C++ONLINE

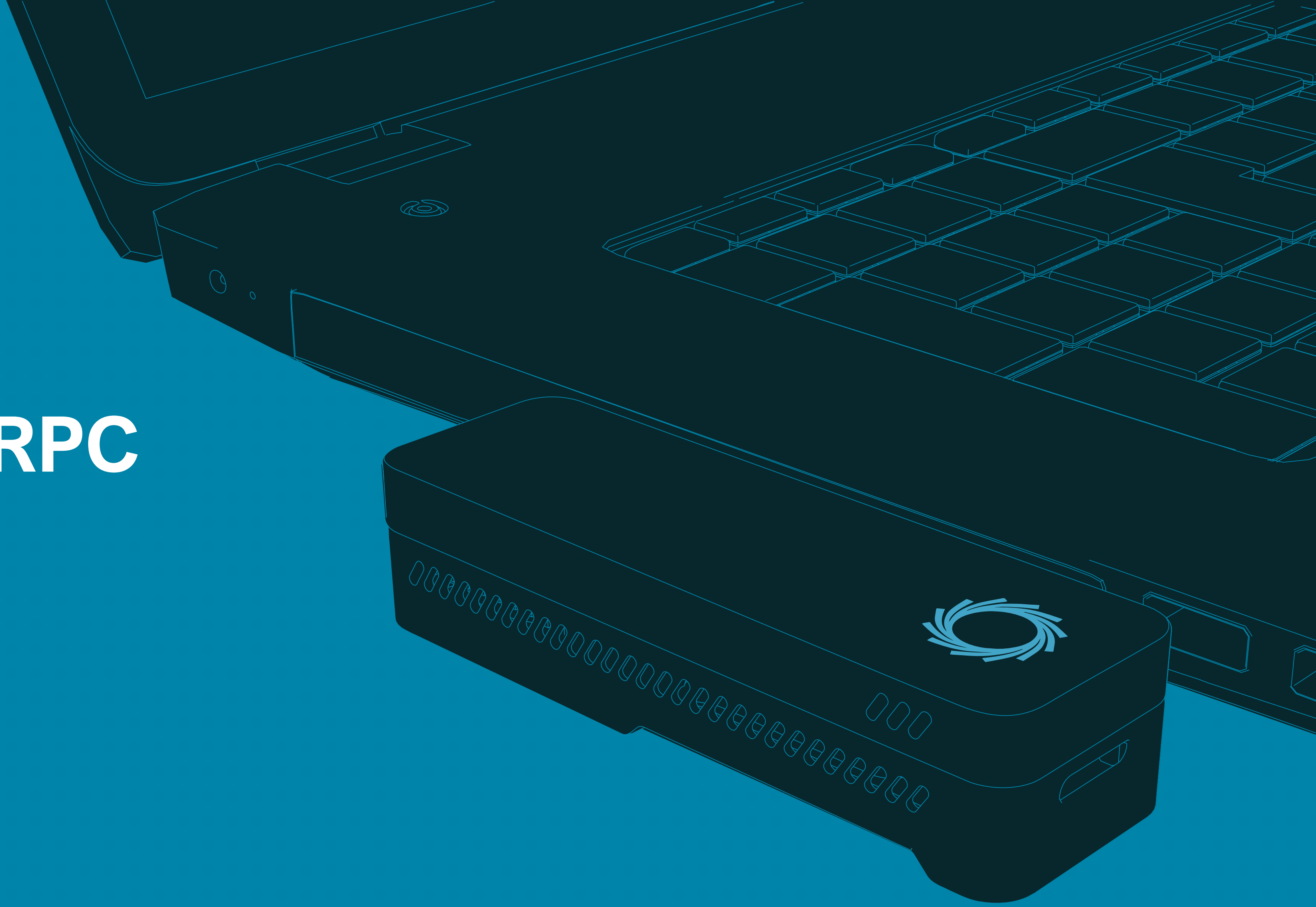
JONATHAN STOREY

COROUTINES AND gRPC

2024

Oxford NANOPORE Technologies

Coroutines and gRPC



Jonathan Storey

02 March 2024

Outline

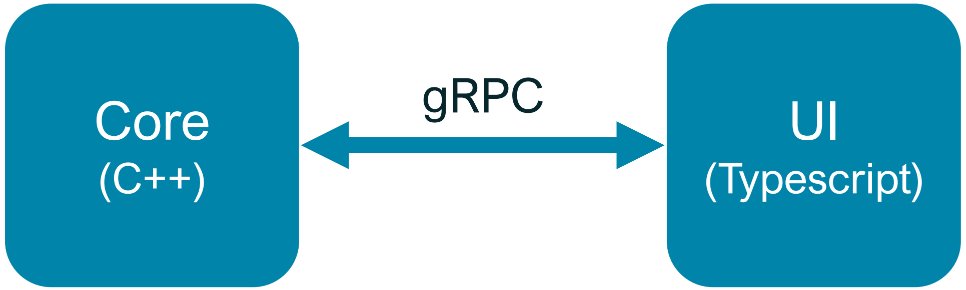
Motivation

At Oxford Nanopore Technologies, we:

- Make nanopore sequencers
- For DNA, RNA and other biomolecules

We have:

- The “Core” software, written in C++ 20
- A separate UI written in Typescript
- A gRPC communication channel between them



We want to:

- Implement a performant gRPC server in C++
- In a straightforward way

Aims

Understand

- What coroutines are
- Where they could be used in your applications

Implement

- Demonstrate basic (but useful!) coroutine implementation
- Focus on callback-based async libraries
- Clear idea of what is needed for coroutine support
- Confident to start experimenting with coroutines

Extend

- Roadmap of things to look at next

Outline

Part 1: An introduction to coroutines and gRPC

- Coroutines
- gRPC
- Coroutines and gRPC

Part 2: Writing coroutine support code

- The simplest coroutine
- Unary gRPC Coroutine
- Streaming gRPC Coroutine

Part 3: Finishing up

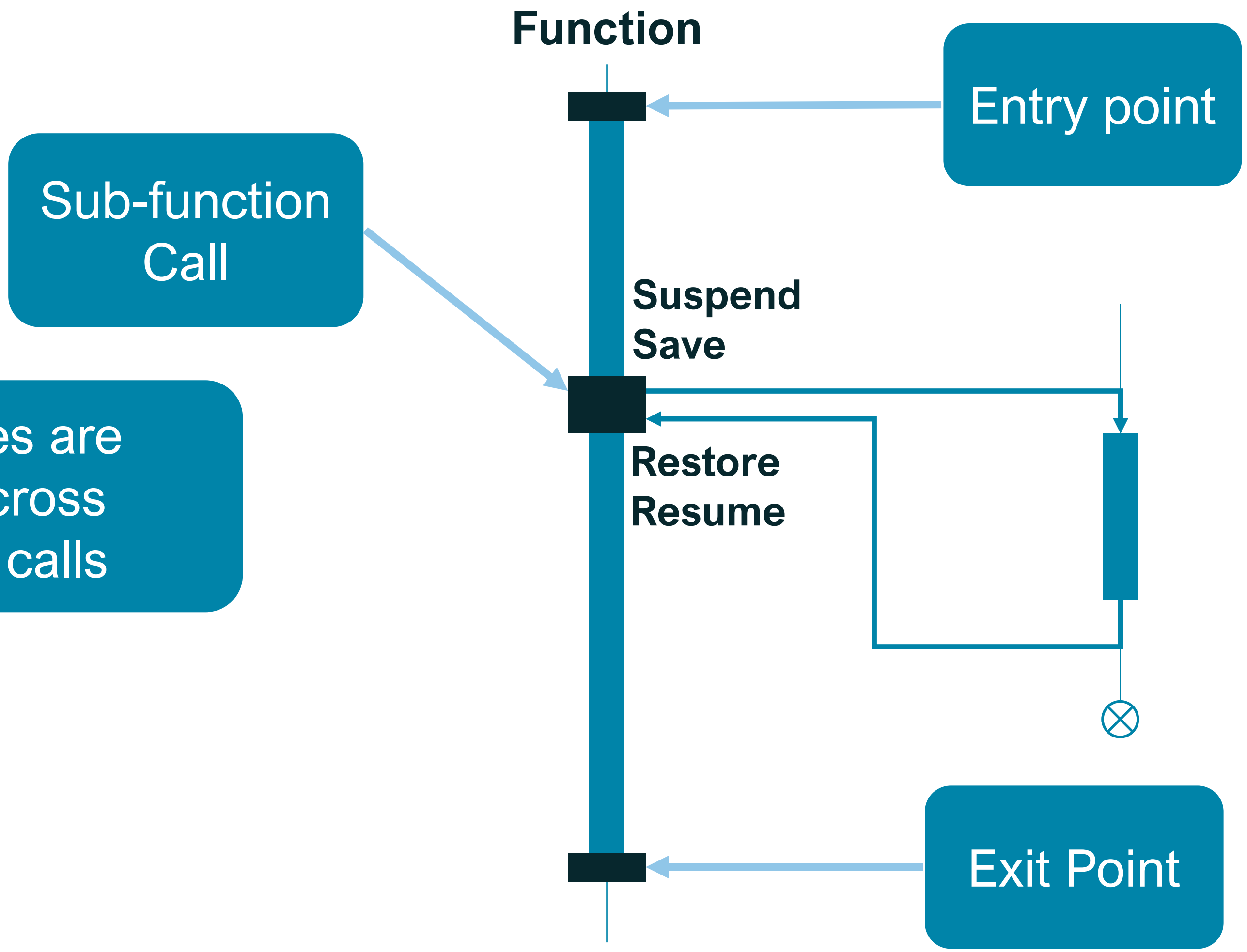
- Summary
- Further Steps

Part 1: An introduction to coroutines and gRPC

Coroutines

What is a coroutine?

