accu
2023

APPLIED C++20 COROUTINES

JIM PASCOE

APPLIED C++20 COROUTINES

Jim (James) Pascoe
http://www.james-pascoe.com
james@james-pascoe.com

http://jamespascoe.github.io/accu2023
https://github.com/jamespascoe/accu2023-example-code.git

ACCU Bristol and Bath Meetup Coordinator

COROUTINES ... WHAT NEXT?

1. Fit within the wider concurrency framework
2. More examples (real-world and learning)
3. Empirical measurements
4. Library support and the future

OUTLINE

- Concurrency in Modern C++
  - How C++20 Coroutines fit (and work)
- Mobile Wireless Networking with Coroutines
- C++20 Example: Web Serving with Boost.Beast
  - Asynchronous, Boost.Coroutine, Awaitables
  - Empirical analysis with Apache bench (ab)
- std::generator, std::execution and libunifex

EXAMPLE CODE: TOOLS & BUILD

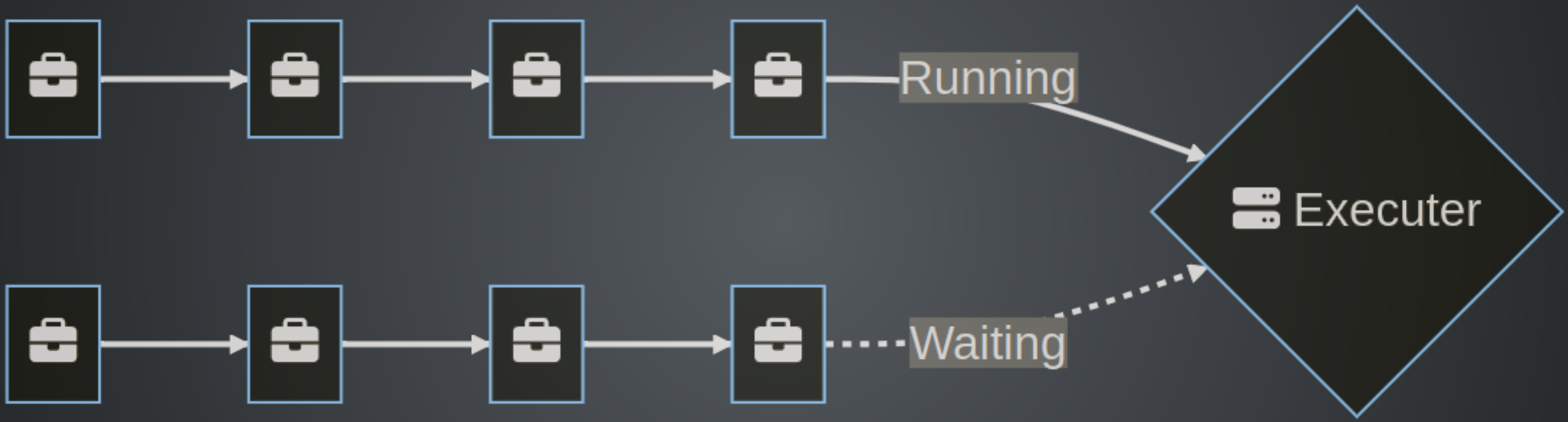
- C++ examples all compile with GCC 12.2:
  - Boost 1.81.0
  - SWIG 4.0.2
  - CMake 3.25.2
- Lua examples run with Lua 5.4.4
- Tested on Linux Mint 19 and Mac OS X (Ventura)

