Capgemini engineering

WHY USE COROUTINES FOR ASYNCHRONOUS APPLICATIONS?

Johan Vanslembrouck

WHY USE COROUTINES FOR ASYNCHRONOUS APPLICATIONS?

Goal of this presentation

- Using a variety of examples, ranging from simple to more complex,
- compare 3 programming styles
  - synchronous (using a synchronous I/O API)
  - asynchronous (using an asynchronous I/O API)
  - coroutines (using the same asynchronous I/O API)
- and demonstrate that coroutines allow writing applications that
  - exhibit asynchronous behaviour
  - using a synchronous programming style
- combining the advantages of both styles, i.e.,
  - the easy-to-write, read and maintain synchronous style
  - with the efficient non-blocking execution of the asynchronous style
- without introducing any major disadvantages (apart from an inevitable learning curve).

WHY USE COROUTINES FOR ASYNCHRONOUS APPLICATIONS?

Some quotes

Lewis Baker in https://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer:

"The Coroutines TS provided a wonderful way to write asynchronous code as if you were writing synchronous code. You just need to sprinkle co_await at appropriate points and the compiler takes care of suspending the coroutine, preserving state across suspend-points and resuming execution of the coroutine later when the operation completes."

Max Arshinov in https://dev.to/maxarshinov/a-brief-history-of-asyncawait-264j:

"The async/await pattern is a syntactic feature of many programming languages that allows an asynchronous, non-blocking function to be structured similarly to an ordinary synchronous function. It is semantically related to the concept of a coroutine and is often implemented using similar techniques."

WHY USE COROUTINES FOR ASYNCHRONOUS APPLICATIONS?

Appetizer example: synchronous versus coroutine

```
int function1(int in1, int in2, int testval)
{
    int out1 = -1, out2 = -1;
    int ret1 = remoteObj1.op1(in1, in2, out1, out2);
    // 1 Do stuff
    if (ret1 == testval) {
        int out3 = -1;
        int ret2 = remoteObj2.op2(in1, in2, out3);
        // 2 Do stuff
        return ret2;
    }
    else {
        int out4 = -1, out5 = -1;
        int ret3 = remoteObj3.op3(in1, out4, out5);
        // 3 Do stuff
        return ret3;
    }
}
```

```
async_task<int> coroutine1(int in1, int in2, int testval)
{
    int out1 = -1, out2 = -1;
    int ret1 = co_await remoteObj1co.op1(in1, in2, out1, out2);
    // 1 Do stuff
    if (ret1 == testval) {
        int out3 = -1;
        int ret2 = co_await remoteObj2co.op2(in1, in2, out3);
        // 2 Do stuff
        co_return ret2;
    }
    else {
        int out4 = -1, out5 = -1;
        int ret3 = co_await remoteObj3co.op3(in1, out4, out5);
        // 3 Do stuff
        co_return ret3;
    }
}
```

In yellow: The synchronous and coroutine examples are syntactically very close to each other. The asynchronous example is more complex, see next slide.

In blue: The synchronous variant calls synchronous operations, the coroutine variant uses asynchronous operations.

WHY USE COROUTINES FOR ASYNCHRONOUS APPLICATIONS?

Appetizer example: asynchronous

```
struct function1_ctxt_t
{
    int in1;
    int in2;
    int testval;
};

void function1(int in1, int in2, int testval, int& ret)
{
    function1_ctxt_t* ctxt = new function1_ctxt_t{in1, in2,
                                                  testval};
    remoteObj1.sendc_op1(in1, in2,
        [this, ctxt, &ret](int out1, int out2, int ret1) {
            this->function1a(ctxt, out1, out2, ret1, ret);
        });
    // 1a Do stuff that doesn't need the result of the RMI
}
```

