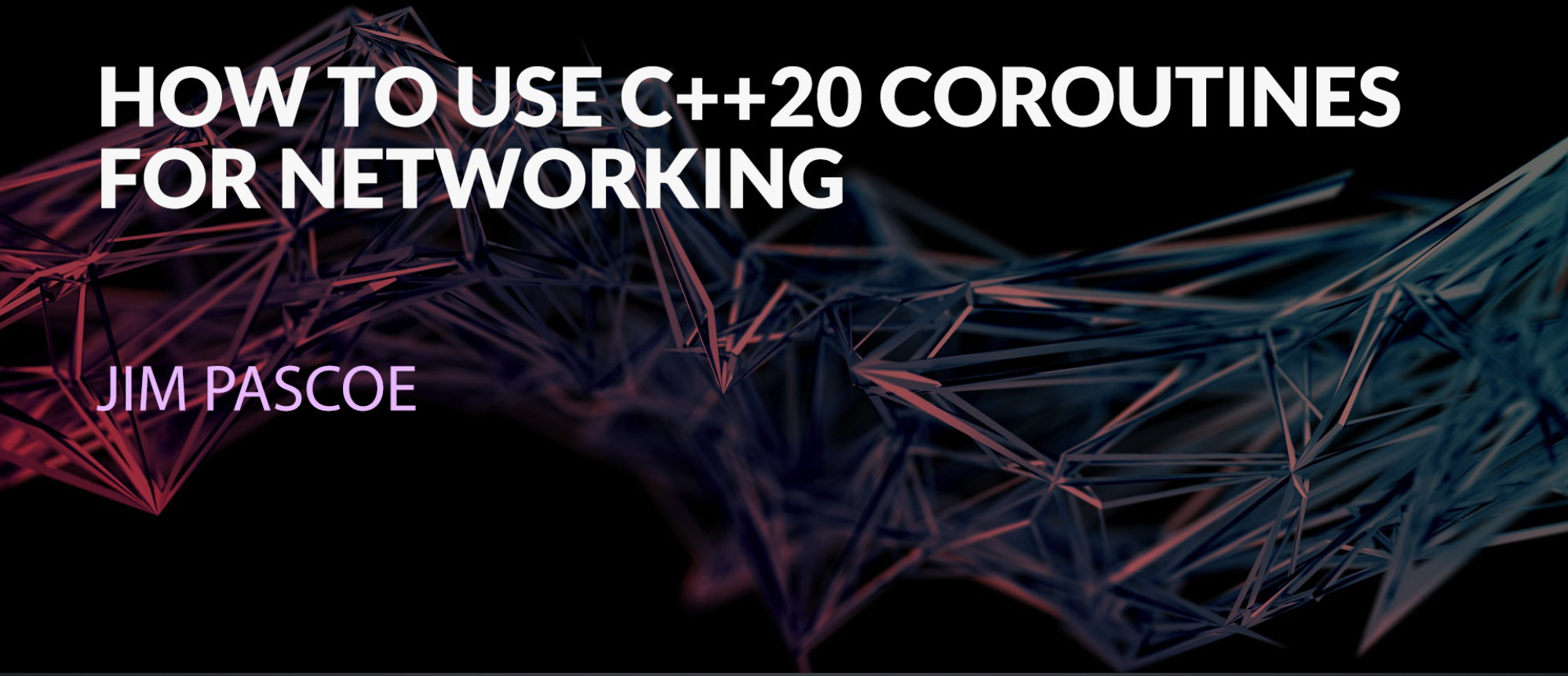
ACCU 2022

HOW TO USE C++20 COROUTINES FOR NETWORKING

JIM PASCOE



HOW TO USE C++20 COROUTINES FOR NETWORKING

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

http://jamespascoe.github.io/accu2022
https://github.com/jamespascoe/accu2022-example-code.git

HOW FAMILIAR ARE YOU WITH COROUTINES?

