Coroutines - State Of Mind Or State Machine

Created by Assaf Cohen and Avi Lachmish

Agenda

- What Is Coroutine?
  - Technical
  - A way to look at a Coroutine
- Our Challenge
- Examples
  - File Transfers
- WrapUp

Attention

Attention

Code samples are abbreviated to fit in slides

What Is Coroutine?

A coroutine is a function that can suspend execution to be resumed later.
Coroutines are stackless: they suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack.
This allows for sequential code that executes asynchronously (e.g. to handle non-blocking I/O without explicit callbacks), and also supports algorithms on lazy-computed infinite sequences and other uses.

from cppreference-coroutines.

What Is Coroutine?

compiler then generates code

A coroutine has at least one of :

What Is Coroutine?

compiler then generates code

A coroutine has at least one of :

co_await

unary operator that suspends a coroutine and returns control to the caller.

What Is Coroutine?

compiler then generates code

A coroutine has at least one of :

co_return

completes execution returning a value

What Is Coroutine?

compiler then generates code

A coroutine has at least one of :

co_yield

returns a value to the caller and suspends the current coroutine: it is the common building block of resume-able generator

What Is Coroutine?

Awaitable

A coroutine is a function that can suspend execution to be resumed later.

What Is Coroutine?

Awaitable

A coroutine is a function that can suspend execution to be resumed later.

```
bool await_ready()
void / bool await_suspend(std::coroutine_handle<> h)
void await_resume()
```

What Is Coroutine?

Awaitable

A coroutine is a function that can suspend execution to be resumed later.

```cpp
// short-cut to avoid the cost of suspension
// if true it's known that the result is ready
// or can be completed synchronously
bool await_ready()
void / bool await_suspend(std::coroutine_handle<> h)
void await_resume()
```

What Is Coroutine?

Awaitable

A coroutine is a function that can suspend execution to be resumed later.

```
bool await_ready()
// * h is the current coroutine std::coroutine_handle
//
// * if it returns void control returns to caller
//   and coroutine is suspended
//
// * if it returns bool with true control returns to caller
//   and coroutine is suspended
//
// * if it returns bool with false the coroutine is resumed
void / bool await_suspend(std::coroutine_handle<> h)
void await_resume()
```

What Is Coroutine?

Awaitable

A coroutine is a function that can suspend execution to be resumed later.

```
bool await_ready()
void / bool await_suspend(std::coroutine_handle<> h)
// called (whether the coroutine was suspended or not),
// and its result is the result of the whole co_await expr expre
// If the coroutine was suspended in the co_await expression,
// and is later resumed,
// the resume point is immediately before the call to await_resu
void await_resume()
```

What Is Coroutine?

Awaitable - Contd

Two awaitables in std

- std::suspend_never -
  await_ready - Always returns true, indicating that an await expression never suspends.
- std::suspend_always -
  await_ready - Always returns false, indicating that an await expression always suspends.

Basic Coroutine

Basic Coroutine

```
#include <iostream>
#include <string_view>
#include <source_location>
#include <coroutine>

void doLog(const std::source_location loc=std::source_location

struct Operation {

  struct promise_type {
    Operation get_return_object()                          {    doLog();
    std::suspend_never initial_suspend() noexcept {    doLog();
    std::suspend_never final_suspend() noexcept   {    doLog();
    void return_void()                                     {    doLog();
    void unhandled_exception()                             {    doLog();
    promise_type()                                         {    doLog();
    ~promise_type()                                        {    doLog();
  };
```

Basic Coroutine

```
#include <iostream>
#include <string_view>
#include <source_location>
#include <coroutine>

void doLog(const std::source_location loc=std::source_location

struct Operation {

  struct promise_type {
    Operation get_return_object()                    {   doLog();
    std::suspend_never initial_suspend() noexcept {   doLog();
    std::suspend_never final_suspend() noexcept   {   doLog();
    void return_void()                               {   doLog();
    void unhandled_exception()                       {   doLog();
    promise_type()                                   {   doLog();
    ~promise_type()                                  {   doLog();
  };
```

Basic Coroutine

```
#include <iostream>
#include <string_view>
#include <source_location>
#include <coroutine>

void doLog(const std::source_location loc=std::source_location

struct Operation {

  struct promise_type {
    Operation get_return_object()                         {   doLog();
    std::suspend_never initial_suspend() noexcept {   doLog();
    std::suspend_never final_suspend() noexcept   {   doLog();
    void return_void()                            {   doLog();
    void unhandled_exception()                    {   doLog();
    promise_type()                                {   doLog();
    ~promise_type()                               {   doLog();
  };
```

Basic Coroutine

```cpp
#include <iostream>
#include <string_view>
#include <source_location>
#include <coroutine>

void doLog(const std::source_location loc=std::source_location

struct Operation {

  struct promise_type {
    Operation get_return_object()                 {   doLog();
    std::suspend_never initial_suspend() noexcept {   doLog();
    std::suspend_never final_suspend() noexcept   {   doLog();
    void return_void()                            {   doLog();
    void unhandled_exception()                    {   doLog();
    promise_type()                                {   doLog();
    ~promise_type()                               {   doLog();
  };
```

Basic Coroutine

```
#include <iostream>
#include <string_view>
#include <source_location>
#include <coroutine>

void doLog(const std::source_location loc=std::source_location

struct Operation {

  struct promise_type {
    Operation get_return_object()                       {   doLog();
    std::suspend_never initial_suspend() noexcept {   doLog();
    std::suspend_never final_suspend() noexcept   {   doLog();
    void return_void()                                  {   doLog();
    void unhandled_exception()                          {   doLog();
    promise_type()                                      {   doLog();
    ~promise_type()                                     {   doLog();
  };
```