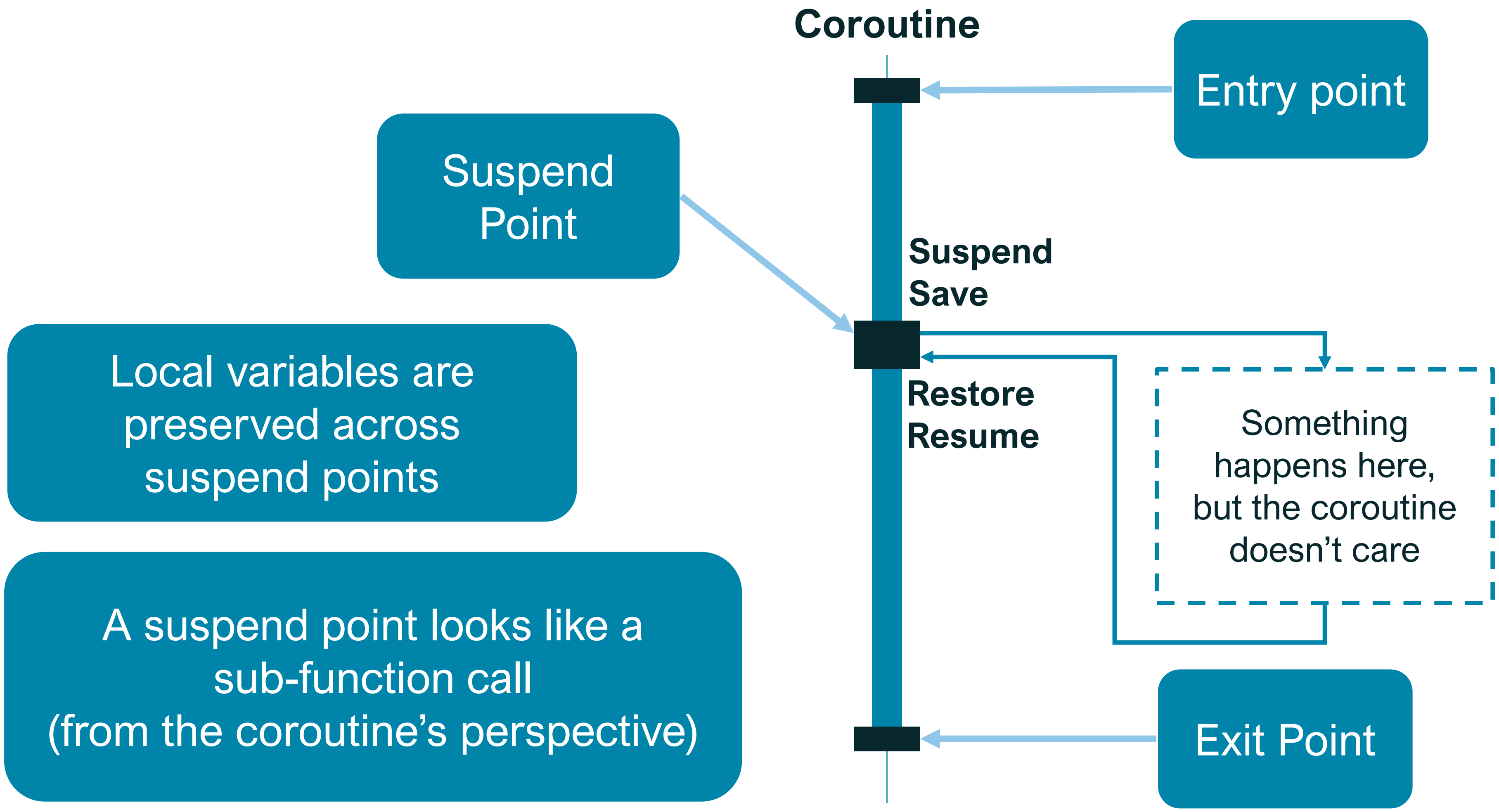
Calling a sub-function



Local variables are preserved across sub-function calls

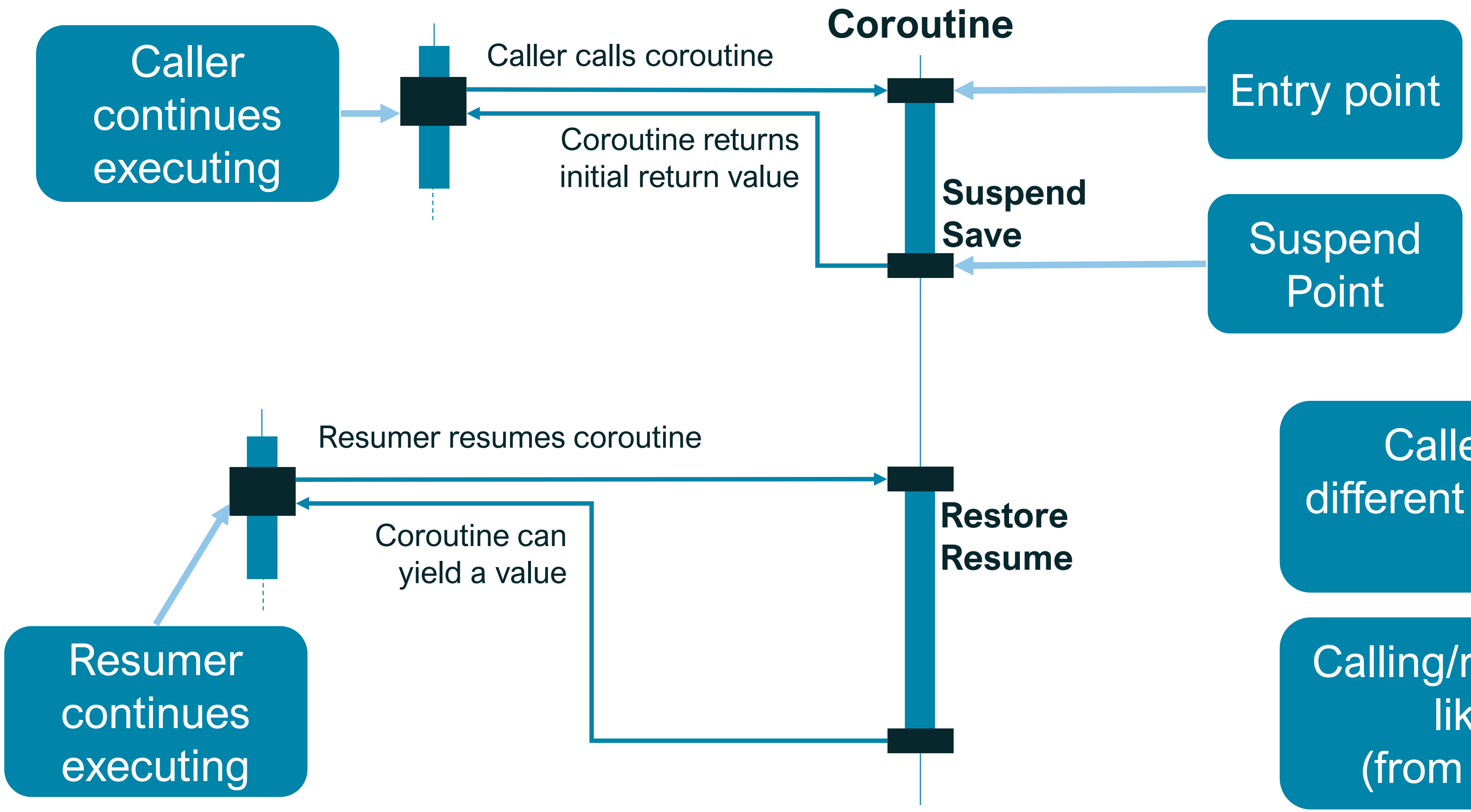
What is a coroutine?

Coroutines from the coroutine's perspective



What is a coroutine?

Coroutines from the caller/resumer’s perspective



Caller and Resumer can be different functions, even running on different threads

Calling/resuming a coroutine looks like a sub-function call (from the caller’s perspective)

What is a coroutine?

Summary

From the coroutine’s perspective:

- A suspend point looks like a normal function call…
- … but it’s not limited to just calling a function

From the caller/resumer’s perspective:

- Calling/resuming a coroutine looks like calling a normal function
- Control returns to the caller/resumer when the coroutine suspends or ends

Coroutines need to be resumed by something, somewhere

- They aren’t automatically resumed
- Need to plan when/where the coroutine will be resumed

A function is a special type of coroutine

- One without any suspend/resume points

Coroutines in C++

What is a coroutine

A coroutine is any function that contains one or more of the keywords

```
co_await
co_yield
co_return
```

Any of these keywords in a function body turns it into a coroutine

Compiler generates coroutine support code

From the outside, coroutines look like normal functions

Calling a coroutine returns the **initial return type** specified in its signature

- This is not what the co_yield or co_return produce!