1. I am a coroutine expert
2. I have a strong grasp but need to fill in some details
3. I have some understanding but want to learn more
4. I am just starting to learn about them

OVERVIEW

Demystify Using C++20 Coroutines for Networking
Emphasis on practical examples and code

- Coroutines: fundamentals, benefits and usage
  - Lua 5.4.4 and C++20
- How to Write Networking Code Using Coroutines
  - CoChat: a coroutine based chat program
  - Boost Asio (1.78)
- C++23/26: the future of Coroutines

EXAMPLE CODE: TOOLS & BUILD

- C++ examples all compile with GCC 11.2:
  - Boost Asio 1.78.0 (also works with 1.76.0)
- Lua examples run with Lua 5.4.4:
  - Requires `luaposix` and `luasocket`
  - `sudo luarocks install luaposix`
  - `sudo luarocks install luasocket`
- Tested on Linux Mint 19 and Mac OS X (Mojave, Big Sur)

FUNDAMENTALS

COROUTINES

Coroutines are subroutines with enhanced semantics

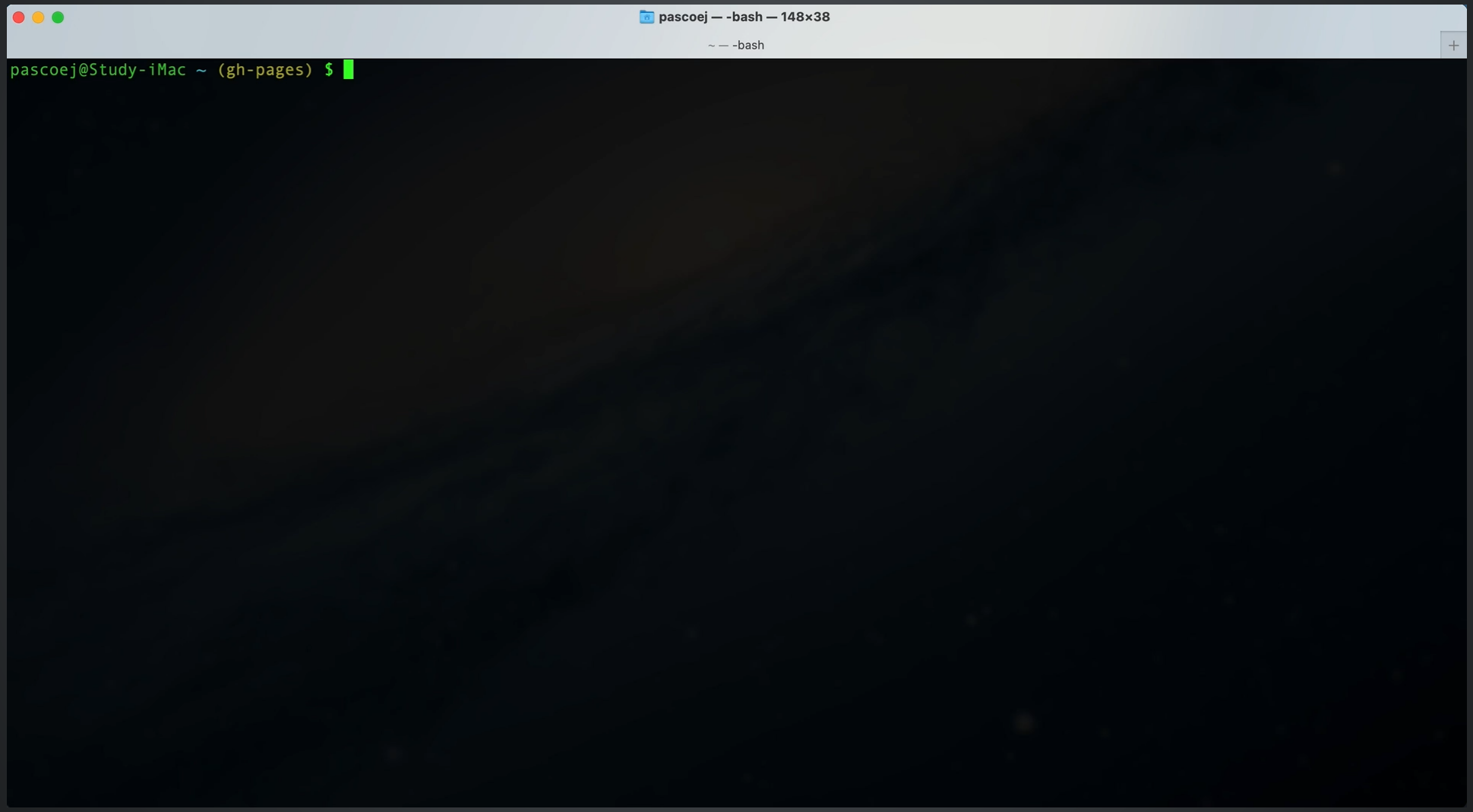
- Invoked by a caller (and return to a caller) ...
- Can suspend execution
- Can resume execution (at a later time)

