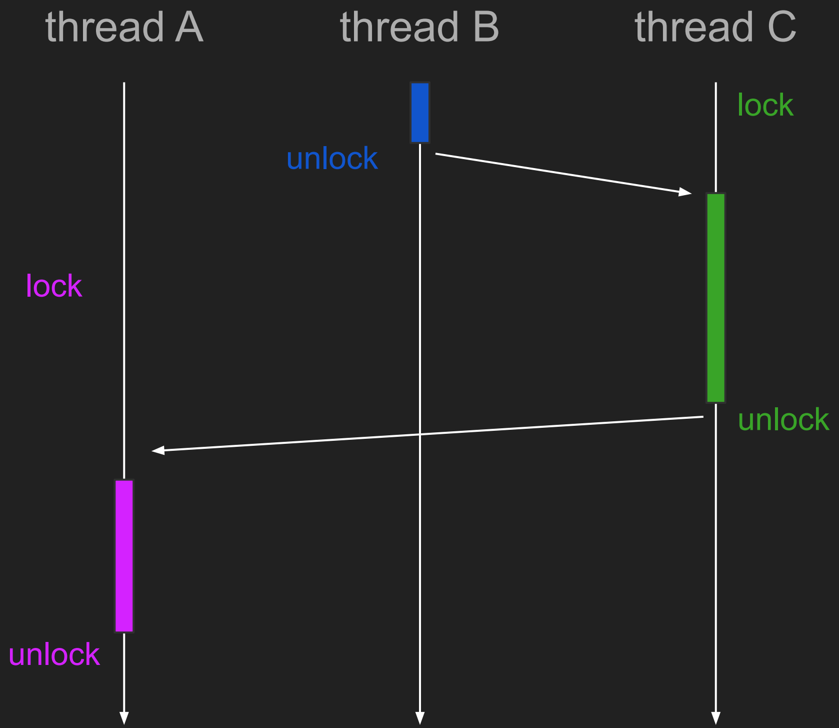
threads (visualized)



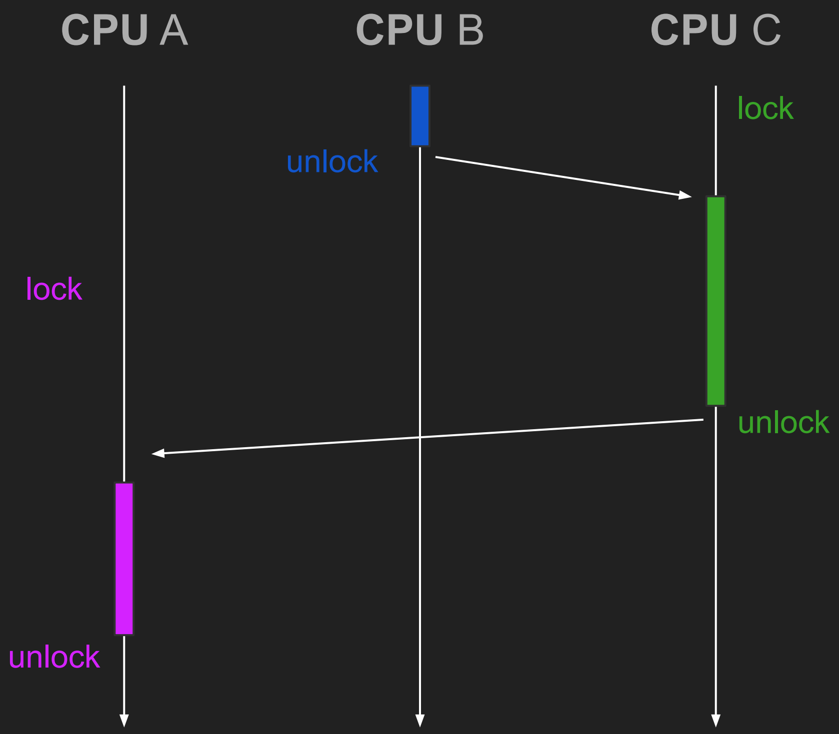
threads (visualized)