In C++, the coroutine's behaviour depends on the **initial return type**

Coroutines in C++

Coroutine Support

Two sides to coroutines:

- Language Support
- Library Support

Currently, language support is pretty good

- You do need a modern compiler

Library support is pretty poor

- std::generator coming in C++23
  - But only implemented in latest GCC trunk
- Some “third party” coroutine support libraries

In the following examples we will assume that library code is available

Coroutines in C++

Illustrative Applications

Two standard examples

Generator

An object that yields a sequence of values

Task

Write asynchronous code that reads like blocking code

Using Coroutines

Generators

Generator

An object that yields a sequence of values

- “Sequence” implies remembering some state
  - (Knows how far it’s got through the sequence)

Coroutines remember state!

- Local variables
- Where it has got to
  - (Where to resume from)

Not going to mention generators any more

```cpp
#include <generator>

// Definition
std::generator<int> iota() {
    for (int i = 0; /* Never stop */ ; ++i) {
        co_yield i;
    }
}

// Usage
#include <print>
#include <ranges>

// Can iterate over generators
for (auto i : iota() | std::views::take(6)) {
    std::print("{} ", i);
}
// prints "0 1 2 3 4 5"
```

Using Coroutines

Generators

Generator

An object that yields a sequence of values

- "Sequence" implies remembering some state
  - (Knows how far it's got through the sequence)

Coroutines remember state!

- Local variables
- Where it has got to
  - (Where to resume from)

Not going to mention generators any more

```cpp
#include <generator>

// Definition
std::generator<int> iota() {
    for (int i = 0; /* Never stop */ ; ++i) {
        co_yield i;
    }
}

// Usage
#include <print>
#include <ranges>

// Can iterate over generators
for (auto i : iota() | std::views::take(6)) {
    std::print("{} ", i);
}
// prints "0 1 2 3 4 5"
```

Behaviour depends on **initial return type**

Remembers `i` between calls

Yields value – different from function return type!

Continues from here when resumed

Using Coroutines

Tasks: Blocking Implementation

Write asynchronous code that reads like blocking code

Simple blocking file copy function

Caller of `copy_file()` can't do anything else until file copying is complete

```
// Blocking example
//
// Copy from source to dest
void copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
) {
    auto data = read(src);
    write(dest, data);
}
```

Using Coroutines

Tasks: Blocking Implementation

Write asynchronous code that reads like blocking code

Simple blocking file copy function

Caller of `copy_file()` can't do anything else until file copying is complete

```
// Blocking example
//
// Copy from source to dest
void copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
) {
    auto data = read(src);
    write(dest, data);
}
```

Blocking Functions

Using Coroutines

Tasks: Coroutine Implementation

Write asynchronous code that reads like blocking code

```
// Blocking example
//
// Copy from source to dest
void copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
) {
    auto data = read(src);
    write(dest, data);
}
```

```
// Coroutine example
//
// Copy from source to dest
task copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
) {
    auto data = co_await async_read(src);
    co_await async_write(dest, data);
}
```

Using Coroutines

Tasks: Coroutine Implementation

Write asynchronous code that reads like blocking code

```
// Blocking example
//
// Copy from source to dest
void copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
) {
    auto data = read(src);
    write(dest, data);
}
```

```
// Coroutine example
//
// Copy from source to dest
task copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
) {
    auto data = co_await async_read(src);
    co_await async_write(dest, data);
}
```

Using Coroutines

Tasks: Coroutine Implementation

Write asynchronous code that reads like blocking code

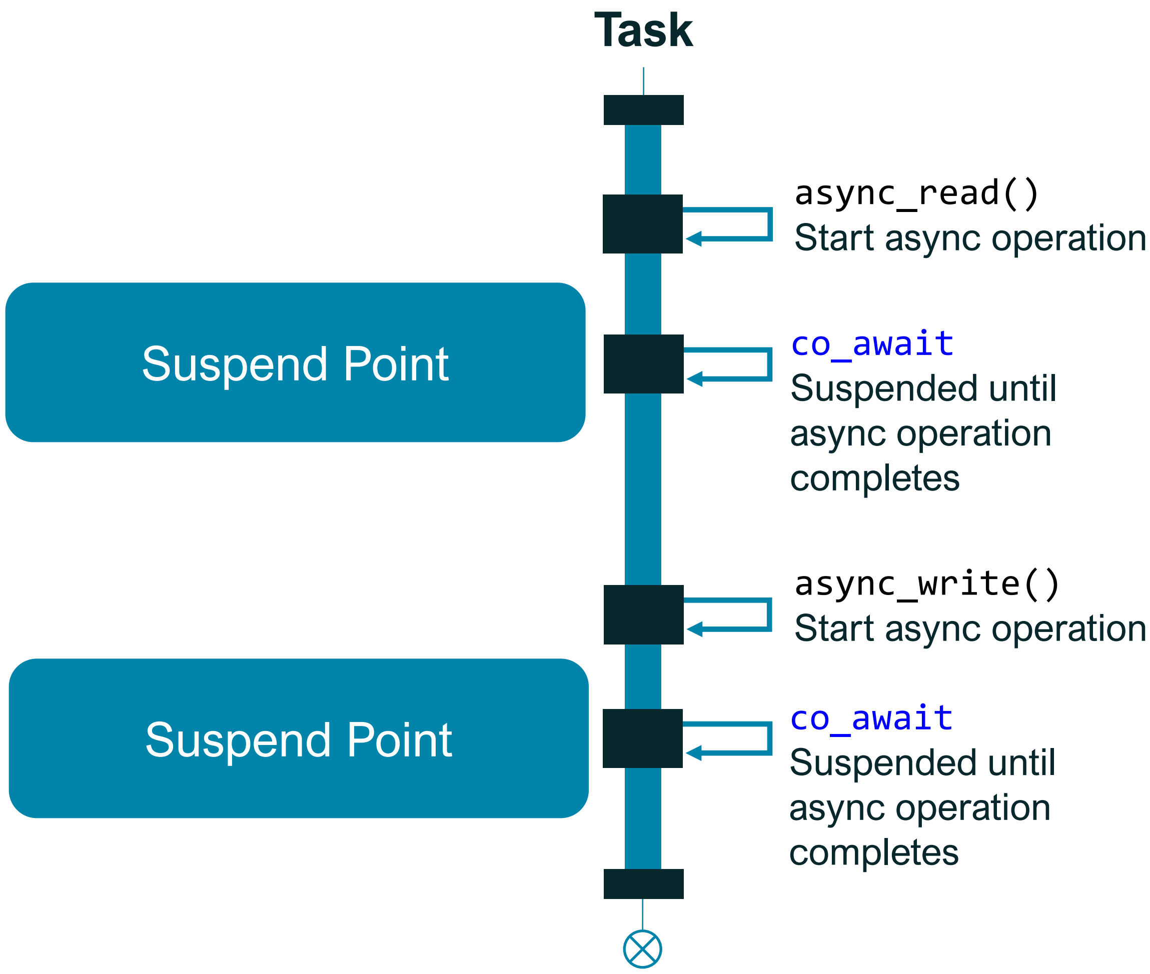
```
// Coroutine example
//
// Copy from source to dest
task copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
) {
    auto data = co_await async_read(src);
    co_await async_write(dest, data);
}
```

Behaviour depends on **initial return type**

"Coroutine-Aware" async functions

Using Coroutines

Tasks: Coroutine Perspective




```cpp
// Coroutine example
//
// Copy from source to dest
task copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
) {
    auto data = co_await async_read(src);
    co_await async_write(dest, data);
}
```

Using Coroutines

Tasks and Executors

Calling `copy_file()` returns a `task` object

A `task` object represents “work to be done”

Calling the coroutine just returns the task object

- No work is done initially
- Coroutine is suspended before executing the coroutine body

To actually do the work, need to execute the task

- Submit it to an executor…
- … which resumes the coroutine, and does the work

At it simplest, an executor is just a list of tasks and a thread to run them on

```
// Coroutine example
//
// Copy from source to dest
task copy_file(
    std::filesystem::path src,
    std::filesystem::path dest
```