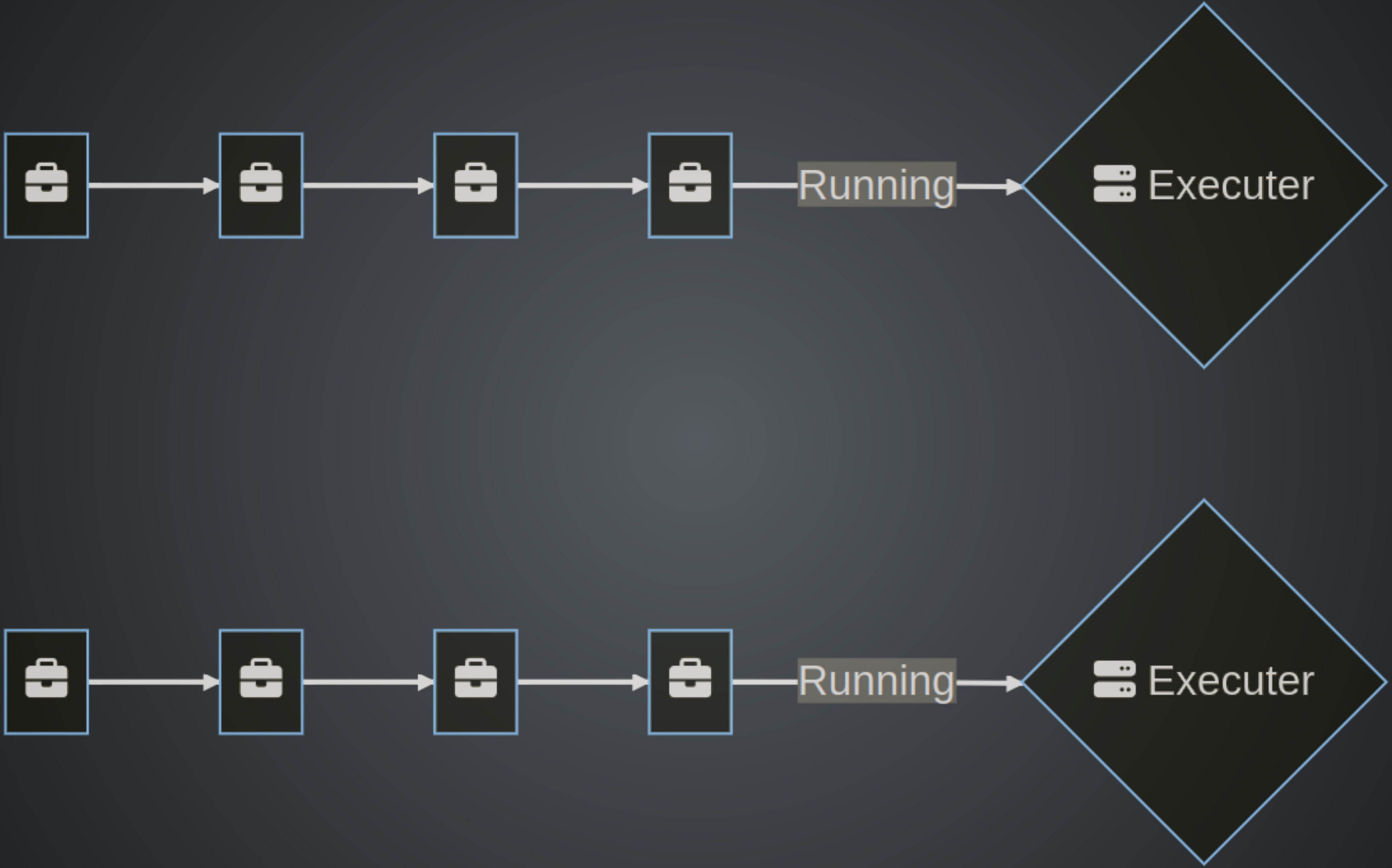
CONCURRENCY BACK-TO-BASICS

CONCURRENCY VS. PARALLELISM

- Concurrency exists when:
  - multiple items of work are 'in progress'
  - e.g. processes, threads or coroutines
  - harnessing windows of latency

- Parallelism exists when:
  - multiple items of work execute simultaneously
  - e.g. threads running on separate CPU cores
  - execution occurs at the same instant in time


Running
Waiting
Executer


Running
Executer
Running
Executer

CONCURRENCY GRANULARITY

1. Multiple processes run on a single computer
2. Multiple threads run within a single process
3. Multiple coroutines run within a single thread

Concurrency allows us to harness latency

PROCESSES

- OS 'multitasks' by forking processes
- Context switch occurs when:
  - a process is blocked (e.g. semaphore)
  - or a pre-emptive time slice expires
- Overhead is high:
  - VM tables, program code, heap, stack, fds, signals
  - Sharing data, synchronisation and scaling are hard

THREADS

- Light-weight threads in a heavy-weight process
- Lower overhead (faster context switch):
  - ... stack, program counter, signal table
- C++03: OS, C++11: std::thread, C++20: std::jthread
- Reentrancy: multiple invocations run concurrently
- Thread safety: the avoidance of race conditions
- Green Threads: scheduled by a runtime library / VM

COROUTINES (AS FIBERS)

- Multiple coroutines in a single thread
- Scheduled by a 'dispatcher' (same thread)
- No races, synchronisation or data sharing issues
- Allows work when part of the thread is blocked
- See Boost.Fiber for details