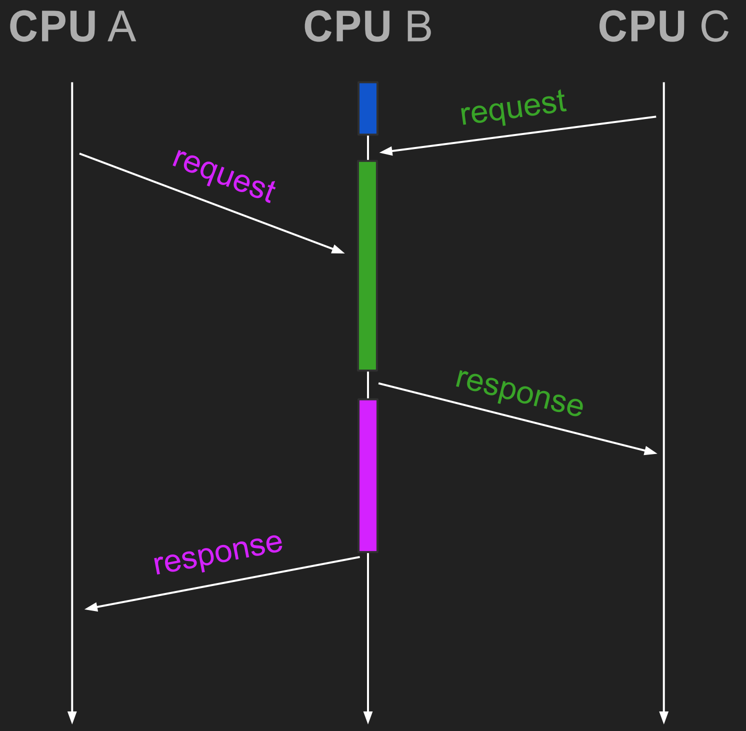
threads (visualized)



threads (visualized)



actors (visualized)



actor performance

for parallel programs data will often need to be shared and/or moved between execution resources (i.e., cores)

but for distributed/network services, the data is often only shared with *other machines* and bouncing the data unnecessarily between arbitrary cores incurs performance slowdowns (and in many cases due to cache line sharing the slowdown impacts everyone)

(check out http://seastar.io for more examples of distributed services using actors)

challenges building systems

(1) you have to wait

(2) you have state ✅

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

circa 2009

- building distributed system Apache Mesos at UC Berkeley
- using C++ to avoid runtime non-determinism that had been plaguing the Hadoop distributed system (due to garbage collection, it was written in Java)
- wanted to use actors

so let's build a library for actors in C++ … libprocess

our novelty: let's combine futures/promises + actors!

why actors need futures/promises

“actors” in C++

```
struct MyActor : public Actor {
  void Receive(ActorId sender, Message message, void* arguments) override {
    switch (message) {
      case MESSAGE_FOO_REQUEST:
        auto* request = (FooRequest*) arguments;
        ...
        Send(sender, MESSAGE_FOO_RESPONSE, response);
        break;
      case MESSAGE_BAR_REQUEST:
      ...
    }
  }
};
```

why actors need futures/promises

hard to reason about control flow between actors

- sending/receiving messages is the “assembly language” of the actor model (even though they do solve “having to wait”)
- messages are like ***gotos***!

gotos considered harmful ...

why actors need futures/promises

instead of gotos we want:

- *function calls* (but without blocking so ... return futures!)

```
MyActor actor;

auto future = actor.Foo(...);
```

why actors need futures/promises

instead of gotos we want:

- *function calls* (but without blocking so ... return futures!)
- *function composition* (but without blocking so ... **`Then()`**)

```
MyActor actor;

auto future = actor.Foo(...)
  .Then([](auto&& response) {
    return ...;
  });
```

libprocess actors (pseudocode)

```
struct MyActor : public Actor {
  Future<FooResponse> Foo(FooRequest request) {
    // Execute the “message” on the actor ‘self()’.
    return On(self(), [this, request]() {
      FooResponse response;
      ...
      return response;
    });
  }
};
```

libprocess actors (pseudocode)

```
struct MyActor : public Actor {
  Future<FooResponse> Foo(FooRequest request) {
    // Execute the “message” on the actor ‘self()’.
    return On(self(), [this, request]() {
      FooResponse response;
      ...
      return response;
    });
  }
};
```

libprocess actors (pseudocode)

```
struct MyActor : public Actor {
  Future<FooResponse> Foo(FooRequest request) {
    // Execute the "message" on the actor 'self()'.
    return On(self(), [this, request]() {
      FooResponse response;
      ...
      return response;
    });
  }
};
```

libprocess actors (pseudocode)

```
struct MyActor : public Actor {
  Future<FooResponse> Foo(FooRequest request) {
    // Execute the “message” on the actor ‘self()’.
    return On(self(), [this, request]() {
      FooResponse response;
      ...
      return response;
    });
  }
};
```

libprocess actors (pseudocode)

```
struct MyActor : public Actor {
  Future<FooResponse> Foo(FooRequest request) {
    // Execute the “message” on the actor ‘self()’.
    return On(self(), [this, request]() {
      FooResponse response;
      ...
      return response;
    });
  }
};
```

libprocess actors (pseudocode)

```
struct MyActor : public Actor {
  Future<FooResponse> Foo(FooRequest request) {
    // Execute the “message” on the actor ‘self()’.
    return On(self(), [this, request]() {
      ...
      return SomeOtherFunctionReturningAFuture()
        .Then([](auto&& value) {
          ...
        });
    });
  }
};
```