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

ECHO SERVER BUILD & RUN



BLOCKING ECHO SERVER

```
1  //
2  // echo_server_blocking.cpp
3  //
4
5  #include <boost/asio.hpp>
6  #include <iostream>
7
8  using boost::asio::buffer;
9  using boost::asio::io_context;
10 using boost::asio::ip::tcp;
11 using boost::system::error_code;
12
13 static const int buf_len = 1000;
14
15 int main() {
16   try {
17     io_context ctx;
18
19     tcp::acceptor acceptor(ctx, tcp::endpoint(tcp::v4(), 6666));
20
21     for (;;) {
```

ASYNCHRONOUS ECHO SERVER

```
1  //
2  // echo_server_async.cpp
3  //
4
5  #include <boost/asio.hpp>
6  #include <iostream>
7
8  using boost::asio::buffer;
9  using boost::asio::io_context;
10 using boost::asio::ip::tcp;
11 using boost::system::error_code;
12
13 static const int buf_len = 1000;
14
15 void accept_handler(error_code const &error,
16                     tcp::socket &peer_socket,
17                     std::array<char, buf_len> &buf,
18                     tcp::acceptor &acceptor);
19
20 void read_handler(error_code const &error,
21                   std::size_t bytes_transferred,
```

COROUTINE ECHO SERVER

```
1  //
2  // echo_server_coroutine.cpp
3  //
4
5  #include <boost/asio.hpp>
6  #include <boost/asio/experimental/as_tuple.hpp>
7
8  #include <iostream>
9
10 using boost::asio::buffer;
11 using boost::asio::detached;
12 using boost::asio::io_context;
13 using boost::asio::use_awaitable;
14 using boost::asio::experimental::as_tuple;
15 using boost::asio::ip::tcp;
16 using boost::system::error_code;
17
18 boost::asio::awaitable<void> echo(tcp::socket peer_socket,
19                                   tcp::acceptor acceptor) {
20   std::array<char, 1000> buf;
21
```

C++20 COROUTINES

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`

TIPS FOR LEARNING

- Principally for library development (not user-code)
- Libraries with coroutine support:
  - Boost Asio:
    - (first in 1.67.0, 'C++20 Support' in 1.77.0)
  - Lewis Baker's CppCoro:
    - Contains coroutine types, awaitables etc.
- References to 'promise' and 'future' are not `std::promise` and `std::future`!

KEY REFERENCES

- Lewis Baker's blog posts:
  - Coroutine Theory
  - Understanding operator co_await
  - Understanding the promise type
  - Understanding Symmetric Transfer
- Dave Mazière's blog post:
  - My tutorial and take on C++20 coroutines
- Gor Nishanov:
  - C++ Extensions for Coroutines (N4775)

TUTORIAL OVERVIEW

- Goal: implement a 'chat' program using coroutines
- Prototype in Lua 5.4.4 (and then write it in C++)
- C++20 coroutines:
  - co_await, co_yield and co_return
  - Return objects, promises, awaitables and traits
  - Generators
- Implement 'chat' in C++ (using Boost Asio 1.78)