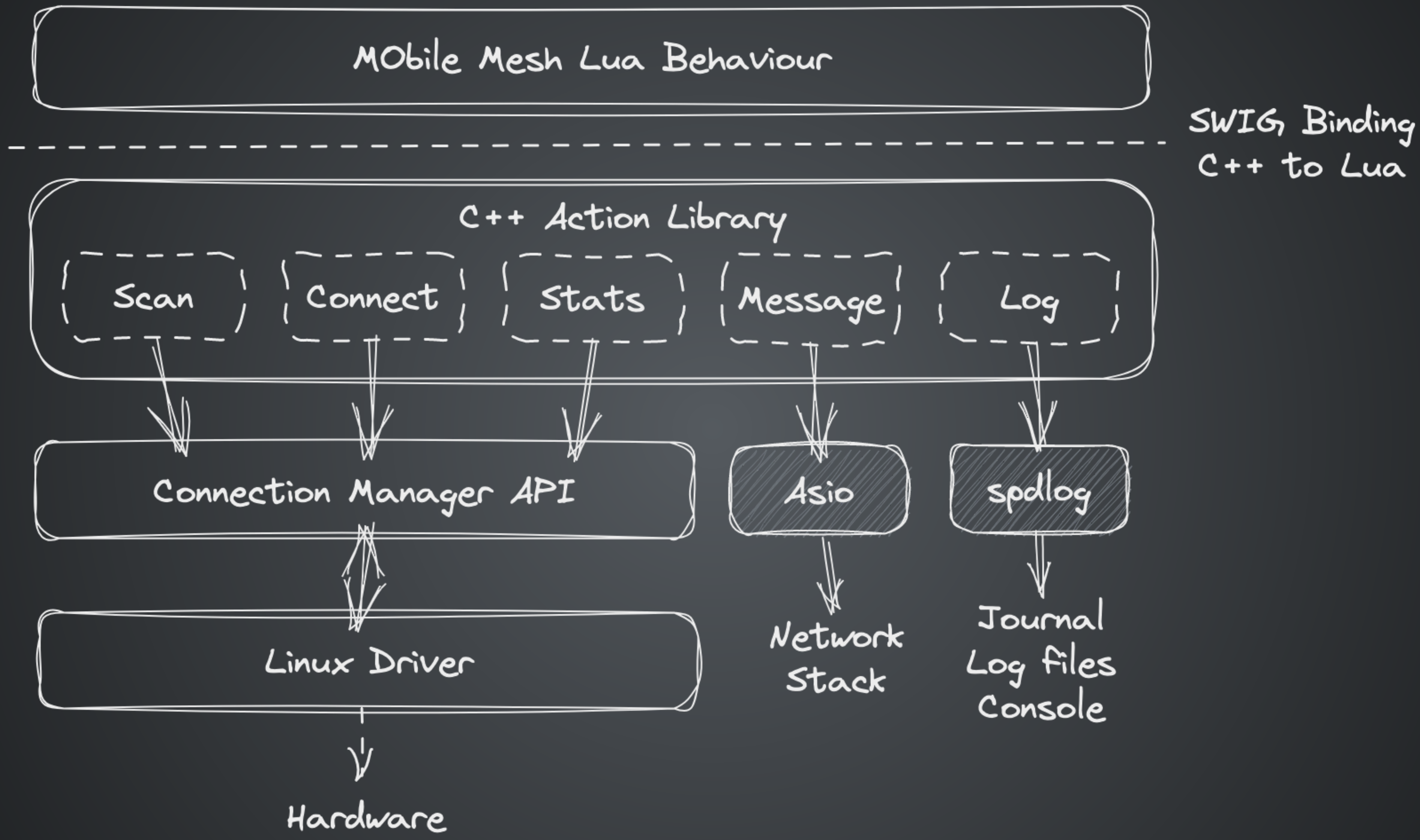
COROUTINES IN THE FIELD

BLU WIRELESS: MOBILE MESH

- IP networking over 5G mmWave (60 GHz) modems
  - 802.11ad MAC + PHY (Hydra) + software
- High-bandwidth, low latency mobile Internet
  - Up to 3 Gbps wireless links (up to 4 km)
- Embedded quad-core ARMv8 NPUs

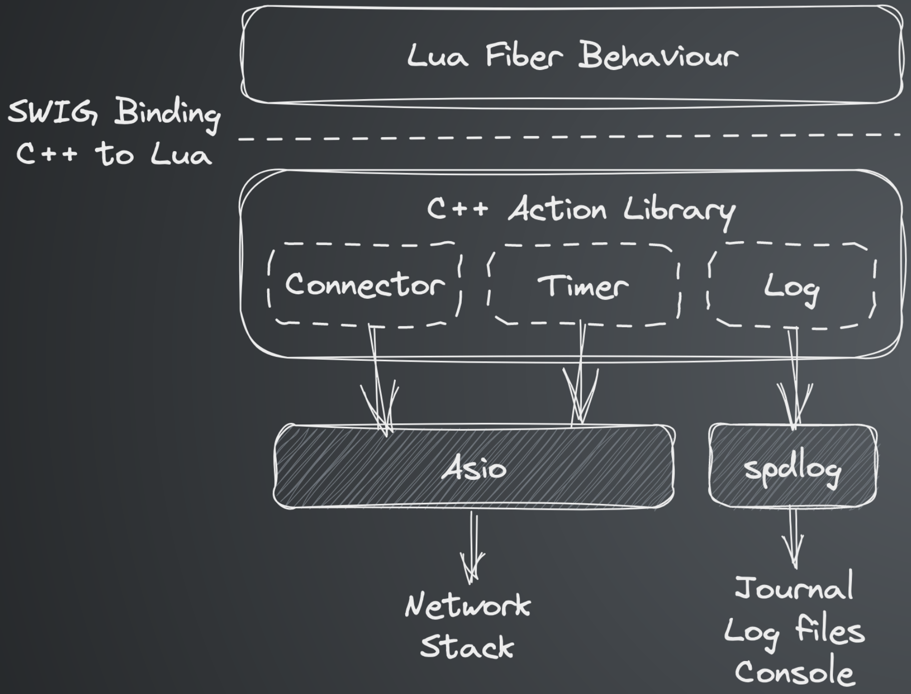
MOBILE CONNECTION MANAGEMENT

- L1 management implemented using coroutines
  - Combination of Modern C++ (17/20) and Lua
- Lots of asynchronous operations
  - Scan, Connect, Disconnect
  - Around 40 primitives (called 'Actions')
- Groups of coroutines operate in threads
  - No race conditions or data sharing limitations
  - Concurrency combined with Parallelism


MObile Mesh Lua Behaviour
SWIG Binding
C++ to Lua
C++ Action Library
Scan
Connect
Stats
Message
Log
Connection Manager API
Asio
spdlog
Linux Driver
Network
Stack
Journal
Log files
Console
Hardware

EXAMPLE: LUA NETWORKING FIBERS

- Lua behaviour: two nodes sending messages
- Includes three actions: 'Connector', 'Timer' and 'Log'
- Also provides: SWIG, CMake, Lua 'main' code
- Other bindings exist: e.g. the PhD's Sol3