This behaviour helps avoid race conditions

No code in the coroutine body is executed on thread that makes the task

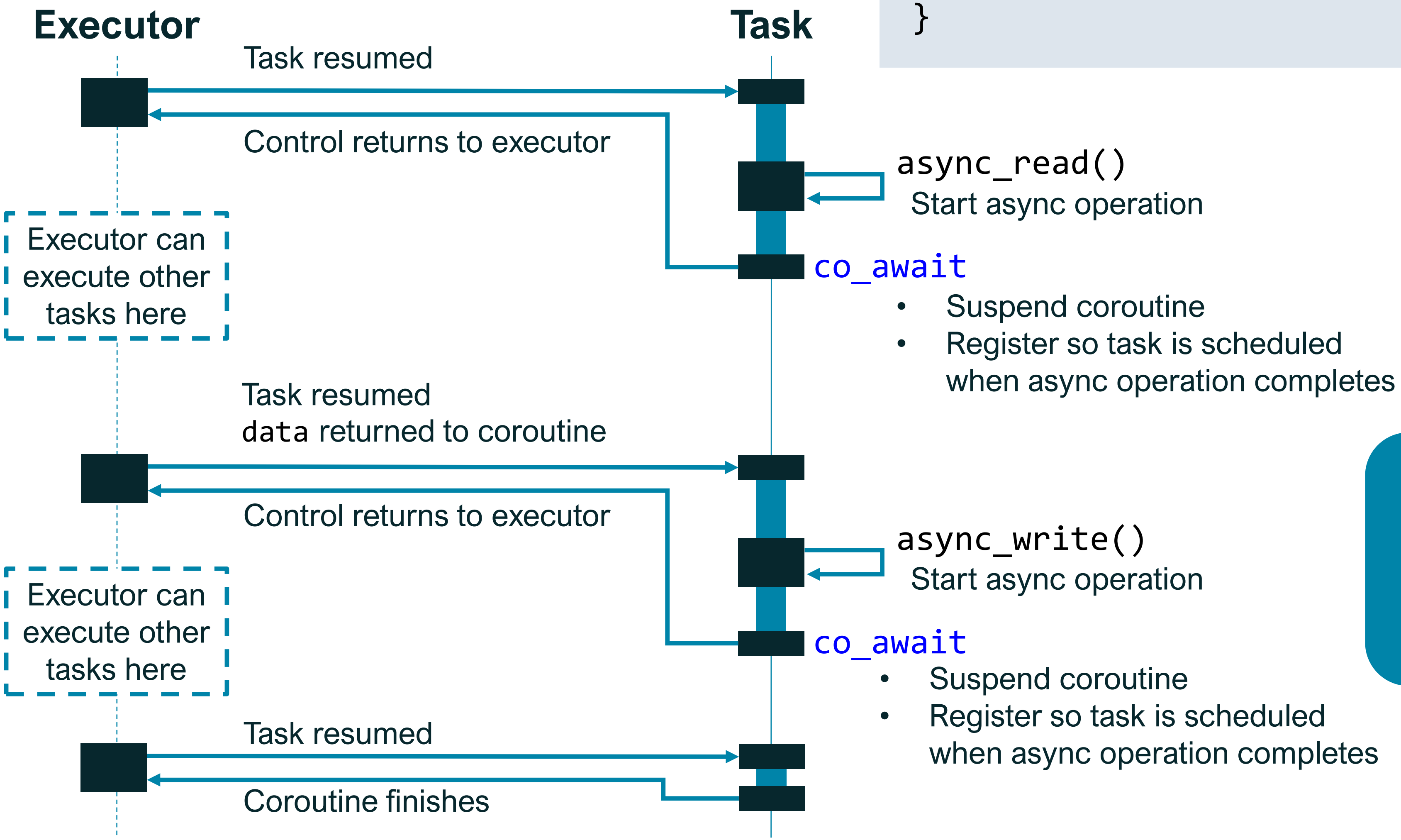
```
task copy_task = copy_file(src, dest);


submit(executor, copy_task);
```

Using Coroutines

Tasks: Executor’s Perspective

```
{
    auto data = co_await async_read(src);
    co_await async_write(dest, data);
}
```



Blocking operations inside the coroutine will block the executor thread from running other tasks

gRPC

gRPC

RPCs in General

RPC = Remote Procedure Call

- Call from one place, execute in another place

“Server” and “Client”

- Client submits requests to the server
- Server does the requested work
- Server returns a response to a client

Implementations provide abstractions

- Client side: Connect to server; call RPC; receive response
- Server side: Listen for requests; call application code when RPC is called

gRPC

gRPC

RPC implementation started by Google

User defines services, calls and messages in Protobuf

Generate framework code from Protobuf description

Call Type	Requests	Responses
Unary	1 (at start)	1 (at end)
Server Streaming	1 (at start)	Zero or more
Client Streaming	Zero or more	1 (at end)
Bidirectional	Zero or more	Zero or more

Implementation	Ease of Use	Scalability
Synchronous	Easy	Low (thread-per-call)
Async (event loop)	Very Difficult	High
Reactor (callback)	Moderate (boilerplate)	High (wrapper around Async)

gRPC

Comparison of Synchronous and Reactor Implementations

Write a simple call which:

- Receives a request indicating how many responses to write
- Writes that many responses
- Finishes with a status of `OK`

Write using both:

- Synchronous Implementation
- Reactor Implementation

Compare and contrast the required code

gRPC

Synchronous Server Streaming Call

gRPC supplies a `grpc::ServerWriter`, for writing responses

`grpc::ServerWriter::Write()` blocks until the write is complete

- User code can also make any other blocking calls they like

gRPC internally spins up a new thread for every call

```
::grpc::Status ServiceImpl::stream_responses(
    ::grpc::ServerContext * /*context*/,
    Request const * request,
    ::grpc::ServerWriter<Response> * writer
) {
    Response response;
    for (int k = 0; k < request->count(); ++k) {
        response.set_idx(k);
        response.set_message(std::format(
            "Message for response {}", k));

        writer->Write(response); // Blocking write
    }

    return ::grpc::Status::OK;
}
```

gRPC

Reactor Server Streaming Call

Reactor call looks different

Return a `Reactor`, which implements the actual logic of the call

Reactors are run on an internal gRPC thread pool

- Scalable (lower overhead per request)
- Blocking calls within the reactor will block threads in the thread pool
  - Reduce throughput!

```
::grpc::ServerWriteReactor<Response> *
ServiceImpl::stream_responses(
    ::grpc::CallbackServerContext *,
    Request const * request
) {
    return new Reactor{ request };
}
```

Need to write this `Reactor` class

gRPC

Writing a Reactor

`grpc::ServerWriteReactor<Response>` uses the following sequence:

- Call `StartWrite(&response)`
- Wait for `OnWriteDone()` to be called
- Repeat these steps for each response

Then, when all responses have been sent

- Call `Finish(status)`
- Wait for `OnDone()` to be called
- Clean up the reactor

In our `Reactor` class, we need to:

- Write some code that calls `StartWrite()` when the reactor is created
- Override `OnWriteDone()`, to either:
  - Call `StartWrite()` to start sending the next response, or
  - Call `Finish(status)` if all responses have been sent
- Override `OnDone()` to clean up the reactor

Server Write Reactor

Example Reactor Implementation

```
struct Reactor : grpc::ServerWriteReactor<Response> {
  Reactor(Request * request)
  : m_target_count(request->count) {
     send();
  }

  void send() {
    if (m_curr_count < m_target_count) {
      m_response.set_idx(m_curr_count);
      m_response.set_message(std::format(
          "Message for response {}", k));
      // Start writing this response
      this->StartWrite(&m_response);
    } else {
      // No more responses -- all done
      this->Finish(::grpc::Status::OK);
    }
  }
```

```
  void OnWriteDone(bool ok) override {
     if (ok) {
       // Send the next response
       ++m_curr_count;
       send();
     }
     else {
       // Write failed -- finish
       this->Finish(::grpc::Status::CANCELLED);
     }
   }

   void OnDone() override {
     // Tidy up `*this` -- this is a gRPC idiom
     delete this;
   }

   std::size_t m_target_count = 0;
   std::size_t m_curr_count = 0;
   Response m_response;
};
```

Keep track of state

Lifetimes!

Server Write Reactor

Example Reactor Implementation

```
struct Reactor : grpc::ServerWriteReactor<Response> {
  Reactor(Request * request)
  : m_target_count(request->count) {
     send();
  }

  void send() {
    if (m_curr_count < m_target_count) {
      m_response.set_idx(m_curr_count);
      m_response.set_message(std::format(
          "Message for response {}", k));
      // Start writing this response
      this->StartWrite(&m_response);
    } else {
      // No more responses -- all done
      this->Finish(::grpc::Status::OK);
    }
  }
```

```
  void OnWriteDone(bool ok) override {
    if (ok) {
      // Send the next response
      ++m_curr_count;
      send();
    }
    else {
      // Write failed -- finish
      this->Finish(::grpc::Status::CANCELLED);
    }
  }

  void OnDone() override {
    // Tidy up `*this` -- this is a gRPC idiom
    delete this;
  }

  std::size_t m_target_count = 0;
  std::size_t m_curr_count = 0;
               esponse;
```

Can you spot the loop?

gRPC

Writing a Reactor

Need to manually manage state

Lots of boilerplate to write a reactor

- Could reduce this with a helper class

Difficult to combine with other functionality

- Send a response every minute
- Send responses as data becomes available
- Would need to write a new reactor in each case

Coroutines and gRPC

Coroutines and gRPC

What client code do we want to write?

Start by writing the code we wish we could write

Remember that task-based coroutines can make async calls look like blocking calls

Ideally, the whole `stream_responses()` call could just be a coroutine

- We could write `co_await` and `co_return` directly inside the function body

```
::grpc::Status
ServiceImpl::stream_responses(
    ::grpc::ServerContext * /*context*/,
    Request const * request,
    ::grpc::ServerWriter<Response> * writer
) {
    Response response;
    for (int k = 0; k < request->count(); ++k) {
        response.set_idx(k);
        response.set_message(std::format(
            "Message for response {}", k));

        writer->Write(response); // Blocking write
    }

    return ::grpc::Status::OK;
}
```

This is the blocking code

Coroutines and gRPC

What client code do we want to write?

Start by writing the code we wish we could write

Remember that task-based coroutines can make async calls look like blocking calls

Ideally, the whole `stream_responses()` call could just be a coroutine

- We could write `co_await` and `co_return` directly inside the function body

```
::grpc::ServerWriteReactor<Response> *
ServiceImpl::stream_responses(
    ::grpc::CallbackServerContext * /*context*/,
    Request const * request

) {
    Response response;
    for (int k = 0; k < request->count(); ++k) {
        response.set_idx(k);
        response.set_message(std::format(
            "Message for response {}", k));

        co_await send(response);
    }

    co_return ::grpc::Status::OK;
}
```

Async code looks just like the blocking code

Coroutines and gRPC

What client code do we want to write?