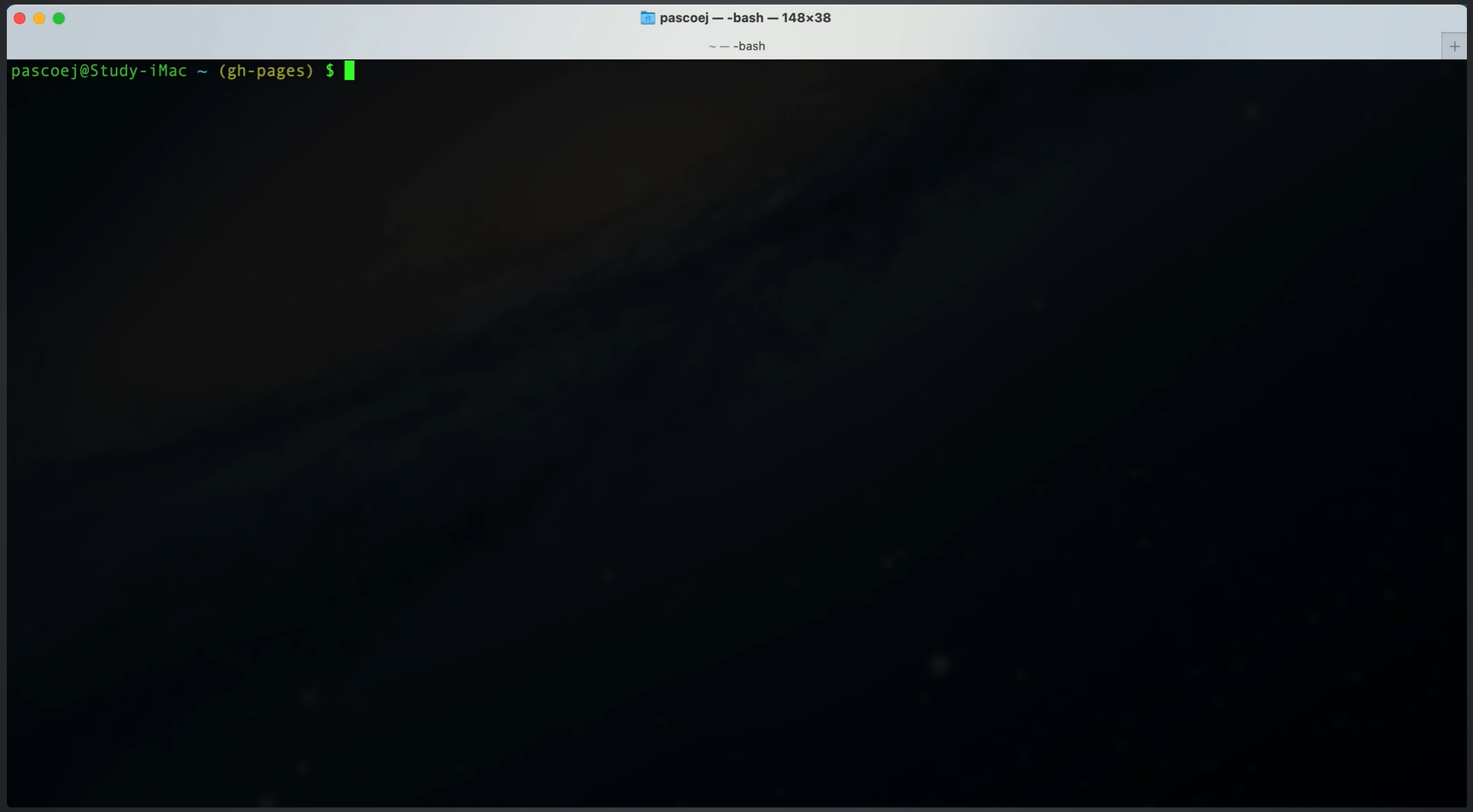
LUA

- Lightweight embeddable scripting language
- Good way of building a mental model of coroutines
- Single threaded so lock-free, no races etc.
- Implement your own dispatcher (executor) in Lua
- Lua coroutines are stackful
- C++20 coroutines are stackless