why futures/promises need actors

why futures/promises need actors

```
struct MyObject {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then([](auto&& value) {
        // Where should this lambda run?????
        ...
      });
  }
};
```

why futures/promises need actors

```
struct MyObject {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then([](auto&& value) {
        // Where should this lambda run?????
        ...
      });
  }
};
```

strawman: using the execution resource that completes the promise associated with the future returned from `SomeFunction()`

why futures/promises need actors

```
struct MyObject {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then([this](auto&& value) {
        // Where should this lambda run?????
        std::unique_lock<std::mutex> lock(mutex_);
        i += value;
      });
  }
 private:
  int i_;
  std::mutex mutex_;
};
```

why futures/promises need actors

```
struct MyObject {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then([this](auto&& value) {
        // Where should this lambda run?????
        std::unique_lock<std::mutex> lock(mutex_);
        i += value;
      });
  }
 private:
  int i_;
  std::mutex mutex_;
};
```

ouch, calling **`promise.Set()`** *might* be blocking!?

hard to reason about due to non-deterministic performance characteristics (kind of like garbage collection, that thing we wanted to avoid)

why futures/promises need actors

```
struct MyObject {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then([this](auto&& value) {
        // Where should this lambda run?????
        return mutex_.Acquire()
          .Then([this]() {
            i += value;
            mutex_.Release();
          });
      });
  }
 private:
  int i_;
  AsyncMutex mutex_;
};
```

asynchronous mutex?

why futures/promises need actors

```
struct MyObject {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then([this](auto&& value) {
        // Where should this lambda run?????
        return mutex_.Acquire()
          .Then([this]() {
            i += value;
            mutex_.Release();
          });
      });
  }
 private:
  int i_;
  AsyncMutex mutex_;
};
```

asynchronous mutex?

ugh, calling **`Release()`** will/must execute any waiters which might not block but could still incur arbitrary non-deterministic execution!

why futures/promises need actors

```
struct MyObject : public Actor {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then([this](auto&& value) {
        return On(self(), [this, value]() {
          i_ += value;
        });
      }));
  }
 private:
  int i_;
};
```

why futures/promises need actors

```
struct MyObject : public Actor {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then(DeferOn(self(), [this](auto&& value) {
        i_ += value;
      }));
  }
 private:
  int i_;
};
```

why futures/promises need actors

```
struct MyObject : public Actor {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then(DeferOn(self(), [this](auto&& value) {
        i_ += value;
      }));
  }
 private:
  int i_;
};
```

- actor provides “executor” to execute the continuation
- setting the promise is fast (non-blocking)
- no need for synchronization!

why futures/promises need actors

```
struct MyObject : public Actor {
  Future<void> SomeMember() {
    return SomeFunction()
      .Then(DeferOn(self(), [this](auto&& value) {
        i_ += value;
      }));
  }
 private:
  int i_;
};
```

- actor provides “executor” to execute the continuation
- setting the promise is fast (non-blocking)
- **no need for synchronization!**

Apache Mesos (built with libprocess)

- over a half million lines of code
- hundreds of contributors
- about a dozen mutexes! (mostly for interfacing with code not written w/ libprocess)
- massive scale (clusters of ~80k physical machines)

chapters

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

revisiting the problem ...

```
std::string SpellAndGrammarCheck(std::string text) {
  text = SpellCheck(text);
  return GrammarCheck(text);
}
```

revisiting the problem ...

```
std::string SpellAndGrammarCheck(std::string text) {
  text = SpellCheck(text);
  return GrammarCheck(text);
}
```

… this code is *sequential*

revisiting the problem ...

```
std::string SpellAndGrammarCheck(std::string text) {
  text = SpellCheck(text);
  return GrammarCheck(text);
}
```

… this code is *sequential*

… definitely not *parallel*

revisiting the problem ...

```
std::string SpellAndGrammarCheck(std::string text) {
  text = SpellCheck(text);
  return GrammarCheck(text);
}
```

… this code is *sequential*

… definitely not *parallel*

… and even if executed *concurrently* has no state to synchronize

revisiting the problem ...

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

acquires locks and dynamically allocates memory!

(acceptable for parallel/concurrent code, but this isn't!)

why do we need locks and dynamic memory?

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

why do we need locks and dynamic memory?

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  ...
  auto future = promise.future();
  http::AsyncPost(
      ...,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });

  return future
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

why do we need locks and dynamic memory?

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  ...
  auto future = promise.future();
  http::AsyncPost( // Non-blocking! Returns immediately!
      ...,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });

  return future
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

why do we need locks and dynamic memory?

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  ...
  auto future = promise.future();
  http::AsyncPost( // Non-blocking! Returns immediately!
      ...,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });

  return future
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

why do we need locks and dynamic memory?