Lua Fiber Behaviour

SWIG Binding
C++ to Lua

C++ Action Library

Connector

Timer

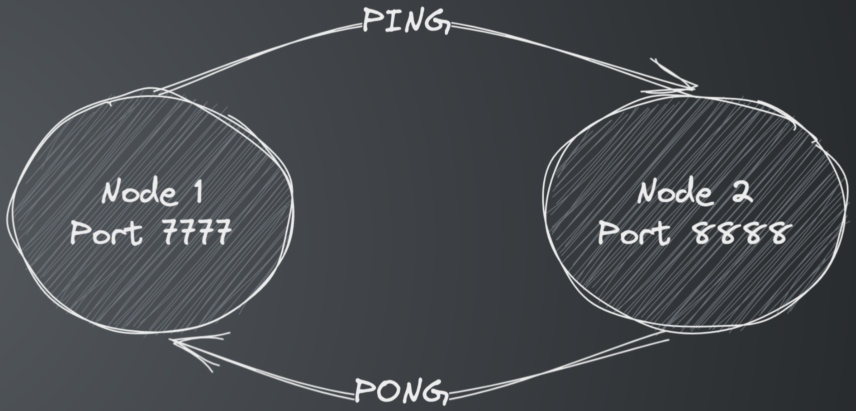
Log

Asio

spdlog

Network
Stack

Journal
Log files
Console



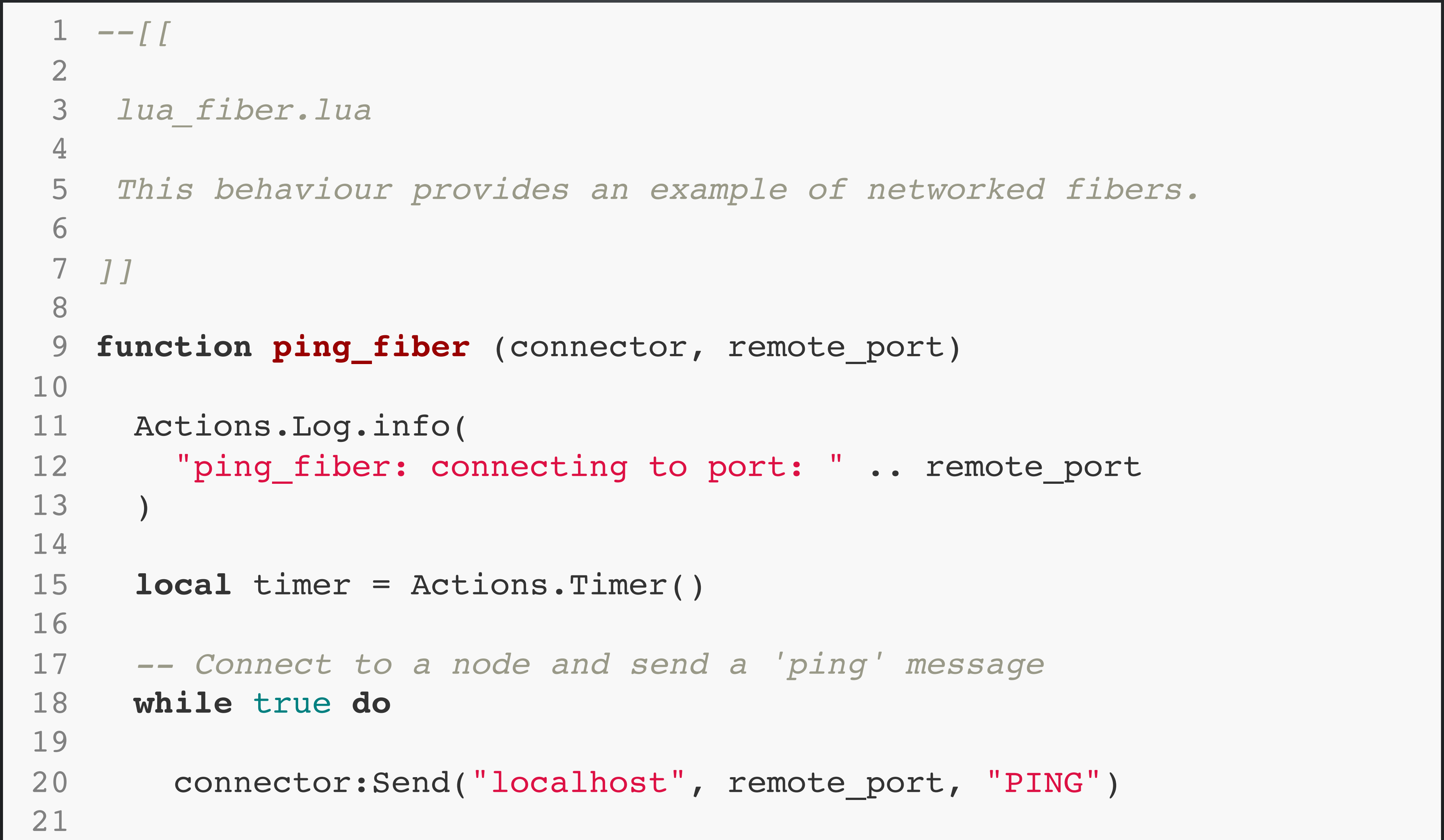
PING

Node 1
Port 7777

Node 2
Port 8888

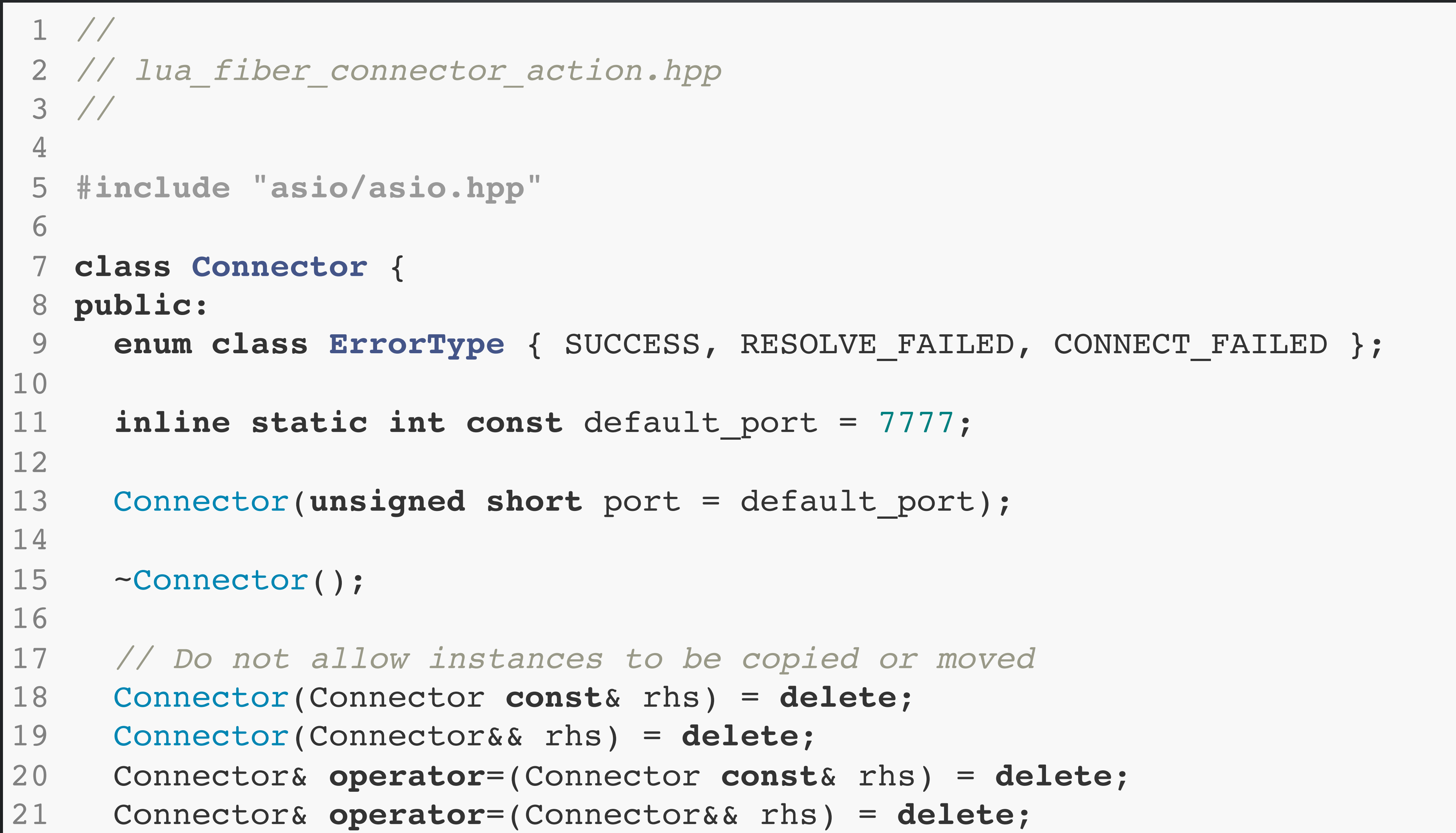
PONG

LUA BEHAVIOR



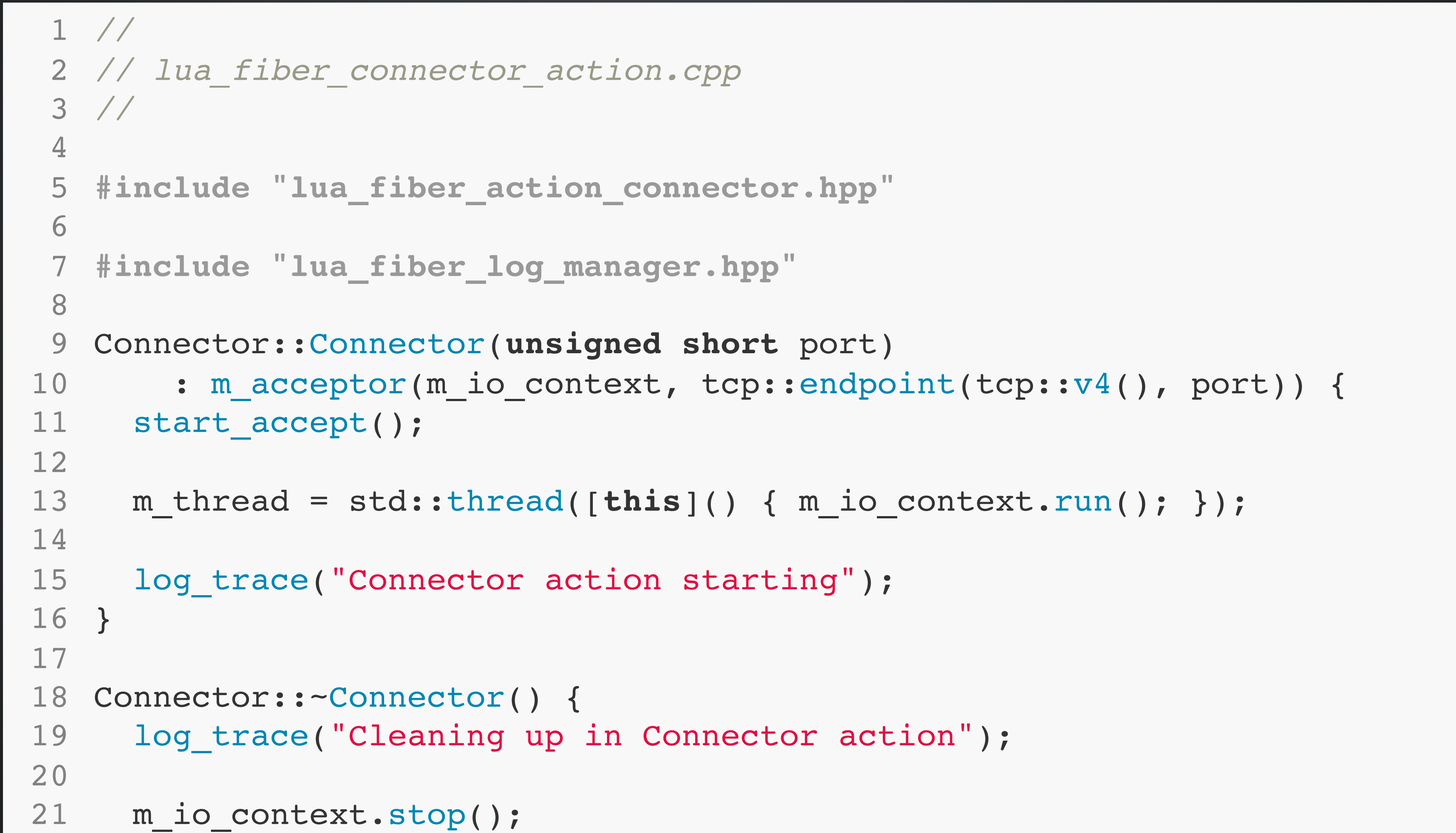
```lua
--[[

 lua_fiber.lua

 This behaviour provides an example of networked fibers.

]]

function ping_fiber (connector, remote_port)

  Actions.Log.info(
    "ping_fiber: connecting to port: " .. remote_port
  )

  local timer = Actions.Timer()

  -- Connect to a node and send a 'ping' message
  while true do

    connector:Send("localhost", remote_port, "PING")