LUACHAT

- Sender coroutine: sends user input to peer
- Receiver coroutine: prints received messages
- Dispatcher: schedules sender and receiver
- main: processes arguments and creates coroutines

LUACHAT SCREEN CAPTURE



SENDER COROUTINE

```lua
-- Connect to the peer and send messages read from stdin
function sender (host, port)

  while true do

    local remote, err = socket.connect(host, port)
    while not remote do
      coroutine.yield()

      remote, err = socket.connect(host, port)
    end

    print("Connected to " .. host .. ":" .. port)

    while err ~= "closed" do
      -- Read from stdin (non-blocking - 1s timeout)
      local ret = require "posix".rpoll(0, 1000)
      if (ret == 1) then
        local message = io.read()
        if (message ~= "") then
          _, err = remote:send(message .. "\n")
```

RECEIVER COROUTINE

```lua
-- Receive messages from our peer and print them
function receiver (port)

  local server = assert(socket.bind("*", port))
  server:settimeout(0.1) -- set non-blocking (100 ms timeout)

  while true do

    local _, port = server:getsockname()
    print("Waiting for connection on port " .. port);

    local client, err = server:accept()
    if (not client and err == "timeout") then
      coroutine.yield()
    else
      local peer_ip, peer_port = client:getpeername()

      client:send("Connected to LuaChat!\n")
      client:settimeout(0.1)

      while err ~= "closed" do
```

DISPATCHER

```lua
function dispatcher (coroutines)

  while true do
    if next(coroutines) == nil then break end -- no more coroutines

    for name, co in pairs(coroutines) do
      local status, res = coroutine.resume(co)

      if res then -- coroutine has returned a result (i.e. finished)
        if type(res) == "string" then -- runtime error
          print("Lua coroutine '" .. name ..
                "' has exited with error: " .. res)
        else
          print("Lua coroutine '" .. name .. "' exited")
        end

        coroutines[name] = nil
      end
    end
  end
end
```

MAIN CODE

```lua
local port, remote_ip, remote_port = tonumber(arg[1]),
                                     arg[2],
                                     tonumber(arg[3])

print(
  string.format("Starting LuaChat:\n" ..
                "  local port: %d\n" ..
                "  remote IP: %s\n" ..
                "  remote port: %d\n\n", port, remote_ip, remote_port)
)

-- Create co-routines
local coroutines = {}
coroutines["receiver"] = coroutine.create(receiver)
coroutine.resume(coroutines["receiver"], port)

coroutines["sender"] = coroutine.create(sender)
coroutine.resume(coroutines["sender"], remote_ip, remote_port)

-- Run the main loop
dispatcher(coroutines)
```

COROUTINE DETAILS

AWAITABLE TYPE

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

```
1 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`

CUSTOMISING CO_AWAIT

- The `await_transform` method:
  - Defined in the `promise_type`
  - Enables types that are not awaitable
  - Disables `co_await` on certain types
  - Modify the behaviour of awaitable values
- Also possible to customise `co_yield`
- See Lewis Baker's excellent Blog Post for details

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