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  ...
  auto future = promise.future();
  http::AsyncPost( // Non-blocking! Returns immediately!
      ...,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });

  return future
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

there is a race!

why do we need locks and dynamic memory?

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  ...
  auto future = promise.future();
  http::AsyncPost( // Non-blocking! Returns immediately!
      ...,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });

  return future
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

there is a race!

promise may be set *at the same time* continuation is composed via **`Then()`**

thus, we need *locks*!

why do we need locks and dynamic memory?

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  ...
  auto future = promise.future();
  http::AsyncPost( // Non-blocking! Returns immediately!
      ...,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });

  return future
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

there is a race!

promise may be set *before* continuation is composed via **`Then()`**

thus, we need *dynamic allocation*!

can we avoid locking?

can we avoid dynamic allocation?

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

callbacks don’t require locks

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

callbacks don’t require locks

```
void SpellCheck(std::string text, std::function<void(std::string)> f) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

callbacks don't require locks

```
void SpellCheck(std::string text, std::function<void(std::string)> f) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

callbacks don’t require locks

```
void SpellCheck(std::string text, std::function<void(std::string)> f) {
  auto body = http::UrlEncode({"text", text});


  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

callbacks don’t require locks

```
void SpellCheck(std::string text, std::function<void(std::string)> f) {
  auto body = http::UrlEncode({"text", text});


  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

callbacks don’t require locks

```
void SpellCheck(std::string text, std::function<void(std::string)> f) {
  auto body = http::UrlEncode({"text", text});


  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [f = std::move(f)](auto&& response) {
        f(response.body);
      });
  return future;
}
```

callbacks don’t require locks

```
void SpellCheck(std::string text, std::function<void(std::string)> f) {
  auto body = http::UrlEncode({"text", text});


  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [f = std::move(f)](auto&& response) {
        f(response.body); // Invoke continuation without locks!
      });
  return future;
}
```

callbacks don’t require locks

```
template <typename K>
void SpellCheck(std::string text, K k) {
  auto body = http::UrlEncode({"text", text});

  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [k = std::move(k)](auto&& response) {
        k(response.body); // Invoke continuation without locks!
      });
  return future;
}
```

callbacks don’t require locks

```
template <typename K>
void SpellCheck(std::string text, K k) {
  auto body = http::UrlEncode({"text", text});
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [k = std::move(k)](auto&& response) {
        k(response.body); // Invoke continuation without locks!
      });
}
```

futures/promises support failures (and cancellation)

```
template <typename K>
void SpellCheck(std::string text, K k) {
  auto body = http::UrlEncode({"text", text});
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [k = std::move(k)](auto&& response) {
        if (response.code != 200) ???
        else k(response.body);
      });
}
```

futures/promises support failures (and cancellation)

```
template <typename Success, typename Failure>
void SpellCheck(std::string text, Success s, Failure f) {
  auto body = http::UrlEncode({"text", text});
  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [s = std::move(s), f = std::move(f)](auto&& response) {
        if (response.code != 200) f(response.code);
        else s(response.body);
      });
}
```

futures/promises support failures (and cancellation)

```
template <typename K>
void SpellCheck(std::string text, K k) {
  auto body = http::UrlEncode({"text", text});
  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [k = std::move(k)](auto&& response) {
        if (response.code != 200) k.Fail(response.code);
        else k.Success(response.body);
      });
}
```

futures/promises support failures (and cancellation)

```
template <typename K>
void SpellCheck(std::string text, K k) {
  auto body = http::UrlEncode({"text", text});
  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [k = std::move(k)](auto&& response) {
        if (response.code != 200) k.Fail(response.code);
        else k.Success(response.body);
      });
}
```

won't be discussing "cancellation/cancelled" in more detail in this talk but check out the repository for implementation can we avoid locking? ✅

can we avoid dynamic allocation?

do we need to dynamically allocate?

```
template <typename K>
void SpellCheck(std::string text, K k) {
  auto body = http::UrlEncode({"text", text});
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [k = std::move(k)](auto&& response) {
        if (response.code != 200) k.Fail(response.code);
        else k.Success(response.body);
      });
}
```

do we need to dynamically allocate?

```
template <typename K>
void SpellCheck(std::string text, K k) {
  auto body = http::UrlEncode({"text", text});
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [k = std::move(k)](auto&& response) {
        if (response.code != 200) k.Fail(response.code);
        else k.Success(response.body);
      });
}
```

need to allocate the continuation somewhere!

```
namespace http {

template <typename K>
void Post(std::string url, std::string body, K k) {
  void* data = new K(std::move(k));
  ...
  http_post(url, body, data, +[](long code, const char* body, void* data) {
    K* k = reinterpret_cast<K*>(data);
    k->Success(http::Response{code, body});
    delete k;
  });
}

} // namespace http
```

need to allocate the continuation somewhere!

```
namespace http {

template <typename K>
void Post(std::string url, std::string body, K k) {
  void* data = new K(std::move(k));
  ...
  http_post(url, body, data, +[]( long code, const char* body, void* data) {
    K* k = reinterpret_cast<K*>(data);
    k->Success(http::Response{code, body});
    delete k;
  });
}

} // namespace http
```

need to allocate the continuation somewhere!

```
namespace http {

template <typename K>
void Post(std::string url, std::string body, K k) {
  void* data = new K(std::move(k));
  ...
  http_post(url, body, data, +[](long code, const char* body, void* data) {
    K* k = reinterpret_cast<K*>(data);
    k->Success(http::Response{code, body});
    delete k;
  });
}

} // namespace http
```

solve all problems with level of indirection ...