Basic Coroutine

```
#include <iostream>
#include <string_view>
#include <source_location>
#include <coroutine>

void doLog(const std::source_location loc=std::source_location

struct Operation {

  struct promise_type {
    Operation get_return_object()                       {   doLog();
    std::suspend_never initial_suspend() noexcept {   doLog();
    std::suspend_never final_suspend() noexcept   {   doLog();
    void return_void()                                  {   doLog();
    void unhandled_exception()                          {   doLog();
    promise_type()                                      {   doLog();
    ~promise_type()                                     {   doLog();
  };
```

Basic Coroutine

```
#include <iostream>
#include <string_view>
#include <source_location>
#include <coroutine>

void doLog(const std::source_location loc=std::source_location

struct Operation {

  struct promise_type {
    Operation get_return_object()                  {   doLog()
    std::suspend_always initial_suspend() noexcept {   doLog()
    std::suspend_never final_suspend() noexcept    {   doLog()
    void return_void()                             {   doLog()
    void unhandled_exception()                     {   doLog()
    promise_type()                                 {   doLog()
    ~promise_type()                                {   doLog()
  };
```

Basic Coroutine

```cpp
void doLog(const std::source_location loc=std::source_location

struct Operation {

  struct promise_type {
    Operation get_return_object()                   {    doLog()
    std::suspend_always initial_suspend() noexcept {    doLog()
    std::suspend_never final_suspend() noexcept    {    doLog()
    void return_void()                              {    doLog()
    void unhandled_exception()                      {    doLog()
    promise_type()                                  {    doLog()
    ~promise_type()                                 {    doLog()
  };
  Operation()  {    doLog();   }
  ~Operation() {    doLog();   }
};

Operation emptyCoroutine() {
  std::cerr << "Inside coroutine.\n";
```

std::coroutine_handle

Refers To A Suspended Or Executing Coroutine

- control - destroy, resume
- observe - done?, is coroutine?
- create from promise
- and more … - out of scope

Basic Coroutine - Fixed

```cpp
struct Operation {
  struct promise_type {
    using Handle = std::coroutine_handle<promise_type>;
    Operation get_return_object()               {   dodoLog();
    std::suspend_always initial_suspend()       {   dodoLog();
    std::suspend_never final_suspend() noexcept {   dodoLog();
    void return_void()                          {   dodoLog();
    void unhandled_exception()                  {   dodoLog();
    promise_type()                              {   dodoLog();
    ~promise_type()                             {   dodoLog();
  };

  explicit Operation(promise_type::Handle coro) : coro_(coro)
  ~Operation() {
    dodoLog();
    if (coro_ && !coro_.done()) {   coro_.destroy();    }
  }
```

Basic Coroutine - Fixed

```
    ~promise_type()                                {   dodoLog();
  };

  explicit Operation(promise_type::Handle coro) : coro_(coro)
  ~Operation() {
    dodoLog();
    if (coro_ && !coro_.done()) {   coro_.destroy();    }
  }

  void destroy()  {   dodoLog();  coro_.destroy();    }
  void resume()   {   dodoLog();  coro_.resume();     }

  private:
      promise_type::Handle coro_;
};

Operation emptyCoroutine() {
  std::cerr << "Inside coroutine.\n";
  co_return;
```

Basic Coroutine - Fixed

```
struct Operation {
  struct promise_type {
    using Handle = std::coroutine_handle<promise_type>;
    Operation get_return_object()                 {   dodoLog();
    std::suspend_always initial_suspend()         {   dodoLog();
    std::suspend_never final_suspend() noexcept {   dodoLog();
    void return_void()                            {   dodoLog();
    void unhandled_exception()                    {   dodoLog();
    promise_type()                                {   dodoLog();
    ~promise_type()                               {   dodoLog();
  };

  explicit Operation(promise_type::Handle coro) : coro_(coro)
  ~Operation() {
    dodoLog();
    if (coro_ && !coro_.done()) {   coro_.destroy();    }
  }
```

Basic Coroutine - Fixed

```
      if (coro_ && !coro_.done()) {   coro_.destroy();     }
    }

    void destroy()  {   dodoLog();  coro_.destroy();     }
    void resume()   {   dodoLog();  coro_.resume();      }

    private:
        promise_type::Handle coro_;
};

Operation emptyCoroutine() {
    std::cerr << "Inside coroutine.\n";
    co_return;
}

std::int32_t main() {
      std::cerr << "Before coroutine\n";
      auto c = emptyCoroutine();
      std::cerr << "After call, before resume.\n";
```

Basic Coroutine - Fixed

```
  private:
      promise_type::Handle coro_;
};

Operation emptyCoroutine() {
  std::cerr << "Inside coroutine.\n";
  co_return;
}

std::int32_t main() {
    std::cerr << "Before coroutine\n";
    auto c = emptyCoroutine();
    std::cerr << "After call, before resume.\n";
    c.resume();
    std::cerr << "After coroutine\n";
  return 0;
}

void dodoLog(const std::source location location) {
```

Basic Coroutine - Fixed

```

Operation emptyCoroutine() {
  std::cerr << "Inside coroutine.\n";
  co_return;
}

std::int32_t main() {
    std::cerr << "Before coroutine\n";
    auto c = emptyCoroutine();
    std::cerr << "After call, before resume.\n";
    c.resume();
    std::cerr << "After coroutine\n";
  return 0;
}

void dodoLog(const std::source_location location) {
  std::cerr << location.function_name() <<"\n";
};
```

Basic Coroutine - Fixed

```cpp

Operation emptyCoroutine() {
  std::cerr << "Inside coroutine.\n";
  co_return;
}

std::int32_t main() {
    std::cerr << "Before coroutine\n";
    auto c = emptyCoroutine();
    std::cerr << "After call, before resume.\n";
    c.resume();
    std::cerr << "After coroutine\n";
  return 0;
}

void dodoLog(const std::source_location location) {
  std::cerr << location.function_name() <<"\n";
};
```

Our Challenge

overview

grid of nodes. sharing data, and passing messages.

Our Challenge

characteristics

I/O bound

Our Challenge

coroutines seems like a valid solution

Coroutine Is Not Async, But...

Coroutine Is Not Async, But...