```
 1 //
 2 // card_dealer.cpp
 3 // ---------------
 4 //
 5
 6 #include <coroutine>
 7 #include <array>
 8 #include <random>
 9 #include <string>
10
11 #include <iostream>
12
13 template <typename T> struct generator {
14   struct promise_type;
15   using coroutine_handle = std::coroutine_handle<promise_type>
16
17   struct promise_type {
18     T current_value;
19
20     auto get_return_object() {
21       return generator{coroutine_handle::from_promise(*this)};
```

TRAFFIC GENERATOR

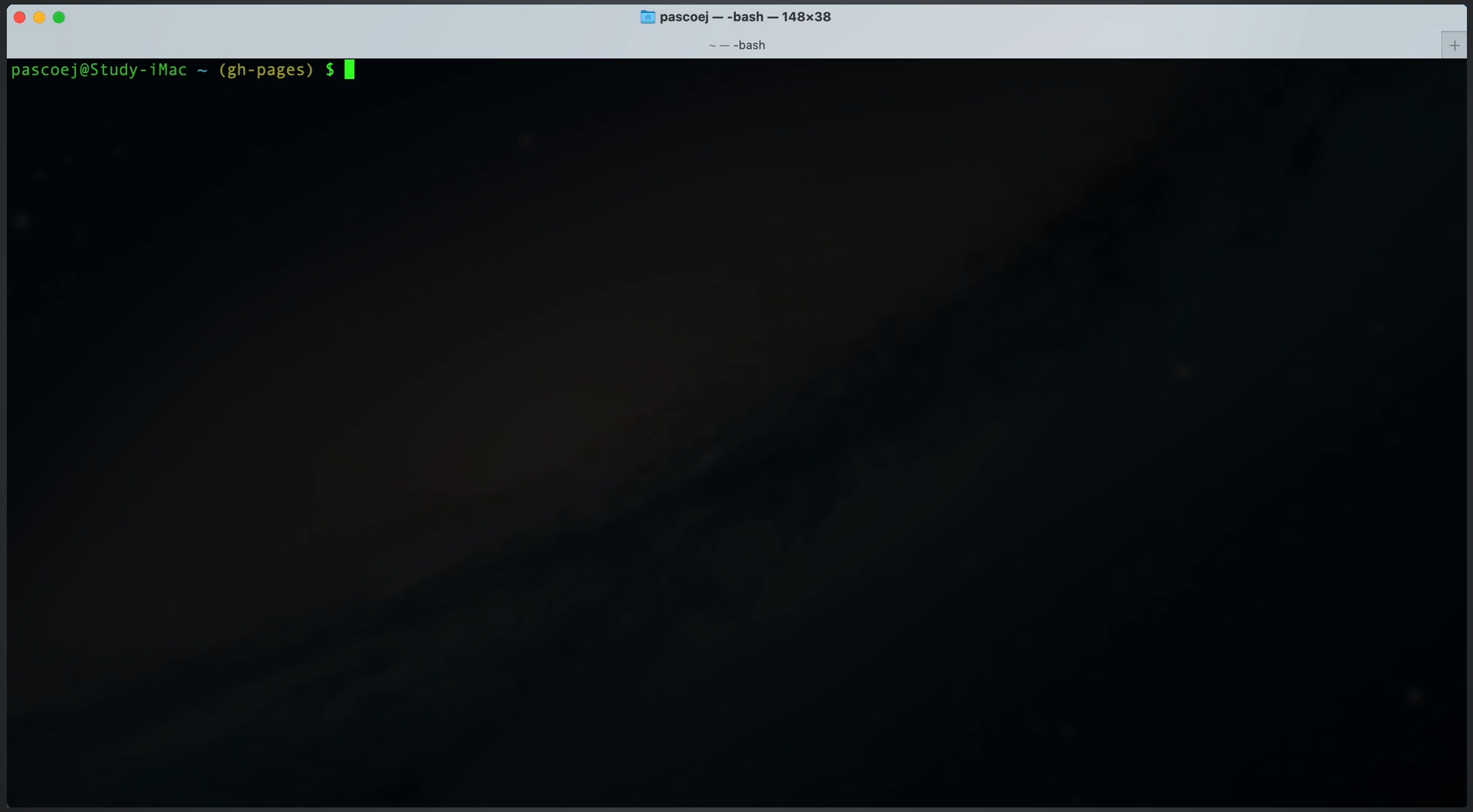
- A networking example would be a traffic generator
- Coroutine generates packets
  - Sequence number, checksums, encryption
- One coroutine transmits with `co_await`
- A second coroutine builds the next packet
  - ... during the latency window

CHAT EXAMPLE

COROUTINE CHAT

- Prototyped in Lua - implement in C++20
- Retain coroutine design:
  - Sender: reads keyboard and sends
  - Receiver: prints messages
  - Dispatcher: Boost IO context
- Must be non-blocking
- Users should be able to reconnect

BUILD & RUN



CHAT CLASS