Turns out, with a bit of coroutine library code, we **can** write exactly this

```
::grpc::ServerWriteReactor<Response> *
ServiceImpl::stream_responses(
    ::grpc::CallbackServerContext * /*context*/,
    Request const * request

) {
    Response response;
    for (int k = 0; k < request->count(); ++k) {
        response.set_idx(k);
        response.set_message(std::format(
            "Message for response {}", k));

        co_await send(response);
    }

    co_return ::grpc::Status::OK;
}
```

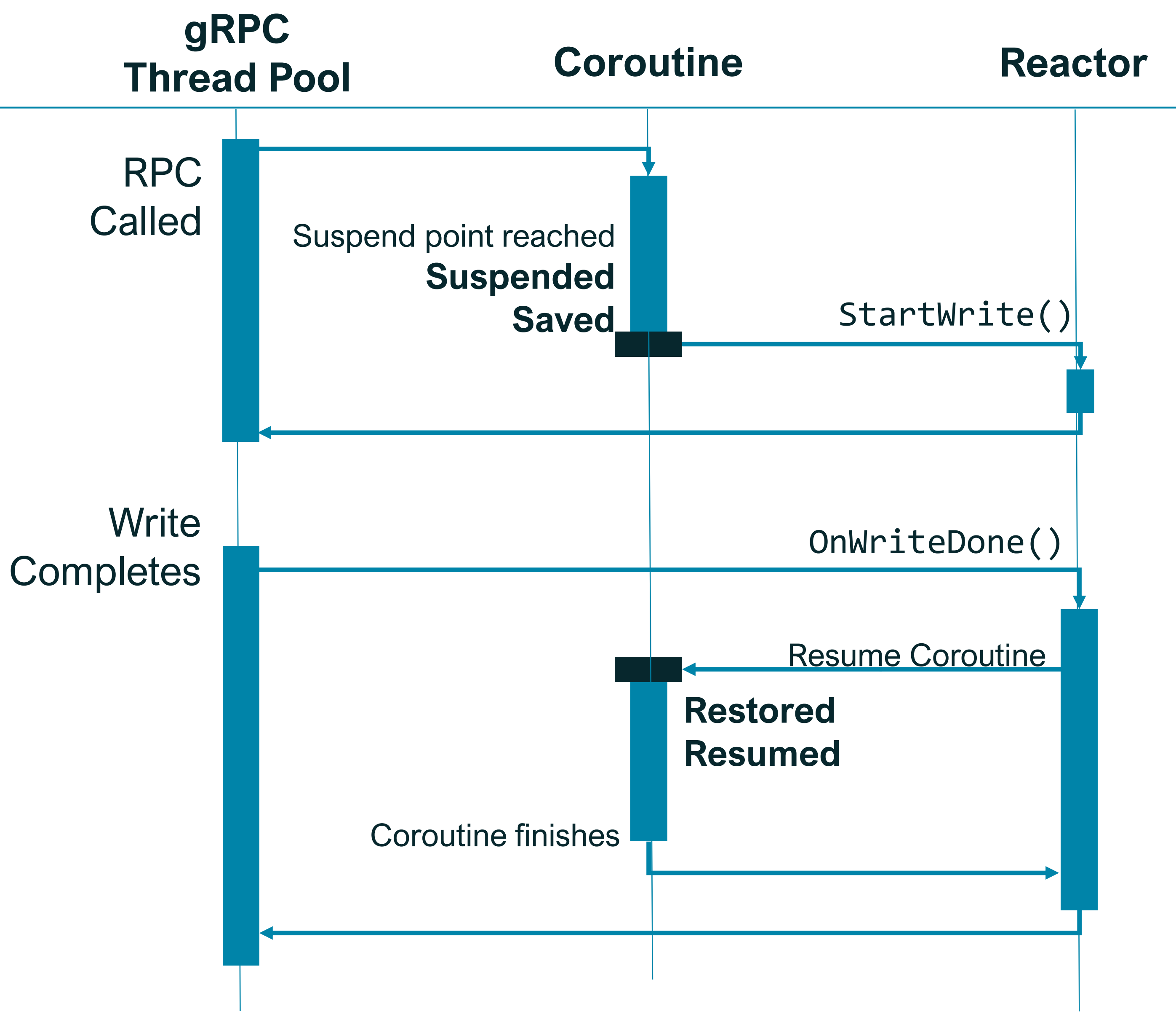
- Suspend the coroutine
- Start sending the response
- Resume the coroutine when the response has been sent

- Finish the call with the supplied status

Coroutines and gRPC

co_await diagram

The gRPC runtime acts as the executor



Coroutines and gRPC

Benefits of using coroutines

Simple to write user code

- It reads just like the synchronous code
- But you get the performance benefits of using the reactor implementation

Impossible to get call order wrong

- In gRPC, you can't have two writes in flight at once
  - `co_await` means that no other calls can be made until the current one finishes
- In gRPC, you can't call anything after finishing
  - `co_return` means that no other calls can be made after returning the final status
- Don't need to worry about exceptions
  - With C++ coroutines, you can specify a custom exception handler for a coroutine

Composable with other coroutine code

- Async timer waits
- Async waits for data to become available
- Even async network or filesystem calls

Summary

Summary of Part 1

What have we seen so far?

C++20 has language support for coroutines, but little library support

Task-style coroutines can make async code look like blocking code

gRPC has a performant Reactor implementation

- But it’s a nuisance to use

We imagined the code we’d like to be able to write

- Reads like blocking code
- Performs like reactor code
- Safe to use
- Composable with other async calls

Outlook for Part 2

Where do we want to go

Find out how to make a function a coroutine

Add support for gRPC coroutines:

- `co_return status;`
- `co_await send(response);`

Part 2: Writing coroutine code

What we are aiming for

And how are we going to get there

Plan

1. Write support code for the simplest coroutine we can imagine

2. Write support for
   ```
   co_return status;
   ```

3. Write support for
   ```
   co_await send(response)
   ```

```cpp
::grpc::ServerWriteReactor<Response> *
ServiceImpl::stream_responses(
    ::grpc::ServerContext * /*context*/,
    Request const * request

) {
    Response response;
    for (int k = 0; k < request->count(); ++k) {
        response.set_idx(k);
        response.set_message(std::format(
            "Message for response {}", k));

        co_await send(response);
    }

    co_return ::grpc::Status::OK;
}
```

The Simplest Coroutine

The Simplest Coroutine

The simplest coroutine

```
#include <coroutine>

struct InitialReturn{};

InitialReturn f() {
   co_return;
}
```

The Simplest Coroutine

The simplest coroutine

```cpp
#include <coroutine>

struct InitialReturn{};

InitialReturn f() {
    co_return;
}
```

Returns `InitialReturn` object to caller…

… but `co_return` doesn't return anything!

Where does the `InitialReturn` object come from? How is it constructed?

The compiler calls a user-supplied function to make an `InitialReturn` object

This function is part of the `promise_type`, which defines the coroutine behaviour

error: this function cannot be a coroutine: '`std::coroutine_traits<InitialReturn>`' has no member named '`promise_type`'

The Simplest Coroutine

The `promise_type`

The `promise_type` defines the behaviour of the coroutine

- We'll see exactly how later!

```
std::coroutine_traits<InitialReturn, Args …>::promise_type
```

- Default looks up `InitialReturn::promise_type`

"Let the compiler guide you" – (Mateusz Pusz)

This is why we said the coroutine behaviour depends on the initial return type!

The Simplest Coroutine

The `promise_type`

Live coding!

godbolt.org

```
#include <coroutine>

struct InitialReturn{};

InitialReturn f() {
   co_return;
}
```

The Simplest Coroutine

The `promise_type`

```cpp
#include <coroutine>

struct InitialReturn {
    struct promise_type {
        constexpr InitialReturn get_return_object() noexcept { return {}; }
        constexpr std::suspend_never initial_suspend() noexcept { return {}; }
        constexpr void return_void() noexcept { /* Do nothing */ }
        constexpr void unhandled_exception() noexcept { /* Swallow exception */ }
        constexpr std::suspend_never final_suspend() noexcept { return {}; }
    };
};

InitialReturn f() {
    co_return;
}
```

The Simplest Coroutine

The compiler transformation

```
InitialReturn f() {
     using promise_type = std::coroutine_traits<InitialReturn>::promise_type;
     promise_type promise;
     auto return_object = promise.get_return_object();

     co_await promise.initial_suspend();  // -> co_await std::suspend_never{};

     try {
          // coroutine body
          // co_return
          promise.return_void();
     }
     catch {
          promise.unhandled_exception();  // -> Never called – body doesn’t throw
     }
     co_await promise.final_suspend();  // -> co_await std::suspend_never{};

     return return_object;
}
```

We saw during live coding that functions are called in this order

The return_object is returned when control is returned to the initial caller

The Simplest Coroutine

Summary

Wrote support for the simplest coroutine

Introduced the `promise_type`

- Defines the coroutine’s behaviour

Looked at the compiler transform

- This is the code that the compiler automatically writes when it encounters a coroutine

The Simplest gRPC Coroutine

The Simplest gRPC Coroutine

“Everything’s OK”

Useless, but allows us to focus on the coroutine support code

```
#include <coroutine>

grpc::ServerUnaryReactor *
everythings_ok(
    grpc::CallbackServerContext * /* ctx */,
    Request const * /* request */,
    Response * /* response */)
{
        co_return ::grpc::Status::OK;
}
```

Unary:
One Request
One Response

Just returns OK!

Why use coroutines for a unary RPC?