Can Be Used For Building Frameworks For Asynchronous Operations Without Callbacks

Coroutine Is Not Async, But...

Can Be Used For Building Frameworks For Asynchronous Operations Without Callbacks

That Is What We Used Coroutines For In Incredibuild

using Asio

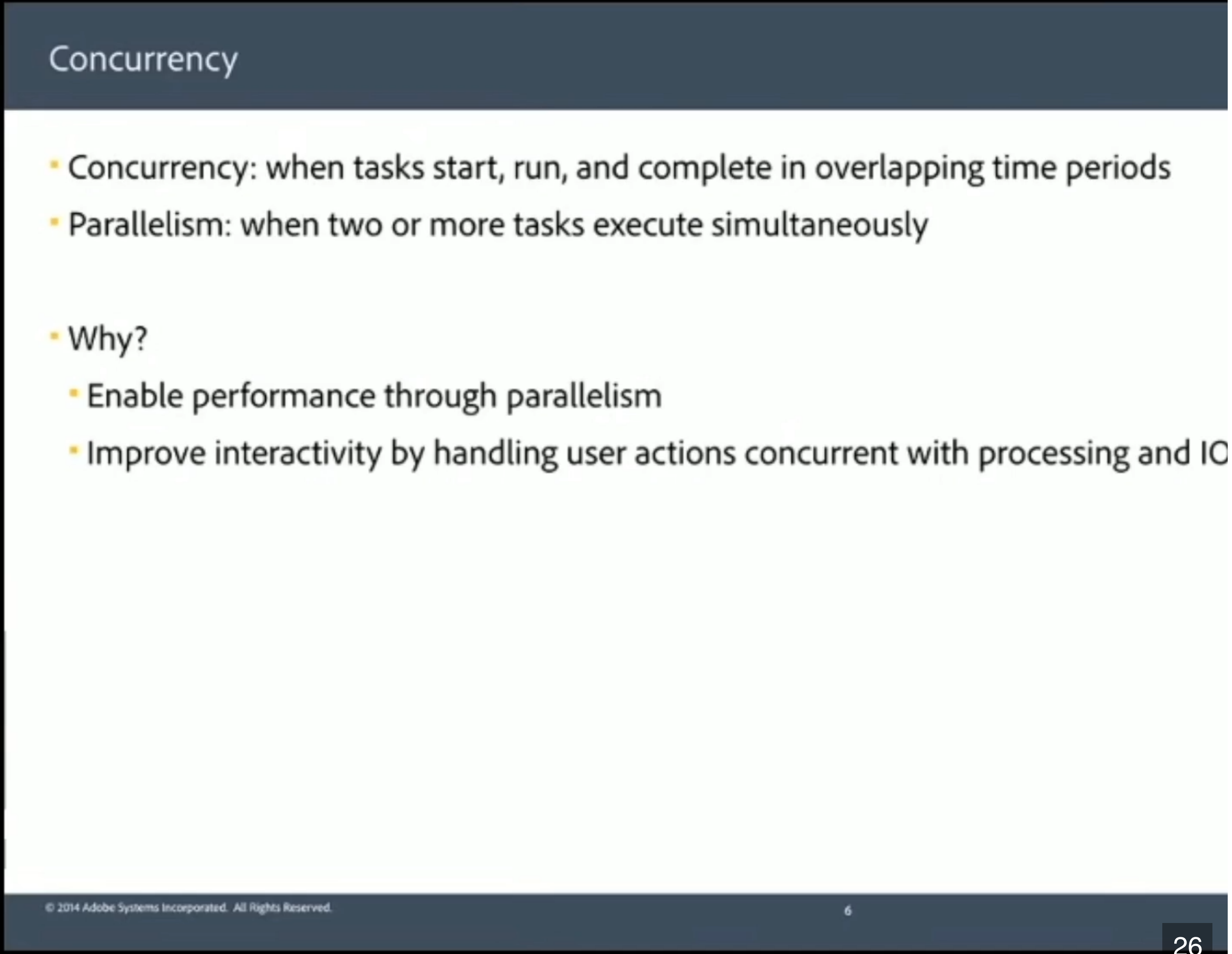
other libraries are out there:

cppcoro, hpx, folly ...

Motivation

NDC { London }
16-20 January 2017

Inspiring Software Developers
since 2008



Concurrency

- Concurrency: when tasks start, run, and complete in overlapping time periods
- Parallelism: when two or more tasks execute simultaneously

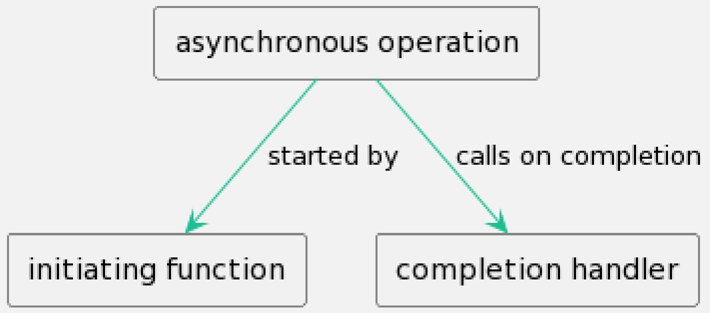
- Why?
  - Enable performance through parallelism
  - Improve interactivity by handling user actions concurrent with processing and IO