💡 could avoid allocation if return continuation K as part of result of function?

solve all problems with level of indirection ...

💡 could avoid allocation if return continuation K as part of result of function?

take a continuation as an argument and *return* a continuation as the result!!!!!!
(that composes/encapsulates the continuation passed in as an argument)

continuations in continuations

```
template <typename K>
void Post(std::string url, std::string body, K k) {
  void* data = new K(std::move(k));
  ...
  http_post(url, body, data, +[](long code, const char* body, void* data) {
    K* k = reinterpret_cast<K*>(data);
    k->Success(http::Response{code, body});
    delete k;
  });
}
```

continuations in continuations

```
template <typename K>
void Post(std::string url, std::string body, K k) {
  void* data = new K(std::move(k));
  ...
  http_post(url, body, data, +[]( long code, const char* body, void* data) {
    K* k = reinterpret_cast<K*>(data);
    k->Success(http::Response{code, body});
    delete k;
  });
}
```

continuations in continuations

```
template <typename K>
void Post(std::string url, std::string body, K k) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
        delete k;
      });
    }
    std::string url, body;
    K k;
  };
}
```

continuations in continuations

```
template <typename K>
void Post(std::string url, std::string body, K k) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
        delete k;
      });
    }
    std::string url, body;
    K k;
  };
}
```

continuations in continuations

```
template <typename K>
void Post(std::string url, std::string body, K k) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
    std::string url, body;
    K k;
  };
}
```

continuations in continuations

```
template <typename K>
void Post(std::string url, std::string body, K k) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
    std::string url, body;
    K k;
  };
}
```

continuations in continuations

```
template <typename K>
auto Post(std::string url, std::string body, K k) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
    std::string url, body;
    K k;
  };
  return Continuation{std::move(url), std::move(body), std::move(k)};
}
```

continuations in continuations

```
template <typename K>
auto Post(std::string url, std::string body, K k) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[](long code, const char* body, void* data) {
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
    std::string url, body;
    K k;
  };
  return Continuation{std::move(url), std::move(body), std::move(k)};
}
```

lazy continuations

```
auto k = http::Post(url, body, /* k' */);
```

- resulting type is the “computational graph”

lazy continuations

```
auto k = http::Post(url, body, /* k' */);

k.Start();
```

- resulting type is the “computational graph”
- the graph is *lazy*, i.e., nothing has started when we get it (tradeoff for dynamic allocation) and must be explicitly started

lazy continuations

```
auto k = http::Post(url, body, /* k' */);

k.Start();
```

- resulting type is the “computational graph”
- the graph is *lazy*, i.e., nothing has started when we get it (tradeoff for dynamic allocation) and must be explicitly started
- the graph can be allocated on the stack, or the heap (but we can do a single heap allocation rather than one for each operation that requires waiting!)

lazy continuations

```
auto k = http::Post(url, body, /* k' */);

k.Start();
```

- resulting type is the “computational graph”
- the graph is *lazy*, i.e., nothing has started when we get it (tradeoff for dynamic allocation) and must be explicitly started
- the graph can be allocated on the stack, or the heap (but we can do a single heap allocation rather than one for each operation that requires waiting!)
- memory must exist until completion!

can we avoid locking? ✅

can we avoid dynamic allocation? ✅

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

lazy continuations

```
auto k = http::Post(url, body, /* k' */);

k.Start();
```

~~lazy continuations~~ *eventuals*

```
auto k = http::Post(url, body, /* k' */);

k.Start();
```

follow along at https://github.com/3rdparty/eventuals

~~lazy continuations~~ *eventuals*

```
auto k = http::Post(url, body, /* k' */);

k.Start();
```

passing around the continuation **`k`** is not very ergonomic!