- Opt in to the more scalable reactor implementation
- Allows the use of other async coroutine code in the implementation

The Simplest gRPC Coroutine

Where to put the promise_type

Previously, put `promise_type` directly in the initial return type `InitialReturnT`

But this time, the return type must be `grpc::ServerUnaryReactor*`

- The type is from the gRPC library
- Can't modify

Instead, specialise `std::coroutine_traits` for unary RPCs

```
template <typename Request, typename Response>
struct std::coroutine_traits<
    grpc::ServerUnaryReactor *,
    grpc:: CallbackServerContext *,
    Request const *, Response *
>
{
    using promise_type = ServerUnaryReactorPromiseType;
};
```

Return type

Function Parameter Types

The promise type to use

Oxford NANOPORE Technologies

The Simplest gRPC Coroutine

Where to put the promise_type

Previously, put `promise_type` directly in the initial return type `InitialReturnT`

But this time, the return type must be `grpc::ServerUnaryReactor*`

- The type is from the gRPC library
- Can't modify

Instead, specialise `std::coroutine_traits` for unary RPCs

```
template <typename ... Args>
struct std::coroutine_traits<
    grpc::ServerUnaryReactor *,
    Args ...

>
{
    using promise_type = ServerUnaryReactorPromiseType;
};
```

Return type

Function Parameter Types

Can also make behaviour depend only on return type

The promise type to use

The Simplest gRPC Coroutine

Writing the `promise_type`

```
struct ServerUnaryReactorPromiseType {
      struct UnaryReactor {
            void OnDone() override {
                  // Clean ourselves up
                  // This is a gRPC idiom
                  delete this;
            }
      };

      UnaryReactor * m_reactor = nullptr;

      void finish(grpc::Status status) {
            assert(m_reactor);
            m_reactor->Finish(status);
            m_reactor = nullptr;
      }

      // Destructor: Finish if not already finished
      ~ServerUnaryReactorPromiseType() {
            if (m_reactor) {
                  finish(::grpc::Status::CANCELLED);
            }
      }
```

```
      grpc::ServerUnaryReactor * get_return_object() {
            m_reactor = new UnaryReactor{};
            return m_reactor;
      }

      constexpr std::suspend_never initial_suspend() {
            return {};
      }

      void return_value(::grpc::Status status) {
            finish(status);
      }

      void unhandled_exception() {
            finish(::grpc::Status{
                  ::grpc::StatusCode::UNKNOWN,
                  “Unhandled Exception”,
            });
      }

      constexpr std::suspend_never final_suspend() noexcept {
                  return {};
      }
};
```

The Simplest gRPC Coroutine

Writing the `promise_type`

Reactor code is very simple!

Always need to call finish

```
struct ServerUnaryReactorPromiseType {
      struct UnaryReactor {
            void OnDone() override {
                  // Clean ourselves up
                  // This is a gRPC idiom
                  delete this;
            }
      };
                                  nullptr;

      void finish(grpc::Status status) {
            assert(m_reactor);
            m_reactor->Finish(status);
            m_reactor = nullptr;
      }

      // Destructor: Finish if not already finished
      ~ServerUnaryReactorPromiseType() {
            if (m_reactor) {
                  finish(::grpc::Status::CANCELLED);
            }
      }
```

Same basic outline as the promise_type we saw before

Using suspend_never

```
      grpc::ServerUnaryReactor * get_return_object() {
            m_reactor = new UnaryReactor{};
            return m_reactor;
      }

      constexpr std::suspend_never initial_suspend() {
            return {};
      }

      void return_value(::grpc::Status status) {
            finish(status);
      }

      void unhandled_exception() {
            finish(::grpc::Status{
                  ::grpc::StatusCode::UNKNOWN,
                  "Unhandled Exception",
            });
      }

      constexpr std::suspend_never final_suspend() noexcept {
                  return {};
      }
};
```

The Simplest gRPC Coroutine

Summary

Added coroutine support to Unary RPCs

Implemented a basic gRPC `promise_type`

- Added support for returning a status using `co_return`
- Saw that the `promise_type` code is mostly boilerplate

Saw that the `Reactor` side of the code is simple

Streaming Responses

Streaming Responses

"Everything's OK"

Now with added Responses!

Server Streaming:
One Request
Zero or More Responses

```
grpc::ServerWriteReactor<Response> *
streaming_everythings_ok(
    grpc::CallbackServerContext * /* ctx */,
    Request const * /* request */)
{
    Response response;
    co_await send(response);
    co_return ::grpc::Status::OK;
}
```

Problem

- We need access to the `Reactor` to write the response
- The reactor lives in the `promise_type`
- How do we get access to the reactor?

Plan

- Suspend the coroutine
- Call `Reactor::StartWrite()`
- Resume the coroutine when `OnWriteDone()` is called

Streaming Responses

Awaitables and awaiters

In order for `co_await send(response)` to work, the result of `send(response)` needs to be **awaitable**

An **awaitable** is something that is (or that can be transformed into) an **awaiter**

For now, make `send()` return an **awaiter** directly

Obtaining the **awaitable** and **awaiter** can be complicated!

Not going to go into more detail in this talk

Some pointers:

- `operator co_await`
  - Obtain an awaiter from an awaitable
  - Like a user-defined conversion function
- `promise_type::await_transform()`
  - Powerful, but difficult to use
  - All-or-nothing!

https://en.cppreference.com/w/cpp/language/coroutines#co_await

Streaming Responses

The compiler transformation

```
auto await_result = co_await awaiter;
```

```
if (!awaiter.await_ready()) {
    // Compiler magic:
    //    - suspend the coroutine
    //    - store the resume point
    //    - get the coroutine_handle
    auto suspend_return =
        awaiter.await_suspend(coroutine_handle);
    // Execution starts again from here when
    //     `coroutine_handle.resume()` is called
    // Compiler magic: restore coroutine state
}
auto await_result = awaiter.await_resume();
// Rest of coroutine body here
```

Streaming Responses

The compiler transformation

```
auto await_result = co_await awaiter;
```

```
if (!awaiter.await_ready()) {
    // Compiler magic:
    //    - suspend the coroutine
    //    - store the resume point
    //    - get the coroutine_handle
    auto suspend_return =
        awaiter.await_suspend(coroutine_handle);
    // Execution starts again from here when
    //     `coroutine_handle.resume()` is called
    // Compiler magic: restore coroutine state
}
auto await_result = awaiter.await_resume();
// Rest of coroutine body here
```

Optimization Opportunity
Only suspend/resume if necessary

`await_suspend()` return value is complicated

- Return to caller
- Resume this coroutine
- Resume another coroutine

Coroutine handle can be used to:

- Resume coroutine
- Destroy coroutine
- Access `promise_type`

Coroutine is **suspended** at this point – safe for resumer to resume immediately!

Provides await result
Can also do tidying up in here
Always called (even if we didn't wait)

Streaming Responses

Implementation Outline

Awaiter

```
void await_suspend(std::coroutine_handle<promise_type> coroutine_handle)
{
    auto & promise = coroutine_handle.promise();
    promise.m_reactor->m_coroutine_handle = coroutine_handle;
    promise.m_reactor->StartWrite(&m_response);
}
Response const & m_response;
```

Reactor

```
void OnWriteDone(bool ok) override {
    m_coroutine_handle.resume();
}

std::coroutine_handle<> m_coroutine_handle;
```

Streaming Responses

Implementation Outline

Awaiter

```
void await_suspend(std::coroutine_handle<promise_type> coroutine_handle)
{
    auto & promise = coroutine_handle.promise();
    promise.m_reactor->m_coroutine_handle = coroutine_handle;
    promise.m_reactor->StartWrite(&m_response);
}
Response const & m_response;
```

Get the promise, so we can access the Reactor

Store the coroutine_handle in the reactor, so we know what to resume when the write completes

Start the write
When it's done, OnWriteDone() will be called

Reactor

```
void OnWriteDone(bool ok) override {
    m_coroutine_handle.resume();
}

std::coroutine_handle<> m_coroutine_handle;
```

When the write is done, resume the coroutine

Streaming Responses

Implementation Outline

Awaiter

```
void await_suspend(std::coroutine_handle<promise_type> coroutine_handle)
{
    auto & promise = coroutine_handle.promise();
    promise.m_reactor->m_coroutine_handle = coroutine_handle;
    promise.m_reactor->StartWrite(&m_response);
}
Response const & m_response;
```

OK to store reference and take pointer to response

The lifetime of local variables/temporaries in a coroutine is extended across suspend point

Reactor

```
void OnWriteDone(bool ok) override {
    m_coroutine_handle.resume();
}

std::coroutine_handle<> m_coroutine_handle;
```

Streaming Responses

Full Awaiter Implementation

```
struct SendResponseAwaiter {
     // Always suspend
     constexpr bool await_ready() noexcept { return false; }

     // Returns void
     //  -> Always return control to coroutine caller/resumer
     void await_suspend(auto coroutine_handle) {
          auto & promise = coroutine_handle.promise();
          promise.m_reactor->m_coroutine_handle = coroutine_handle;
          promise.m_reactor->StartWrite(&m_response);
     }

     constexpr void await_resume() noexcept { /* Do  nothing */ }

     Response const & m_response;
}



SendResponseAwaiter send(Response const & response) { return {response}; }
```

Streaming Responses

Full Reactor Implementation

```
struct WriteReactor {
    void OnDone() override { delete this; }

    void OnWriteDone(bool ok) override {
        if (ok) {
            m_coroutine_handle.resume();
        } else {
            // RPC cancelled – no point in continuing with coroutine
            m_coroutine_handle.destroy();
        }
    }

    std::coroutine_handle<promise_type> m_coroutine_handle;
};
```

Handle the case where the write fails

The rest of the promise_type code is the same as in the Unary case!