ASIO - Structured Concurrency - Overview

ASIO - Structured Concurrency - Overview

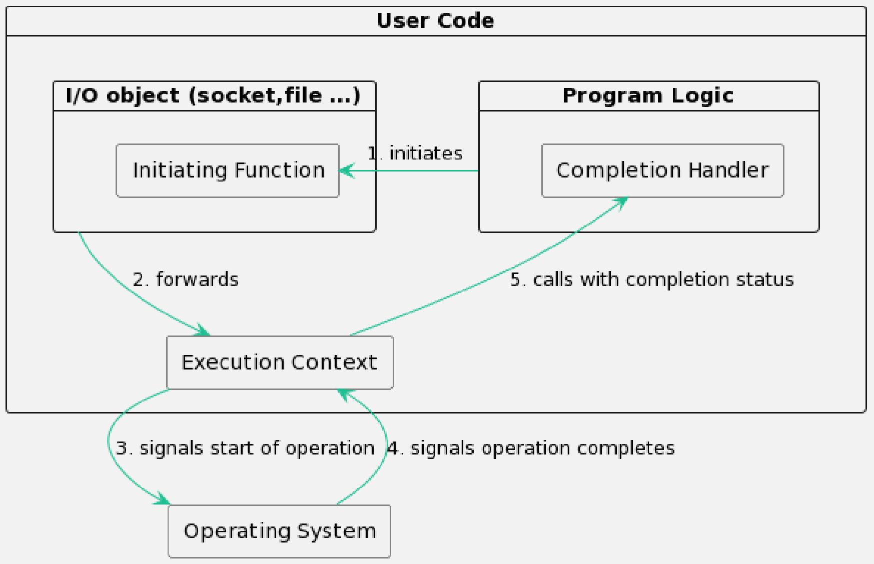
asynchronous operation

work that is launched and performed in the background



ASIO - Structured Concurrency - Overview

asynchronous model


User Code
I/O object (socket,file ...)
Initiating Function
Program Logic
Completion Handler
1. initiates
2. forwards
5. calls with completion status
Execution Context
3. signals start of operation
4. signals operation completes
Operating System

ASIO - Structured Concurrency - Overview

completion token

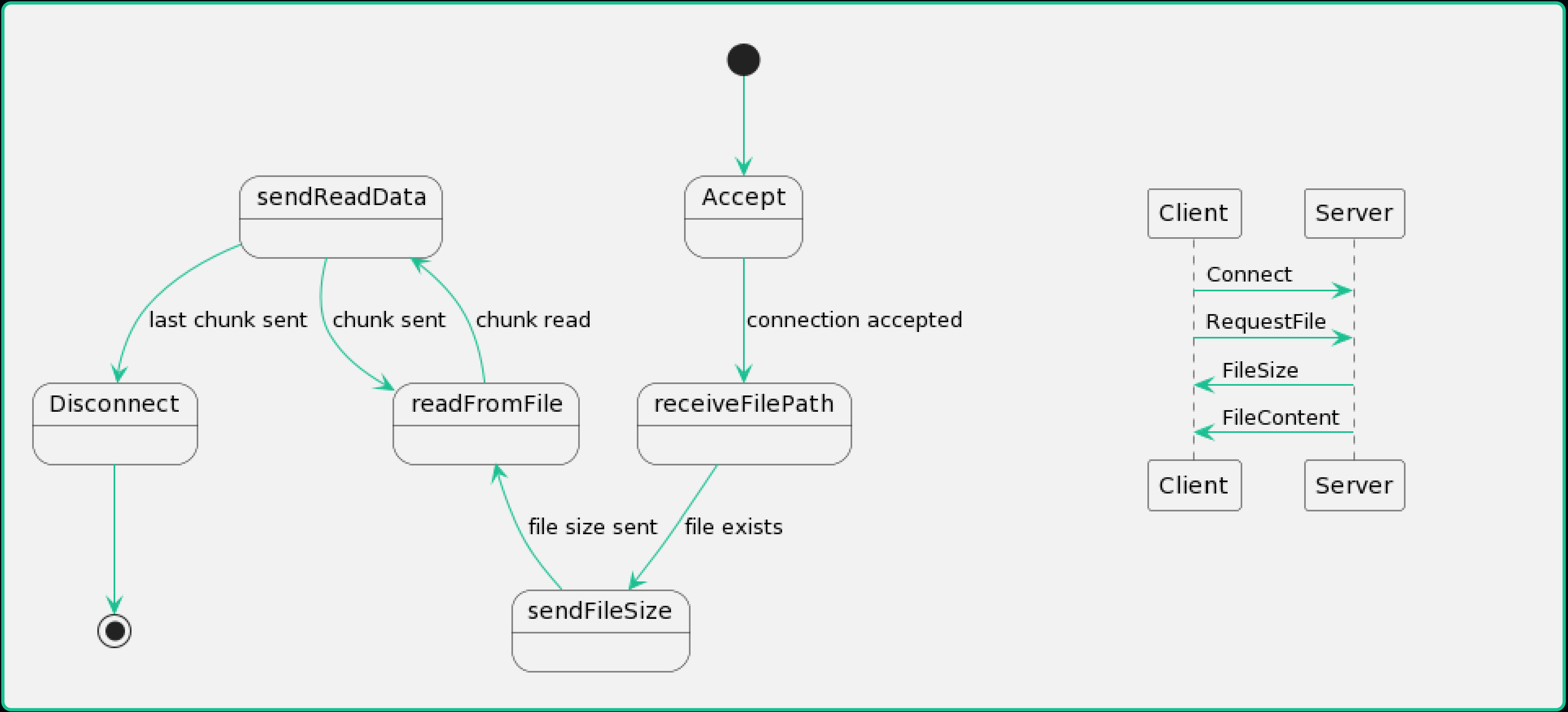
- lambda
- function object
- use_future
- use_awaitable
- ...