Can be implemented in different ways.
See further in this presentation.
The behavior (control flow) of the coroutine example and the asynchronous example is very similar.

```
void function1a(function1_ctxt_t* ctxt, int out1, int out2,
                int ret1, int& ret)
{
    // 1b Do stuff that needs the result of the RMI
    if (ret1 == ctxt->testval) {
        remoteObj2.sendc_op2(ctxt->in1, ctxt->in2,
            [this, &ret](int out1, int ret1) {
                this->function1b(out1, ret1, ret);
            });
        // 2a Do stuff that doesn't need the result of the RMI
    }
    else {
        remoteObj3.sendc_op3(ctxt->in1,
            [this, &ret](int out1, int out2, int ret1) {
                this->function1c(out1, out2, ret1, ret);
            });
        // 3a Do stuff that doesn't need the result of the RMI
    }
    delete ctxt;
}
void function1b(int out3, int ret2, int& ret)
{
    // 2b Do stuff that needs the result of the RMI
    ret = ret2;
}
void function1c(int out4, int out5, int ret3, int& ret)
{
    // 3b Do stuff that needs the result of the RMI
    ret = ret3;
}
```

BIOGRAPHY

Johan Vanslembrouck

- I started my professional career on 1 October 1984 at Bell Telephone Manufacturing Company (BTMC) in Antwerp, Belgium
  - BTMC (part of ITT until 1986) became Alcatel Bell (Telephone), then Alcatel-Lucent, and is now Nokia.
- I became a consultant for Altran-Europe on 1 February 1999
  - Altran became Capgemini Engineering in April 2022.
  - I worked for clients in various sectors: telecommunication, defence, banking (electronic payment terminals), aeronautics, industrial automation and pharmaceutics. 16 different clients in total, not including company renaming.
- I have been studying and using C++ coroutines since June 2019, in combination with various asynchronous communication frameworks such as Boost ASIO, Qt5, gRPC, TAO, ROS2, Win32 overlapped I/O.
  - Coroutines are fascinating and I did not want to become obsolete before my retirement date.
  - Still, I haven't had the chance to use coroutines in client projects (although I did see some opportunities).

INTRODUCTION

Introductory notes

- This presentation is an evolution of a presentation I gave at the Belgian C++ Users Group on 15 January 2025 PresentationBelgianC++UsersGroup (68 slides), which is an evolution of presentations I gave at 3 client companies, which were evolutions of my very first external presentation on coroutines for Meeting C++ in 2022 https://meetingcpp.com/mcpp/slides/2022/Corolib-DistributedProgrammingWithC++Coroutines1807.pdf (74 slides).
- In this presentation I have added slides referring to software I have encountered during my career and that could have benefited from the use of (C++) coroutines... if only coroutines had been available at that time.
- Because of the large number of slides (130), the focus in this presentation will be more on these real-world case studies and less on detailed technical aspects.
- The original title of the presentation was “Why use coroutines for asynchronous programming?”
  - The term “asynchronous programming” is more often used.
  - But the result of asynchronous programming is an asynchronous program or application.
  - I will use both terms in the rest of this presentation.

AGENDA

1. Brief introduction to C++ coroutines (14 slides)
2. Brief introduction to (a)synchronous programming (4 slides)
3. Why use coroutines for asynchronous programming?
4. When not to use coroutines? (3 slides)
5. Summary and conclusions (4 slides)
6. Appendix: brief introduction to corolib (4 slides)

AGENDA

1. **Brief introduction to C++ coroutines (14 slides)**
2. Brief introduction to (a)synchronous programming
3. Why use coroutines for asynchronous programming?
4. When not to use coroutines?
5. Summary and conclusions
6. Appendix: brief introduction to corolib

INTRODUCTION TO C++ COROUTINES

What is a coroutine?

A coroutine is a generalized routine that in addition to the traditional subroutine operations **call** and **return**, supports **suspend** and **resume** operations.