The overall coroutine behaviour is the same

Streaming Responses

Full Reactor Implementation

```
struct WriteReactor {
    void OnDone() override { delete this; }

    void OnWriteDone(bool ok) override {
        if (ok) {
            m_coroutine_handle.resume();
        } else {
            // RPC cancelled – no point in continuing with coroutine
            m_coroutine_handle.destroy();
        }
    }

    std::coroutine_handle<> m_coroutine_handle;
};
```

Handle the case where the write fails

Type erasure on the coroutine_handle type

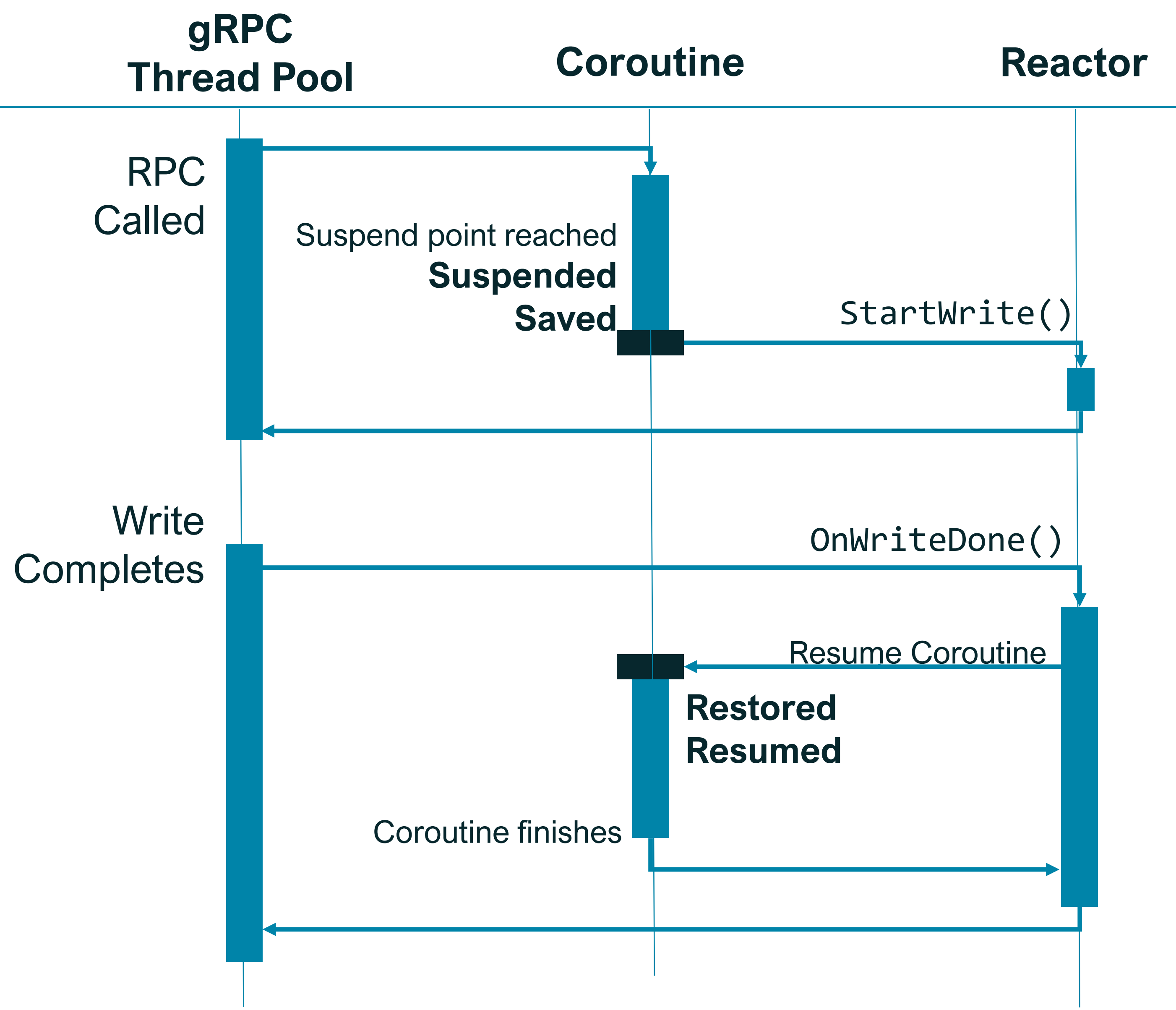
The rest of the promise_type code is the same as in the Unary case!

The overall coroutine behaviour is the same

Streaming Responses

Diagram

The gRPC runtime acts as the executor



Streaming Responses

Summary

Introduced awaiters and awaitables

Wrote an awaiter that…

- Suspends the coroutine
- Starts writing a response
- Schedules the coroutine to be resumed when the write completes

…when it is awaited

Saw that the `Reactor` side of the code is simple

Part 3: Finishing Up

Summary

Adding Coroutine Support for gRPC

Achievements

Integrated coroutines and gRPC!

The support code we wrote is straightforward

- Self-contained
- Can be understood and built on
- Could support most of gRPC in a few hundred lines of code

Example of adding coroutine support to callback-style async code

- Take the techniques presented in this talk
- Apply them to your own callback-style API
- Combine your callback-style API with other coroutine code

Future directions

Where could we go from here

Lots of options for further work

Aims of this section:

- **Build** on the simple code we've seen so far
- **Highlight** what kind of things other people have done
- **Inspire** you when you come to write your own coroutine support code
    - Useful, not necessarily obvious, things

Future directions

Expand gRPC support

Add support for the rest of the server operations

- They all follow the same pattern
  - Start operation
  - `OnOperationDone()`
- Adding support for them is straightforward

Add support for client operations

- Again, very similar to server operations
- Some additional complications to consider
  - But not insurmountable

Add support for gRPC timers

- “Alarms” in gRPC
- Useful for doing async “sleeps”, etc.

Future Directions

Make the reactor more flexible

Make reactor functions “coroutine-aware”

- `read()`
- `write()`

When calling these functions, they:

- Start the operation
- Return an `Awaitable` directly
- Calling `co_await awaitable` suspends the coroutine until the operation is complete

This is how coroutine-aware code is “expected” to work

```
Reactor reactor;

co_await reactor.write(response);

// Can be split
Awaitable awaitable = reactor.write(response);
co_await awaitable;
```

Allows running operations in parallel!

```
Awaitable write_awaitable = reactor.write(response);
Request request = co_await reactor.read();
Response new_response = make_response(request);

// Need to wait for previous write to finish before
// starting another write
co_await write_awaitable;
co_await reactor.write(new_response);
```

“Make easy things easy, make hard things possible”
-Larry Wall

Future Directions

Cancellation

The client can cancel an ongoing gRPC call at any time

Support for cancelling operations is included in most coroutine support libraries

If the gRPC call is cancelled, we'd like to stop immediately

- e.g. if sleeping on a timer, no need to continue sleeping if cancellation has occurred

gRPC has a cancellation callback

- Can use this to cancel non-gRPC operations

Could use a `CancellationToken` from the `Reactor`

Other Resources

Coroutine support libraries

Let me know if I've missed one!

C++23

- std::generator
- Only currently supported in GCC trunk

boost::asio

- Has support for C++20 coroutines
- Quite nice to use
- "Walled Garden" – have to use boost asio all the way

asio-grpc

- https://github.com/Tradias/asio-grpc
- Integrates gRPC with boost::asio
- Can use asio coroutines with gRPC
- Need to use the async (event loop) gRPC interface
  - Slightly steeper learning curve
  - Slightly lower overhead

boost::cobalt

- Opinionated coroutines library
- Single threaded async
- "Open" – can interoperate with other awaitables
- Lots of examples

cppcoro

- https://github.com/andreasbuhr/cppcoro
  - Maintained fork of https://github.com/lewissbaker/cppcoro
- Experimental
- Relatively low-level
- Lots of vocabulary types

libunifex

- Implementation of Unified Executors proposal
- Includes some coroutine bits
- Experimental

Other Resources

Articles

Let me know what you have found helpful!

Lewis Baker – Asymmetric Transfer

- https://lewissbaker.github.io/
- Very in-depth articles about all aspects of coroutines
- Highly recommended
- Original author of **cppcoro**, and many