A Real World

Yet Shortened

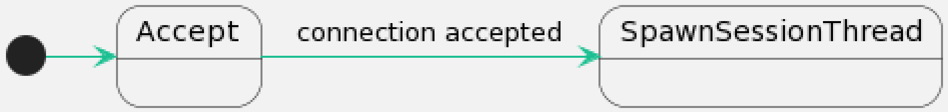
Example

Sending A File



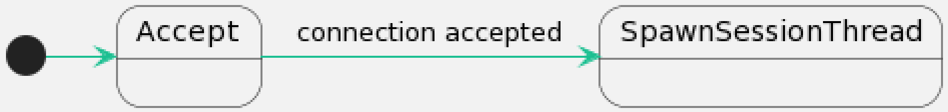
Blocking File Send - Thread Per Connection - Main

```cpp
#include <fmt/core.h>
#include <boost/bind/bind.hpp>
#include <boost/asio.hpp>

#include "FileSender.hpp"

using namespace corecpp2022;
using boost::asio::ip::tcp;

void session(tcp::socket sock)
{
  try
  {
    char fileName[256];
```



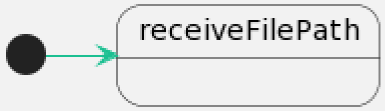
Blocking File Send - Thread Per Connection - Accept

```cpp
#include <fmt/core.h>
#include <boost/bind/bind.hpp>
#include <boost/asio.hpp>

#include "FileSender.hpp"

using namespace corecpp2022;
using boost::asio::ip::tcp;

void session(tcp::socket sock)
{
  try
  {
    char fileName[256];
```



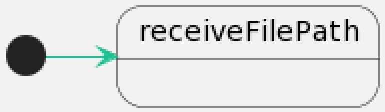
Blocking File Send - Receive File Path

```cpp
#include <fmt/core.h>
#include <boost/bind/bind.hpp>
#include <boost/asio.hpp>

#include "FileSender.hpp"

using namespace corecpp2022;
using boost::asio::ip::tcp;

void session(tcp::socket sock)
{
  try
  {
    char fileName[256];
```



Blocking File Send - Receive File Path

```
            sock.read_some(boost::asio::buffer(fileName, sizeof
        FileSender fileSender(sock);
        fileSender.sendFileSize(std::string_view(fileName, bytes
    }
    catch (std::exception& e)
    {
        fmt::print("Exception in thread: {}\n", e.what());
    }
}

[[noreturn]] void server(boost::asio::io_context& io_context
{
    tcp::acceptor acceptor(io_context, tcp::endpoint(tcp::v4()
    for (;;)
    {
```



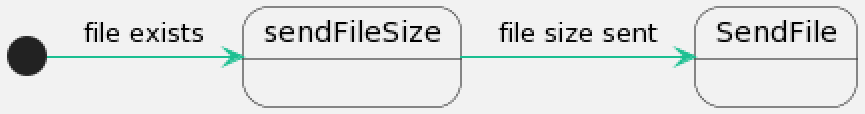
Blocking File Send - Receive File Path

```
  {
    fmt::print("Exception in thread: {}\n", e.what());
  }
}

[[noreturn]] void server(boost::asio::io_context& io_context
{
  tcp::acceptor acceptor(io_context, tcp::endpoint(tcp::v4()
  for (;;)
  {
    std::thread(session, acceptor.accept()).detach();
  }
}

std::int32_t main() noexcept
```

receiveFilePath

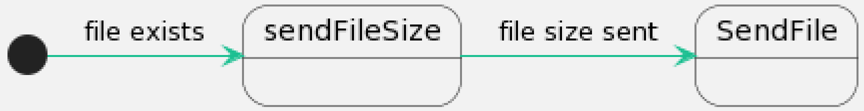
Blocking File Send - Send File Size

```cpp
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
using fs::path;
using std::size_t;

static constexpr size_t buffSize = 1024 * 64;
```



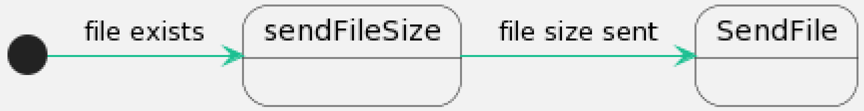
Blocking File Send - Send File Size

```cpp
void FileSender::sendFile(const path& filePath)
{
  const auto fileSize = static_cast<size_t>(fs::file_size(fi
  auto buff = std::make_unique<std::byte[]>(buffSize);
  asio::stream_file file(mStream.get_executor(),
                         filePath.string(),
                         asio::file_base::read_only);
  size_t totalreadBytes = 0, totalsentBytes = 0, readBytes =
  while (totalreadBytes < fileSize)
  {
    totalreadBytes += readBytes =
        file.read_some(asio::buffer(buff.get(), buffSize));
    totalsentBytes +=
        asio::write(mStream, asio::buffer(buff.get(), readBy
  }
```



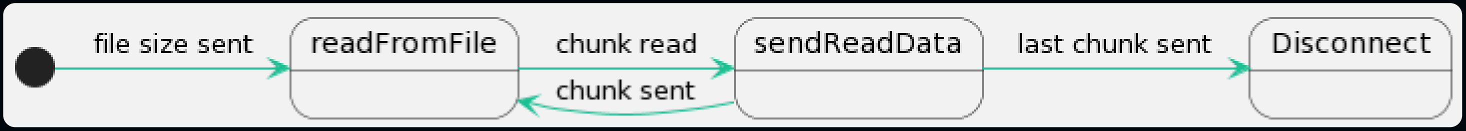
Blocking File Send - Send File Size

```cpp
  const auto fileSize = static_cast<size_t>(fs::file_size(fi
  auto buff = std::make_unique<std::byte[]>(buffSize);
  asio::stream_file file(mStream.get_executor(),
                          filePath.string(),
                          asio::file_base::read_only);
  size_t totalreadBytes = 0, totalsentBytes = 0, readBytes =
  while (totalreadBytes < fileSize)
  {
    totalreadBytes += readBytes =
        file.read_some(asio::buffer(buff.get(), buffSize));
    totalsentBytes +=
        asio::write(mStream, asio::buffer(buff.get(), readBy
  }
}
} // namespace corecpp2022
```



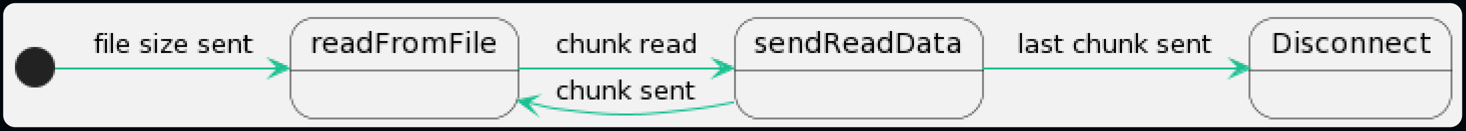
Blocking File Send - Send File Content

```cpp
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
using fs::path;
using std::size_t;

static constexpr size_t buffSize = 1024 * 64;
```