```

CONNECTOR ACTION



```cpp
//
// lua_fiber_connector_action.hpp
//

#include "asio/asio.hpp"

class Connector {
public:
  enum class ErrorType { SUCCESS, RESOLVE_FAILED, CONNECT_FAILED };

  inline static int const default_port = 7777;

  Connector(unsigned short port = default_port);

  ~Connector();

  // Do not allow instances to be copied or moved
  Connector(Connector const& rhs) = delete;
  Connector(Connector&& rhs) = delete;
  Connector& operator=(Connector const& rhs) = delete;
  Connector& operator=(Connector&& rhs) = delete;
```

CONNECTOR ACTION



```cpp
//
// lua_fiber_connector_action.cpp
//

#include "lua_fiber_action_connector.hpp"

#include "lua_fiber_log_manager.hpp"

Connector::Connector(unsigned short port)
    : m_acceptor(m_io_context, tcp::endpoint(tcp::v4(), port)) {
  start_accept();

  m_thread = std::thread([this]() { m_io_context.run(); });

  log_trace("Connector action starting");
}

Connector::~Connector() {
  log_trace("Cleaning up in Connector action");

  m_io_context.stop();
```

C++20 COROUTINES

COROUTINES

Coroutines are subroutines with enhanced semantics

- Invoked by a caller (and return to a caller) ...
- Can suspend execution
- Can resume execution (at a later time)

BENEFITS

Write asynchronous code ...
with the readability of synchronous code

- Useful for networking
- Lots of blocking operations (connect, send, receive)
- Multi-threading (send and receive threads)
- Asynchronous operations mean callbacks
- Control flow fragments

COROUTINE SUPPORT IN C++20

- Three new keywords: co_await, co_yield, co_return
- New types:
  - `coroutine_handle<P>`
  - `coroutine_traits<Ts...>`
- Trivial awaitables:
  - `std::suspend_always`
  - `std::suspend_never`

KEY TALKS AND REFERENCES

- Lots of good talks at CppCon 2022
  - Understanding C++ Coroutines by Example: Generators - Pavel Novikov
  - Deciphering C++ Coroutines - A Diagrammatic Cheat Sheet - Andreas Weis
  - C++ Coroutines, from Scratch - Phil Nash
  - C++20's Coroutines for Beginners - Andreas Fertig

- Lewis Baker's blog posts:
  - Coroutine Theory
  - Understanding operator co_await
  - Understanding the promise type
  - Understanding Symmetric Transfer

AWAITABLE TYPE

- Supports the `co_await` operator
- Controls the semantics of an await-expression
- Informs the compiler how to obtain the awaiter

```
co_await async_write(..., use_awaitable);
```

AWAITER TYPE

- Defines suspend and resume behaviour
- `await_ready`: is suspend required?
- `await_suspend`: schedule resume
- `await_resume`: `co_await` return result
- Can be the same as the awaitable type

COROUTINE RETURN TYPE

- Declares the promise type to the compiler
  - Using `coroutine_traits`
- E.g. '`task<T>`' or '`generator<T>`'
- CppCoro defines several return types
- Referred to as a 'future' in some WG21 papers
- Not to be confused with `std::future`

PROMISE TYPE

- Controls the coroutine's behaviour
  - ... example coming up
- Implements methods that are called at specific points during the execution of the coroutine
- Conveys coroutine result (or exception)
- Again - not to be confused with `std::promise`

COROUTINE HANDLES

- Handle to a coroutine frame on the heap
- Means through which coroutines are resumed
- Also provide access to the promise type
- Non-owning - have to be destroyed explicitly
  - Often through RAII in the coroutine return type

GENERATOR EXAMPLE

```cpp
//
// card_dealer.cpp
// ---------------
//

#include <coroutine>
#include <array>
#include <random>
#include <string>

#include <iostream>

template <typename T> struct generator {
  struct promise_type;
  using coroutine_handle = std::coroutine_handle<promise_type>

  struct promise_type {
    T current_value;

    auto get_return_object() {
      return generator{coroutine_handle::from_promise(*this)};
```

COROUTINES APPLIED

OBSERVATIONS

- C++20 coroutines are powerful ... but complex
- At the application level, how do we:
  - Compare different forms of asynchrony
  - Evaluate/benchmark performance
  - Understand what's going on at the hardware level
- What is a practical methodology for doing this?