Eventual<T>

```
template <typename K>
auto Post(std::string url, std::string body, K k) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[](long code, const char* body, void* data) {
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
    std::string url, body;
    K k;
  };
  return Continuation{std::move(url), std::move(body), std::move(k)};
}
```

Eventual<T>

```
template <typename K>
auto Post(std::string url, std::string body, K k) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
    std::string url, body;
    K k;
  };
  return Continuation{std::move(url), std::move(body), std::move(k)};
}
```

Eventual<T>

```
auto Post(std::string url, std::string body) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
    std::string url, body;
    K k;
  };
  return Continuation{std::move(url), std::move(body), std::move(k)};
}
```

Eventual<T>

```
auto Post(std::string url, std::string body) {
  struct Continuation {
    void Start() {
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
    std::string url, body;
    K k;
  };
  return Continuation{std::move(url), std::move(body), std::move(k)};
}
```

Eventual<T>

```
auto Post(std::string url, std::string body) {
  return Eventual<http::Response>([url, body](auto& k) {
      using K = std::decay_t<decltype(k)>;
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }


  });

}
```

Eventual<T>

```
auto Post(std::string url, std::string body) {
  return Eventual<http::Response>([url, body](auto& k) {
      using K = std::decay_t<decltype(k)>;
      void* data = &k;
      http_post(url, body, data, +[](long code, const char* body, void* data) {
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
  });
}
```

Eventual<T>

```
Future<std::string> SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

Eventual<T>

```
auto SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

Eventual<T>

```
auto SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  Promise<std::string> promise;
  auto future = promise.Future();
  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });
  return future;
}
```

Eventual<T>

```
auto SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});


  http::AsyncPost(
      "https://www.online-spellcheck.com",
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });

}
```

Eventual<T>

```
auto SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});


  http::AsyncPost(
      "https://www.online-spellcheck.com" ,
      body,
      [promise = std::move(promise)](auto&& response) {
        promise.Set(response.body);
      });

}
```

Eventual<T>

```
auto SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});


  return http::Post(
      "https://www.online-spellcheck.com",
      body);

}
```

Eventual<T>

```
auto SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  return http::Post(
      "https://www.online-spellcheck.com",
      body);
}
```

Then()

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

Then()

```
Future<std::string> SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
        .Then([](auto&& text) {
          return GrammarCheck(text);
        });
}
```

Then()

```
auto SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

Then()

```
auto SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      .Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

Then()

```
auto SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      | Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

Then()

```
auto SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      | Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

Then()

```
auto SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      | Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

>90% of time sequencing/composing eventuals using combinators like **`Then()`**

<10% of the time using **`Eventual<T>()`** or creating your own eventual type

combinators

- `Let`
- `Conditional`
- `Raise`
- `Catch`
- `Lock`
- `Terminal`
- `Closure`
- `...`

event loop powered by libuv

- `Timer`
- `Signal`
- `DomainNameResolve`
- `http::Get, http::Post, ...`
- `OpenFile, ReadFile, WriteFile, ...`
- `...`

eventuals + actors

```
struct MyActor : public Actor {
  Future<FooResponse> Foo(FooRequest request) {
    // Execute the "message" on the actor 'self()'.
    return On(self(), [this, request]() {
      FooRequest response;
      ...
      return response;
    });
  }
};
```

eventuals + actors

```
struct MyActor : public Actor {
  Future<FooResponse> Foo(FooRequest request) {
    // Execute the “message” on the actor ‘self()’.
    return On(self(), [this, request]() {
      FooRequest response;
      ...
      return response;
    });
  }
};
```

eventuals + actors

```
struct MyActor : public Actor {
  auto Foo(FooRequest request) {
    // Execute the “message” on the actor ‘self()’.
    return On(self(), [this, request]() {
      FooRequest response;
      ...
      return response;
    });
  }
};
```

eventuals + actors

```
struct MyActor : public Actor {
  auto Foo(FooRequest request) {
    // Execute the “message” on the actor ‘self()’.
    return On(self(), [this, request]() {
      FooRequest response;
      ...
      return response;
    });
  }
};
```

eventuals + actors

```
struct MyActor : public Actor {
  auto Foo(FooRequest request) {
    // Execute the "message" on the actor.
    return Schedule([this, request]() {
      FooRequest response;
      ...
      return response;
    });
  }
};
```

eventuals + actors

```
struct MyActor : public Actor {
  auto Foo(FooRequest request) {
    // Execute the “message” on the actor.
    return Schedule([this, request]() {
      FooRequest response;
      ...
      return response;
    });
  }
};
```

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

where should the continuation run?

```
auto Post(std::string url, std::string body) {
  return Eventual<http::Response>([url, body](auto& k) {
      using K = std::decay_t<decltype(k)>;
      void* data = &k;
      http_post(url, body, data, +[](long code, const char* body, void* data) {
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
  });
}
```

where should the continuation run?

```
auto Post(std::string url, std::string body) {
  return Eventual<http::Response>([url, body]( auto& k) {
      using K = std::decay_t<decltype(k)>;
      void* data = &k;
      http_post(url, body, data,  +[](long code, const char* body, void* data) {
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
  });
}
```

where should the continuation run?