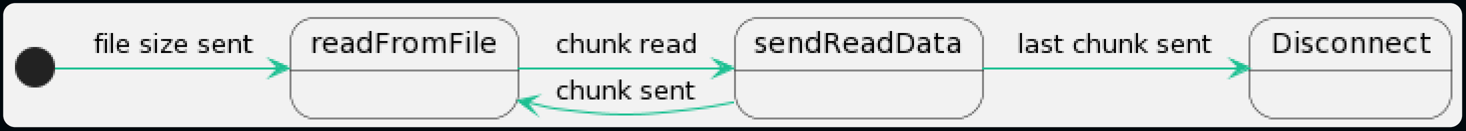
Blocking File Send - Send File Content

```
    auto buff = std::make_unique<std::byte[]>(buffSize);
    asio::stream_file file(mStream.get_executor(),
                           filePath.string(),
                           asio::file_base::read_only);
    size_t totalreadBytes = 0, totalsentBytes = 0, readBytes =
    while (totalreadBytes < fileSize)
    {
      totalreadBytes += readBytes =
          file.read_some(asio::buffer(buff.get(), buffSize));
      totalsentBytes +=
          asio::write(mStream, asio::buffer(buff.get(), readBy
    }
  }
  } // namespace corecpp2022
```



Blocking File Send - Send File Content

```cpp
  auto buff = std::make_unique<std::byte[]>(buffSize);
  asio::stream_file file(mStream.get_executor(),
                      filePath.string(),
                      asio::file_base::read_only);
  size_t totalreadBytes = 0, totalsentBytes = 0, readBytes =
  while (totalreadBytes < fileSize)
  {
    totalreadBytes += readBytes =
        file.read_some(asio::buffer(buff.get(), buffSize));
    totalsentBytes +=
        asio::write(mStream, asio::buffer(buff.get(), readBy
  }
}
} // namespace corecpp2022
```

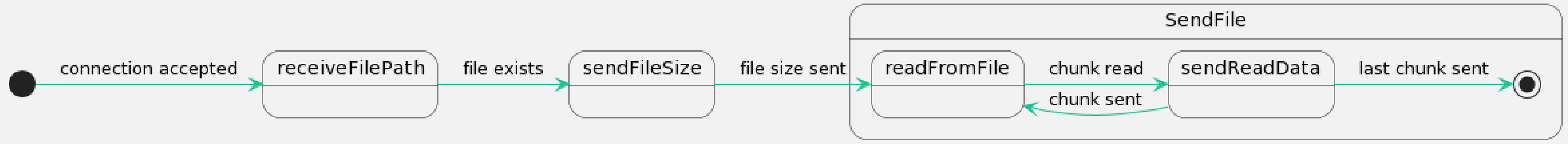


Async File Send - Callbacks - FileSender Layout

```
#pragma once

#include <filesystem>
#include <boost/asio.hpp>

namespace corecpp2022
{
class FileSender : public std::enable_shared_from_this<FileS
{
```

Async File Send - Callbacks - FileSender Layout

```
#pragma once

#include <filesystem>
#include <boost/asio.hpp>

namespace corecpp2022
{
class FileSender : public std::enable_shared_from_this<FileS
{
```

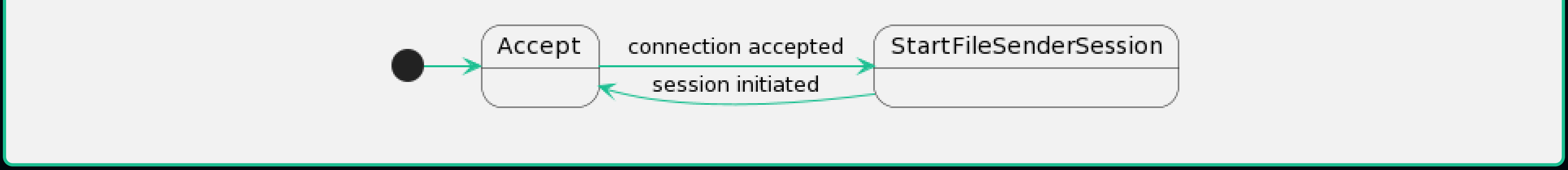


Async File Send - Callbacks - Main

```cpp
#include <fmt/core.h>
#include <boost/asio.hpp>
#include <memory>
#include <utility>
#include "FileSender.hpp"

using namespace corecpp2022;
using boost::asio::ip::tcp;

class Server
{
public:
  Server(boost::asio::io_context& ioContext, std::uint16_t p
      mAcceptor(ioContext, tcp::endpoint(tcp::v4(), port))
  {
    doAccept();
  }

```

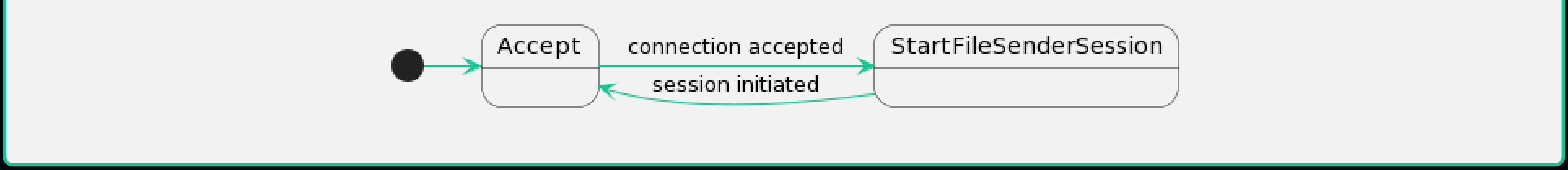
Async File Send - Callbacks - Accept

```cpp
#include <fmt/core.h>
#include <boost/asio.hpp>
#include <memory>
#include <utility>
#include "FileSender.hpp"

using namespace corecpp2022;
using boost::asio::ip::tcp;

class Server
{
public:
  Server(boost::asio::io_context& ioContext, std::uint16_t p
      mAcceptor(ioContext, tcp::endpoint(tcp::v4(), port))
  {
    doAccept();
  }
```