```
 1 class chat {
 2
 3 public:
 4   chat(char *addr, char *port, char *remote_addr,
 5        char *remote_port) {
 6
 7     tcp::resolver resolver(ctx);
 8     basic_resolver_entry<tcp> listen_endpoint;
 9     basic_resolver_entry<tcp> remote_endpoint;
10
11     listen_endpoint = *resolver.resolve(addr, port, passive);
12     remote_endpoint = *resolver.resolve(remote_addr, remote_port);
13
14     co_spawn(ctx, sender(remote_endpoint), detached);
15     co_spawn(ctx, receiver(listen_endpoint), detached);
16
17     ctx.run();
18   }
19
20 private:
21   awaitable<void> sender(tcp::endpoint remote) {
```

SENDER COROUTINE

```
1  awaitable<void> sender(tcp::endpoint remote) {
2
3    for (;;) {
4      tcp::socket remote_sock(ctx);
5
6      auto [error] = co_await remote_sock.async_connect(
7          remote, as_tuple(use_awaitable));
8      if (!error) {
9        std::cout << "Connected to: " << remote << std::endl;
10       connected = true;
11     } else {
12       std::cout << "Could not connect to: " << remote
13                 << " - retrying in 1s " << std::endl;
14
15       steady_timer timer(
16           co_await boost::asio::this_coro::executor);
17       timer.expires_after(std::chrono::seconds(1));
18       co_await timer.async_wait(use_awaitable);
19
20       continue;
21     }
```

RECEIVER COROUTINE

```
1  awaitable<void> receiver(tcp::endpoint listen) {
2
3    tcp::acceptor acceptor(ctx, listen);
4
5    for (;;) {
6      auto [error, client] =
7          co_await acceptor.async_accept(as_tuple(use_awaitable));
8
9      if (!error) {
10       std::string data;
11
12       for (;;) {
13         auto [error, len] = co_await async_read_until(
14             client, dynamic_buffer(data), boost::regex("\r\n"),
15             as_tuple(use_awaitable));
16
17         if (error == boost::asio::error::eof) {
18           // remote has disconnected
19           connected = false;
20           break;
21         }
```

COMPLETE LISTING

```
 1 #include <boost/asio.hpp>
 2 #include <boost/asio/experimental/as_tuple.hpp>
 3 #include <boost/regex.hpp>
 4
 5 #include <iostream>
 6
 7 using boost::asio::awaitable;
 8 using boost::asio::buffer;
 9 using boost::asio::co_spawn;
10 using boost::asio::detached;
11 using boost::asio::dynamic_buffer;
12 using boost::asio::io_context;
13 using boost::asio::steady_timer;
14 using boost::asio::use_awaitable;
15 using boost::asio::experimental::as_tuple;
16 using boost::asio::ip::basic_resolver_entry;
17 using boost::asio::ip::tcp;
18 using boost::asio::ip::tcp::resolver::passive;
19
20 using boost::system::error_code;
21
```

CONCLUSION

WHAT'S COMING IN C++23/26?

- Library support for coroutines was planned
- P2502: standardised generator `std::generator`
  - Models `std::ranges::input_range`
  - Accepted by LEWG in Jan 2022 - sent to LWG
  - But, appears to have missed C++23
- P2300: standardised execution `std::execution`
  - Targeting C++26 (see also: libunifex)

CONCLUSION

- C++20 coroutines provide a language capability
  - Functions can be suspended and resumed
- Coroutines allow asynchronous code to be written
  - With the readability of synchronous code
- Using coroutines in user-code:
  - Library support is improving
  - Persevere with key references

QUESTIONS?

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

http://jamespascoe.github.io/accu2022

https://github.com/jamespascoe/accu2022-example-code.git