BOOST.BEAST

- HTTP and WebSocket built on Boost.Asio
- Excellent web server examples:
  - Asynchronous (callback based)
  - Stackful coroutines (Boost.Coroutine)
  - C++20 coroutines (awaitables)
- Recode for simplicity and test with Apache Bench

APACHE BENCH

- 'ab' is a tool for benchmarking HTTP servers
- Mature implementation with extensive set of options
- Number of concurrent requests is configurable
- Measures 'requests per second' that can be serviced

HTTP SERVER: ASYNCHRONOUS

```cpp
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <boost/asio/strand.hpp>

#include <iostream>
#include <thread>
#include <format>

namespace beast = boost::beast;
namespace http = beast::http;
namespace asio = boost::asio;
using tcp = boost::asio::ip::tcp;

void error(beast::error_code ec, char const* what)
{
  std::cerr << std::format("Error: {} : {}\n", what, ec.message());
  return;
};

// Handles an HTTP server connection
class session : public std::enable_shared_from_this<session>
```

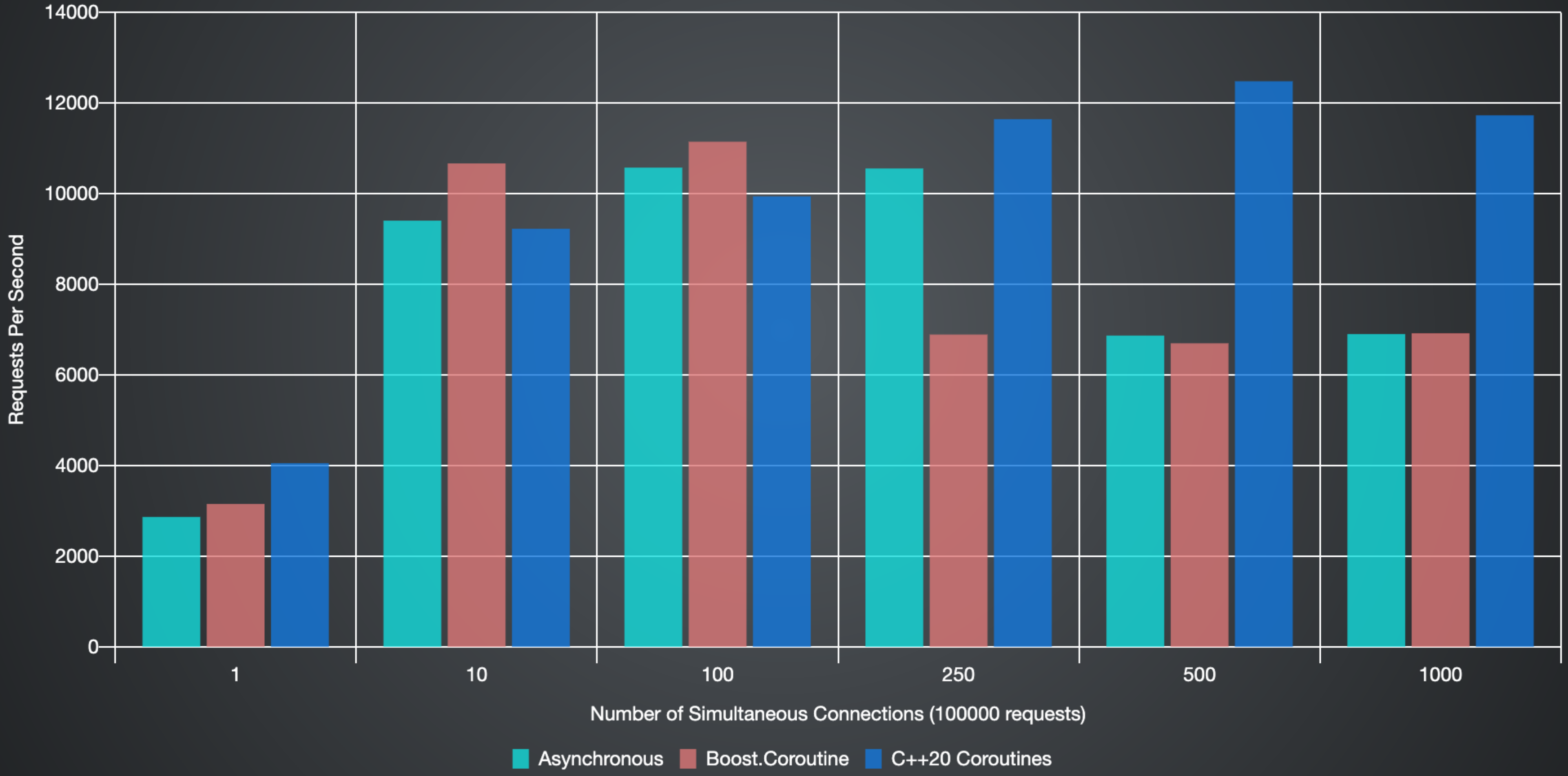
HTTP SERVER: STACKFUL COROUTINES

```cpp
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <boost/asio/spawn.hpp>

#include <iostream>
#include <thread>
#include <vector>
#include <format>

namespace beast = boost::beast;
namespace http = beast::http;
namespace asio = boost::asio;
using tcp = boost::asio::ip::tcp;

// Report an error
void error(beast::error_code ec, char const* msg)
{
  std::cerr << std::format("Error: {} - {}\n", msg, ec.message());
}

void do_session(
```