```
auto Post(std::string url, std::string body) {
  return Eventual<http::Response>([url, body](auto& k) {
        using K = std::decay_t<decltype(k)>;
        void* data = &k;
        http_post(url, body, data, +[]( long code, const char* body, void* data)
{
          K* k = reinterpret_cast<K*>(data);
          k->Success(http::Response{code, body});
        });
      }
  });
}
```

if continution is getting invoked by an ***event loop***, don’t want to keep using the same execution resource!

where should the continuation run?

```
auto Post(std::string url, std::string body) {
  return Eventual<http::Response>([url, body](auto& k) {
      using K = std::decay_t<decltype(k)>;
      void* data = &k;
      http_post(url, body, data, +[]( long code, const char* body, void* data)
{
        K* k = reinterpret_cast<K*>(data);
        k->Success(http::Response{code, body});
      });
    }
  });
}
```

if continution is getting invoked by an ***actor***, don’t want to keep using the same execution resource!

where should the continuation run?

motivating example

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

function abstraction

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

function abstraction

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

function abstraction enables us to

separate the concerns of

interface and implementation

function abstraction

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

function abstraction allows us to not need to care about how `http::Post` is implemented

function abstraction

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

function abstraction allows us to not need to care about how `http::Post` is implemented

- if it uses multiple threads, we don’t care, nor do we need to!

function abstraction

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

function abstraction allows us to not need to care about how `http::Post` is implemented

- if it uses multiple threads, we don't care, nor do we need to!
- if it uses a GPU, we don't care, nor do we need to!

function abstraction

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

function abstraction allows us to not need to care about how `http::Post` is implemented

- if it uses multiple threads, we don’t care, nor do we need to!
- if it uses a GPU, we don’t care, nor do we need to!
- if it uses an FPGA or a SoC, we don’t care, nor do we need to!

function abstraction

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

moreover, if control was returned to us after executing `http::Post` and we were executing ...

- … on a different thread than the one we started on, we'd be surprised!
- ... on a GPU when we started on a CPU, we'd be surprised!

function abstraction

```
std::string SpellCheck(std::string text) {
  auto body = http::UrlEncode({"text", text});
  auto response = http::Post("https://www.online-spellcheck.com", body);
  return response.body;
}
```

moreover, if control was returned to us after executing `http::Post` and we were executing ...

- … on a different thread than the one we started on, we’d be surprised!
- ... on a GPU when we started on a CPU, we’d be surprised!

breaks the principle of least astonishment!

where should the continuation run?

using what ever execution resource it was using before you had to wait!

otherwise ...

```
auto SpellAndGrammarCheck(std::string text) {
  return SpellCheck(text)
      | Then([](auto&& text) {
        return GrammarCheck(text);
      });
}
```

otherwise ...

```
auto SpellAndGrammarCheck(std::string text) {
  return ThreadPool::Schedule([text]() {
    return SpellCheck(text)
        | Then([](auto&& text) {
            return GrammarCheck(text);
          });
  });
}
```

otherwise ...

```
auto SpellAndGrammarCheck(std::string text) {
  return ThreadPool::Schedule([text]() {
    return SpellCheck(text)
        | ThreadPool::Schedule([text]() {
            return Then([](auto&& text) {
              return GrammarCheck(text);
            });
          });
  });
}
```

otherwise ...

```
auto SpellAndGrammarCheck(std::string text) {
  return ThreadPool::Schedule([text]() {
    return SpellCheck(text)
        // Rescheduling on thread pool because we looked at
        // documentation of 'SpellCheck()' and it continues on the
        // event loop which we don't want to be on.
        | ThreadPool::Schedule([text]() {
            return Then([](auto&& text) {
              return GrammarCheck(text);
            });
          });
  });
}
```

otherwise ...

```
auto SpellAndGrammarCheck(std::string text) {
  return ThreadPool::Schedule([text]() {
    return SpellCheck(text)
        // Rescheduling on thread pool because we looked at  the
        // _implementation_ of 'SpellCheck()' and it continues on the
        // event loop which we don't want to be on.
        | ThreadPool::Schedule([text]() {
            return Then([](auto&& text) {
              return GrammarCheck(text);
            });
          });
  });
}
```

otherwise ...