Raymond Chen – Coroutines Series

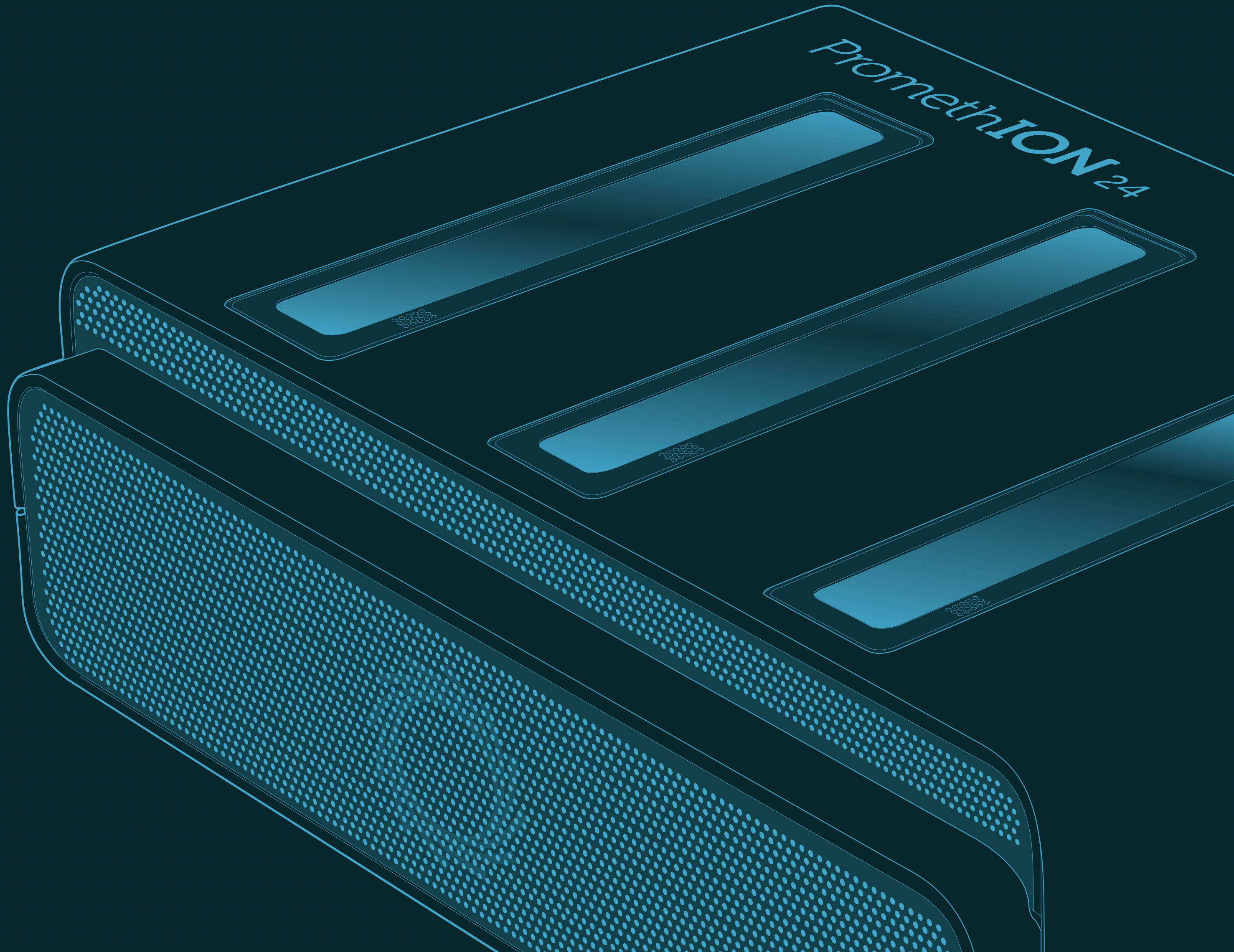
- https://devblogs.microsoft.com/oldnewthing/20210504-01/?p=105178
- Part of his “Old New Thing” blog
- Lots of shorter articles

cppreference.com

- https://en.cppreference.com/w/cpp/language/coroutines
- More for reference than learning

Thank you

Bonus!



Future Directions

Abstract “completion”

Abstract away concept of “wait for a value to be produced”

- Simplify the code in the Reactor
- e.g. `AwaitablePromise<T>/AwaitableFuture<T>`
- Can `co_await` an `AwaitableFuture`
  - Suspend coroutine until promise has value/exception set

This kind of abstraction is present in most coroutine libraries

Completion tokens/Completion signals

- Even more abstract
- Effectively, “call some code when the operation completes”

These kind of abstractions are a fundamental part of boost::asio and the Unified Executors proposal

```
struct Reactor {
    AwaitableFuture<void> start_write(
        Response const & response
    ) {
        m_write_promise = AwaitablePromise<void>{};
        StartWrite(&response);
        return m_write_promise.get_future();
    }

    void OnWriteDone(bool ok) override {
        if (ok) {
            m_write_promise.set_value();
        } else {
            m_write_promise.set_exception(
                std::make_exception_ptr(Cancelled{})
            );
        }
    }

    AwaitablePromise<void> m_write_promise;
};

Reactor reactor;
co_await reactor.start_write(response);
```

Future Directions

Getting the reactor in the coroutine body

Make the reactor available in the gRPC coroutine

Allows a top-level gRPC coroutine, and also passing the reactor to sub-functions, other tasks, etc.

```
// Desired usage
auto & reactor = co_await get_reactor();
```

Can be used inside task-type coroutines to get the executor that the coroutine is running on

```
struct GetReactorAwaiter {
    constexpr bool await_ready() noexcept {
        return false;
    }

    template <typename T>
    bool await_suspend(
        std::coroutine_handle<T> coroutine_handle
    ) {
        m_reactor =
            coroutine_handle.promise().m_reactor;
        return false; // Continue executing coroutine
    }

    Reactor & await_resume() noexcept {
        return *m_reactor;
    }

    Reactor * m_reactor;
}

constexpr GetReactorAwaiter get_reactor() noexcept {
    return {};
}
```

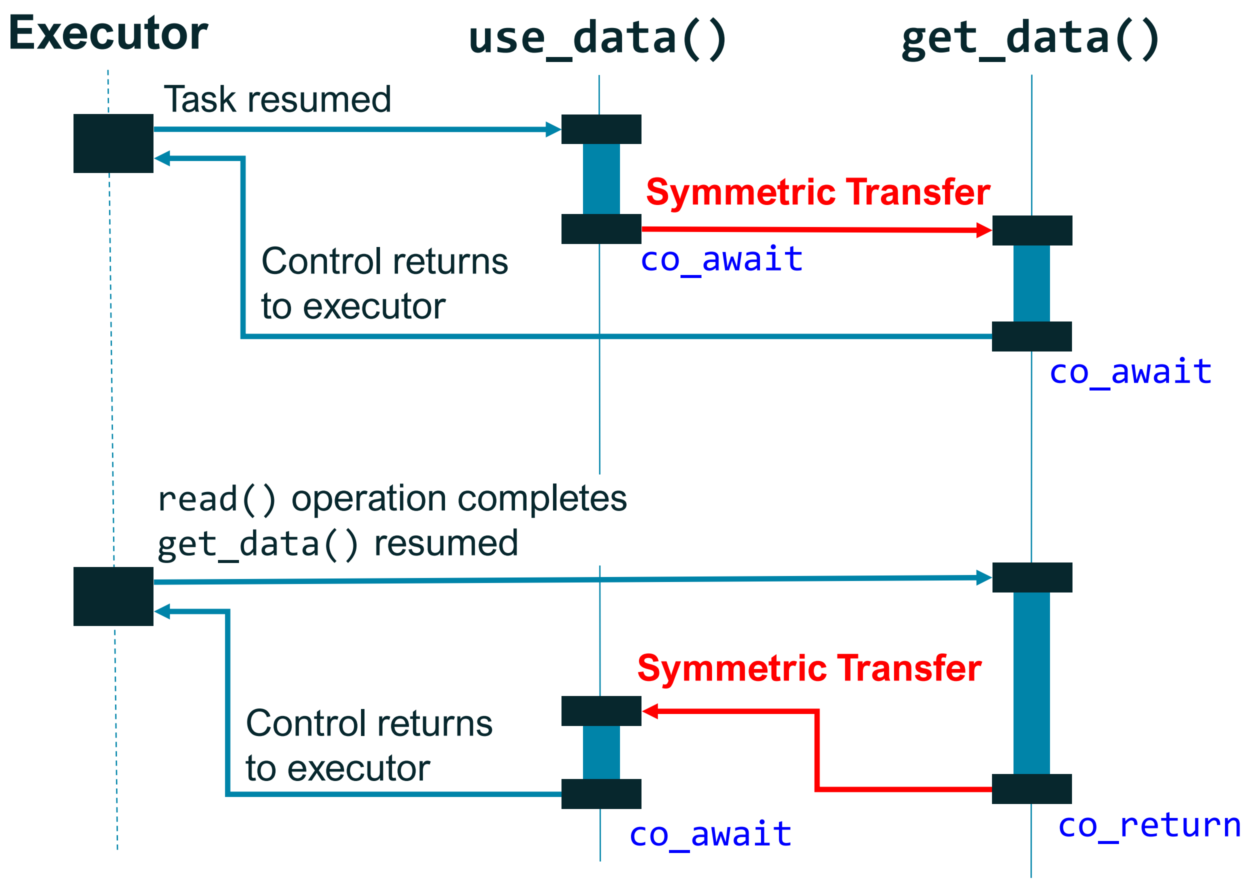
Future Directions

Implemented in most coroutine support libraries

Nesting coroutines

```
task use_data() {
     auto data = co_await get_data();
     // Another co_await here
}

task get_data() {
     auto data = co_await read();
     co_return data;
}
```



Huge topic!

"Symmetric transfer" is key in allowing this to be efficient

- Switch between coroutines with minimal overhead

https://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer

Future Directions

Library Support: Waiting on multiple awaitables

Similar facilities are available in most coroutine support libraries

Wait for any

```
// The variant holds the result of the awaitable that completed
// All other awaitables are cancelled
std::variant<std::monostate, Request> = co_await send(response) || receive();

// Return an awaitable for awaitables that did not finish
std::tuple<AwaitableOrResult<void>, AwaitableOrResult<Request>> =
    co_await wait_for_any(send(response), receive());
```

Operator overloading for awaitables!

Wait for all

```
// If any awaitable fails with an exception, all awaitables are cancelled
std::tuple<std::monostate, Request> = co_await send(response) && receive();

// Store the exception for awaitables that finished with an exception
std::tuple<ResultOrException<void>, ResultOrException<Request>> =
    co_await wait_for_all(send(response) && receive());
```