Accept
connection accepted
StartFileSenderSession
session initiated

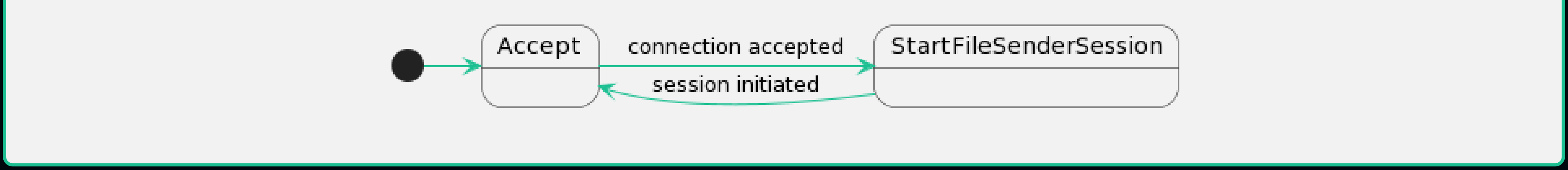
Async File Send - Callbacks - Accept

```
};

std::int32_t main() noexcept
{
  try
  {
    boost::asio::io_context ioContext;

    Server s(ioContext, 2022);

    ioContext.run();
  }
  catch (std::exception& e)
  {

    fmt::print("Exception: {}\n", e.what());
  }
  return 0;
}
```


Accept
connection accepted
session initiated
StartFileSenderSession

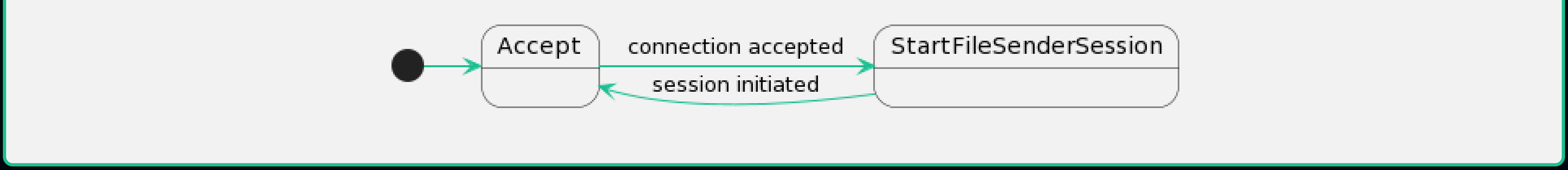
Async File Send - Callbacks - Accept

```

std::int32_t main() noexcept
{
  try
  {
    boost::asio::io_context ioContext;

    Server s(ioContext, 2022);

    ioContext.run();
  }
  catch (std::exception& e)
  {

    fmt::print("Exception: {}\n", e.what());
  }
  return 0;
}
```


Accept
connection accepted
StartFileSenderSession
session initiated

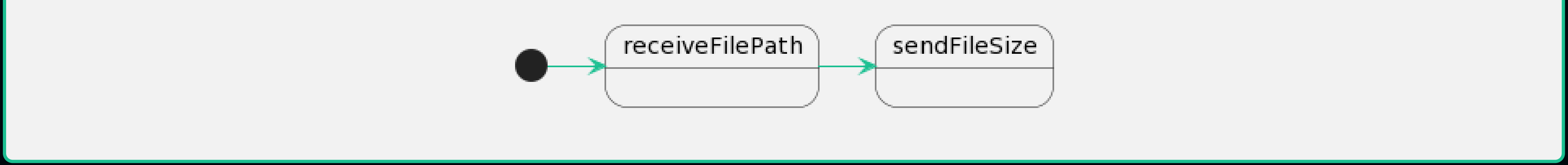
Async File Send - Callbacks - Accept

```cpp

std::int32_t main() noexcept
{
  try
  {
    boost::asio::io_context ioContext;

    Server s(ioContext, 2022);

    ioContext.run();
  }
  catch (std::exception& e)
  {

    fmt::print("Exception: {}\n", e.what());
  }
  return 0;
}
```


Accept
connection accepted
session initiated
StartFileSenderSession

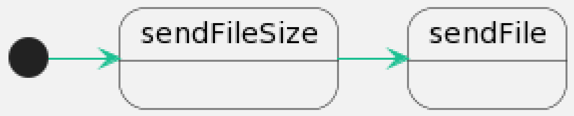
Async File Send - Callbacks - Receive File Path

```cpp
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
using fs::path;
using std::size_t;
static constexpr size_t buffSize = 1024 * 64;

void FileSender::start()
{
  receiveFilePath();
}
void FileSender::receiveFilePath()
```


receiveFilePath
sendFileSize

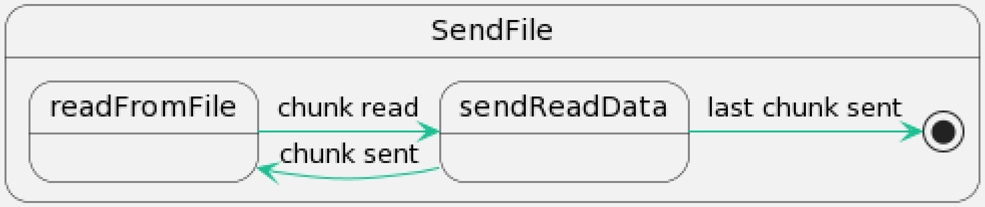
Async File Send - Callbacks - Send File Size

```cpp
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
using fs::path;
using std::size_t;
static constexpr size_t buffSize = 1024 * 64;

void FileSender::start()
```