```
auto SpellAndGrammarCheck(std::string text) {
  return ThreadPool::Schedule([text]() {
    return SpellCheck(text)
        // Rescheduling on thread pool because we  emailed the developers
        // of 'SpellCheck()'  and they said they'll use the event loop
        // in the future which we don't want to be on  so we're being
        // proactive now rather than deal with issues when the code changes.
        | ThreadPool::Schedule([text]() {
            return Then([](auto&& text) {
              return GrammarCheck(text);
            });
          });
  });
}
```

P2300R1 `std::execution`

Working with Asynchrony Generically: A Tour of C++ Executors - Eric Niebler

hierarchies of schedulers

“Composing Software Efficiently with Lithe” (PLDI 2010)

- allows for many simultaneous schedulers to be responsible for subtrees of the computation call graph (hierarchical)
- all computations have a scheduler context
- formal interface for resubmiting work to the scheduler that owns a context

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

streams

- “generator” type that follows the eventuals model/concept
- natural extension to eventuals
- perfect match for protobuf + gRPC

```
service RouteGuide {
  // Unary RPC.
  rpc GetFeature(Point) returns (Feature) {}

  // Server streaming RPC.
  rpc ListFeatures(Rectangle) returns (stream Feature) {}

  // Client streaming RPC.
  rpc RecordRoute(stream Point) returns (RouteSummary) {}

  // Bidirectional streaming RPC.
  rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}
```

```
service RouteGuide {
  // Unary RPC.
  rpc GetFeature(Point) returns (Feature) {}

  // Server streaming RPC.
  rpc ListFeatures(Rectangle) returns (stream Feature) {}

  // Client streaming RPC.
  rpc RecordRoute(stream Point) returns (RouteSummary) {}

  // Bidirectional streaming RPC.
  rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}
```

streams

```
auto ListFeatures(ServerContext* context, Rectangle* rectangle) {
  auto [left, right, top, bottom] = GetBoundingBox(*rectangle);
  return Iterate(feature_list_)
      | Filter([left, right, top, bottom](const Feature& f) {
          return f.location().longitude() < left
              || f.location().longitude() > right
              || f.location().latitude() < bottom
              || f.location().latitude() > top;
        });
}
```

streams

```
auto ListFeatures(ServerContext* context, Rectangle* rectangle) {
  auto [left, right, top, bottom] = GetBoundingBox(*rectangle);
  return Iterate(feature_list_)
      | Filter([left, right, top, bottom](const Feature& f) {
          return f.location().longitude() < left
              || f.location().longitude() > right
              || f.location().latitude() < bottom
              || f.location().latitude() > top;
        });
}
```

streams

```
auto ListFeatures(ServerContext* context, Rectangle* rectangle) {
  auto [left, right, top, bottom] = GetBoundingBox(*rectangle);
  return Iterate(feature_list_)
      | Filter([left, right, top, bottom](const Feature& f) {
          return f.location().longitude() < left
              || f.location().longitude() > right
              || f.location().latitude() < bottom
              || f.location().latitude() > top;
        });
}
```

follow along at https://github.com/3rdparty/eventuals-grpc

combinators

- `Map`
- `Reduce`
- `Head`
- `Take`
- `Collect`
- `StreamForEach` (aka nested `Map`)
- `Until` (aka “break”)
- ...

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

chapters

(1) motivating futures/promises + actors
(2) libprocess
(3) revisiting the problem
(4) evolution of libprocess
(5) eventuals
(6) scheduling
(7) streams
(8) type erasure

non-trivial types

- `auto` everywhere, loss of documentation from types
- header “only” development
- longer compile times
- debuggability(?)

type erasure

- type erase behind `Task<T>` and `Generator<T>`
- nothing is free, we trade off with heap allocation here
- can define everything behind interfaces, faster compilation & documentation
- naming mimics `task` and `generator` from C++ coroutines (which also heap allocate)

epilogue

epilogue

- you have to wait and you have state, be thoughtful about your approach

epilogue

- you have to wait and you have state, be thoughtful about your approach
- futures/promises are a good approach to waiting, but they have overhead

epilogue

- you have to wait and you have state, be thoughtful about your approach
- futures/promises are a good approach to waiting, but they have overhead
- continuations in continuations, aka lazy continuations, aka eventuals, can let you have your cake and eat it too, but there's no free lunch (dessert)

epilogue

- you have to wait and you have state, be thoughtful about your approach
- futures/promises are a good approach to waiting, but they have overhead
- continuations in continuations, aka lazy continuations, aka eventuals, can let you have your cake and eat it too, but there’s no free lunch (dessert)
- actors are a good approach to state, but you have to consider scheduling

epilogue

- you have to wait and you have state, be thoughtful about your approach
- futures/promises are a good approach to waiting, but they have overhead
- continuations in continuations, aka lazy continuations, aka eventuals, can let you have your cake and eat it too, but there’s no free lunch (dessert)
- actors are a good approach to state, but you have to consider scheduling
- functional composition is fundamental

epilogue

- you have to wait and you have state, be thoughtful about your approach
- futures/promises are a good approach to waiting, but they have overhead
- continuations in continuations, aka lazy continuations, aka eventuals, can let you have your cake and eat it too, but there’s no free lunch (dessert)
- actors are a good approach to state, but you have to consider scheduling
- functional composition is fundamental
- functional abstraction is fundamental

thanks!

@benh

https://github.com/3rdparty/eventuals

https://github.com/3rdparty/eventuals-grpc