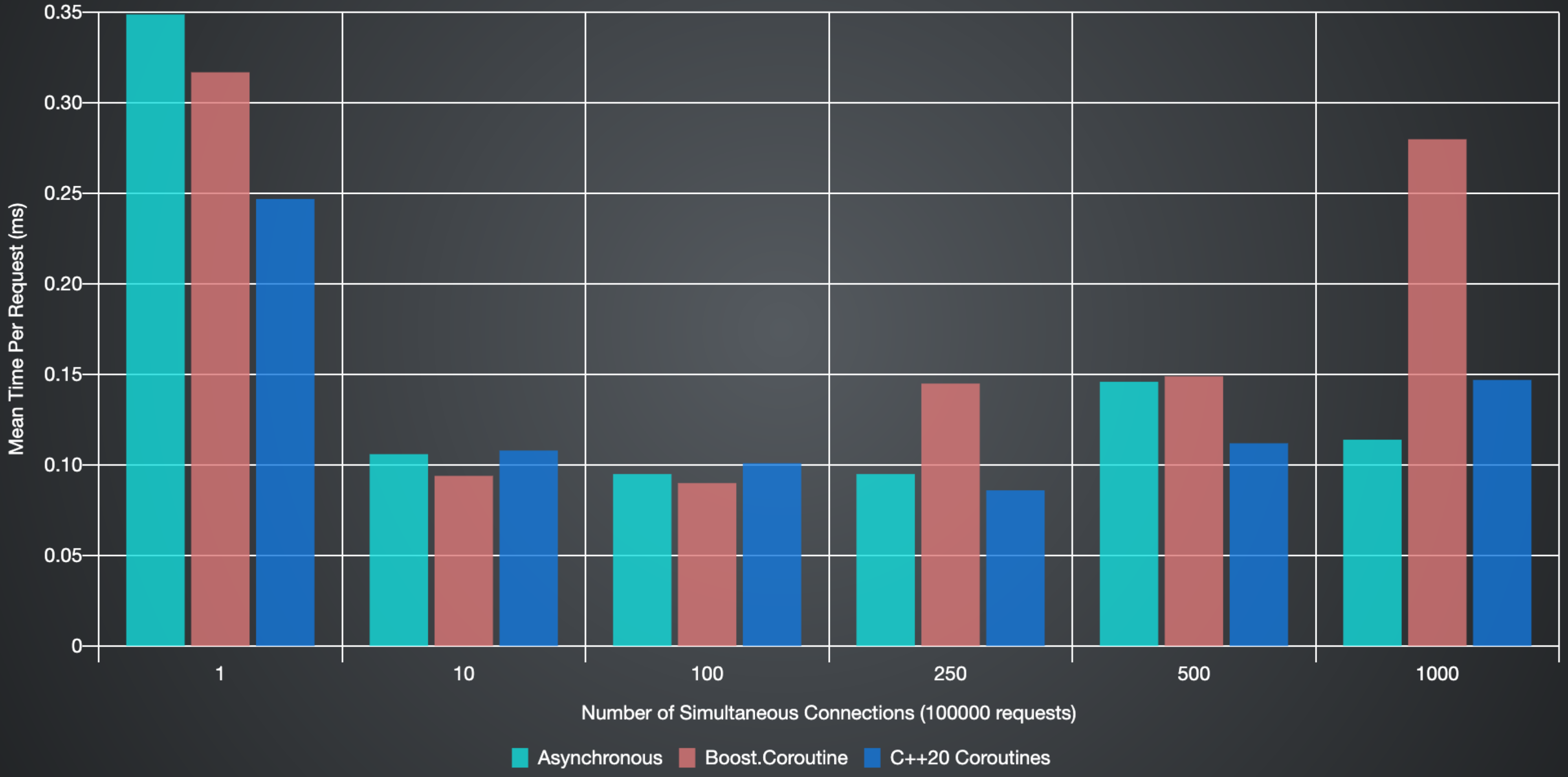
HTTP SERVER: C++20 COROUTINES

```cpp
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <boost/asio/awaitable.hpp>
#include <boost/asio/co_spawn.hpp>
#include <boost/asio/use_awaitable.hpp>

#include <iostream>
#include <thread>
#include <vector>
#include <format>

namespace beast = boost::beast;
namespace http = beast::http;
namespace asio = boost::asio;
using tcp = boost::asio::ip::tcp;

using tcp_stream = typename beast::tcp_stream::rebind_executor<
  asio::use_awaitable_t<>::
    executor_with_default<asio::any_io_executor>>::other;

// Handles an HTTP server connection
```

WEB SERVER PERFORMANCE COMPARISON: X86-64



TIME PER REQUEST (MS): X86-64



CONCLUSIONS

DEBUGGING TIPS

- Design concurrency before implementing
  - Eliminate bugs by design e.g. race conditions
- Be careful with object lifetimes
  - Common idiom: RAII class that inherits from `std::enable_shared_from_this`
  - Check for resource exhaustion e.g. `lsof -p`

C++23 STACKTRACE

- C++23 stacktrace can be very helpful:
  - Good support in GCC 12.2
  - Configure with: --enable-libstdcxx-backtrace=yes
  - Compile with: -std=c++23 -lstdc++_libbacktrace

C++23/26 COROUTINE UPDATE

- P2502: standardised generator `std::generator`
  - Models `std::ranges::input_range`
  - Approved for C++23 (June 2022)
  - Not yet implemented in standard libraries
  - Reference implementation: godbolt.org
- P2300: `std::execution`
  - Standardised asynchronous execution
  - ... on generic execution contexts
  - Targeting C++26 (see also: libunifex)

CONCLUSION

- Coroutines allow asynchronous code to be written
  - With the readability of synchronous code
  - Fibers: a light-weight alternative to threading
  - Empirical insights are compelling
- Using coroutines in user-code:
  - Boost.Asio and Boost.Beast are great for this
  - Persevere with key references

QUESTIONS?

http://www.james-pascoe.com
james@james-pascoe.com

http://jamespascoe.github.io/accu2023

https://github.com/jamespascoe/accu2023-example-code.git

ACCU Bristol and Bath Meetup Coordinator