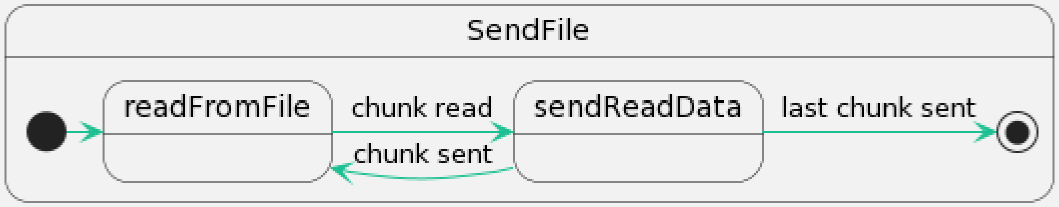
Async File Send - Callbacks - Initiate

```
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
```



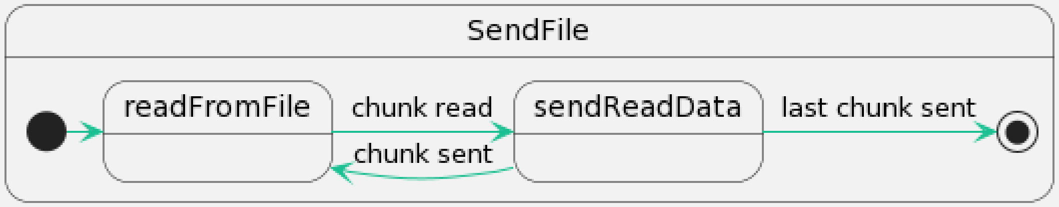
Async File Send - Callbacks - Read From File

```cpp
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
using fs::path;
using std::size_t;
static constexpr size_t buffSize = 1024 * 64;

```



Async File Send - Callbacks - Send Read Data

```cpp
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
using fs::path;
using std::size_t;
static constexpr size_t buffSize = 1024 * 64;

```



Recap

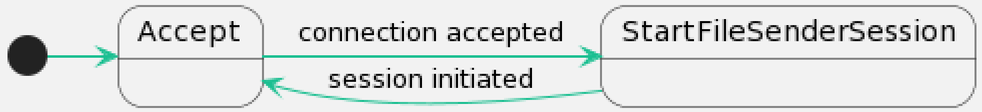
	Callbacks	Blocking
Maintainable	—	—
Readable	—	—
Complexity	—	—
Efficiency	—	—

Recap

	Coroutines
Maintainable	—
Readable	—
Complexity	—
Efficiency	—

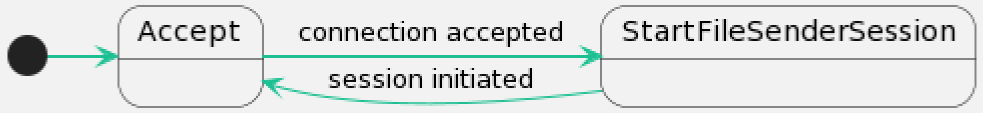
Coroutines File Send - Accept

```cpp
#include <fmt/core.h>
#include <boost/asio.hpp>
#include <boost/asio/experimental/as_tuple.hpp>
#include <memory>
#include <utility>
#include "FileSender.hpp"

using namespace corecpp2022;

using boost::asio::detached;
using boost::asio::awaitable;
using boost::asio::buffer;
using boost::asio::co_spawn;
using boost::asio::ip::tcp;
```



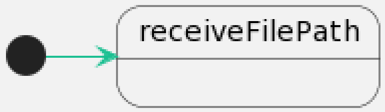
Coroutines File Send - Accept

```
                std::string_view(fileName, bytesRead));
          },
          detached);
    }
    else
    {
      fmt::print("Accept failed: {}\n", e.message());

      steady_timer timer(co_await this_coro::executor);
      timer.expires_after(100ms);
      co_await timer.async_wait(use_awaitable);
    }
  }
}
```



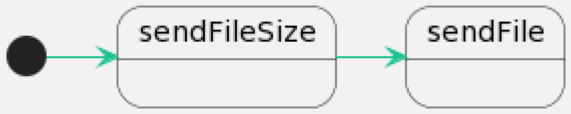
Coroutines File Send - Receive File Path

```cpp
#include <fmt/core.h>
#include <boost/asio.hpp>
#include <boost/asio/experimental/as_tuple.hpp>
#include <memory>
#include <utility>
#include "FileSender.hpp"

using namespace corecpp2022;

using boost::asio::detached;
using boost::asio::awaitable;
using boost::asio::buffer;
using boost::asio::co_spawn;
using boost::asio::ip::tcp;
```



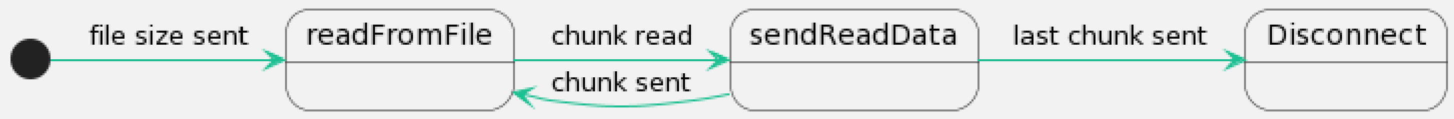
Coroutines File Send - Send File Size

```cpp
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
using fs::path;
using std::size_t;

static constexpr size_t buffSize = 1024 * 64;

asio::awaitable<error_code> FileSender::sendFileSize(const p
{
```



Coroutines File Send - Send File Content

```cpp
#include <fmt/core.h>

#include "FileSender.hpp"

namespace corecpp2022
{
using boost::system::error_code;
namespace fs = std::filesystem;
namespace asio = boost::asio;
using fs::path;
using std::size_t;

static constexpr size_t buffSize = 1024 * 64;

asio::awaitable<error_code> FileSender::sendFileSize(const p
{
```



Code Examples

Coroutines - State Of Mind Or State Machine