- Name coined by Melvin Conway in 1958, first publication in 1963.
- Fortran code on the right comes from https://web.chem.ox.ac.uk/fortran/subprograms.html
- Early programming languages, such as Fortran, used the term “subroutine“ instead of “procedure” or “function.”
- Therefore “coroutine” seems to be a natural name for a more general “subroutine.”
- In Fortran CALL and RETURN are explicit “operation” names.
- Processors only support call and return operations. Suspend and resume are implemented with return and call, respectively, at the processor level.

```
PROGRAM SUBDEM
REAL A,B,C,SUM,SUMSQ
CALL INPUT( + A,B,C)
CALL CALC(A,B,C,SUM,SUMSQ)
CALL OUTPUT(SUM,SUMSQ)
END

SUBROUTINE INPUT(X, Y, Z)
REAL X,Y,Z
PRINT *,'ENTER THREE NUMBERS => '
READ *,X,Y,Z
RETURN
END

SUBROUTINE CALC(A,B,C, SUM,SUMSQ)
REAL A,B,C,SUM,SUMSQ
SUM = A + B + C
SUMSQ = SUM **2
RETURN
END

SUBROUTINE OUTPUT(SUM,SUMSQ)
REAL SUM, SUMSQ
PRINT *,'The sum of the numbers you entered are: ',SUM
PRINT *,'And the square of the sum is:',SUMSQ
RETURN
END
```

INTRODUCTION TO C++ COROUTINES

C++: what is a coroutine, how do you recognize a coroutine?

A C++ function is a coroutine if it contains one or more of the following:

- a **co_return** statement: returns from a coroutine (just using **return** is not allowed)
- a **co_await** expression: (conditionally) suspends evaluation of a coroutine while waiting for a computation to finish
- a **co_yield** expression: returns a value from a coroutine back to the caller and suspends the coroutine; subsequently calling the coroutine again continues its execution.

A coroutine must return an object of a coroutine type (which often has ‘task’ in its name)

- It cannot return just an int, void, double, etc.
- Consequently, main() cannot be a coroutine (the OS will not resume main() if it has suspended).

The C++20 standard only defines mechanisms (low-level primitives) to define coroutine (types).

- You must implement a coroutine support library (with coroutine types and coroutines) yourself.
- Or find an implementation on the Internet: https://github.com/JohanVanslembrouck/corolib

INTRODUCTION TO C++ COROUTINES

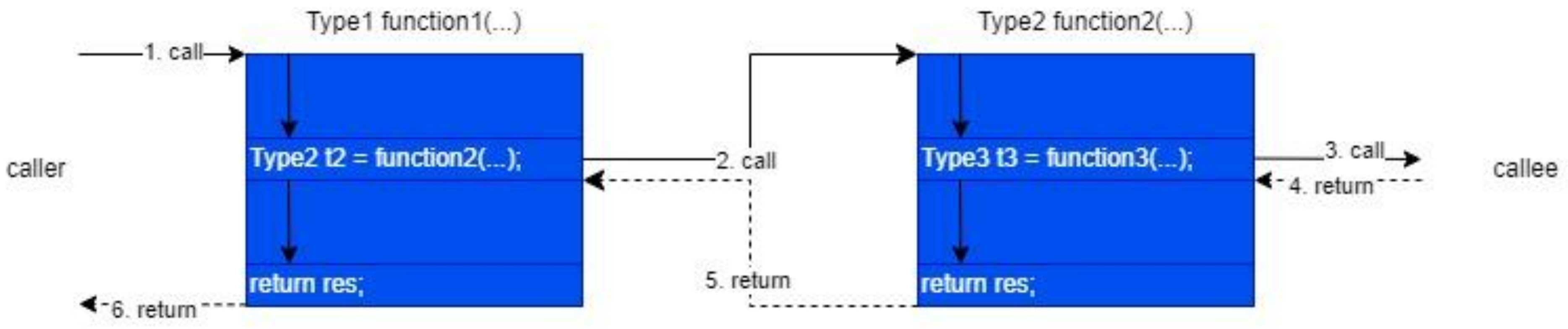
Dynamic C: costatements and cofunctions

Dynamic C provides extensions to the C language that support real-world embedded system development

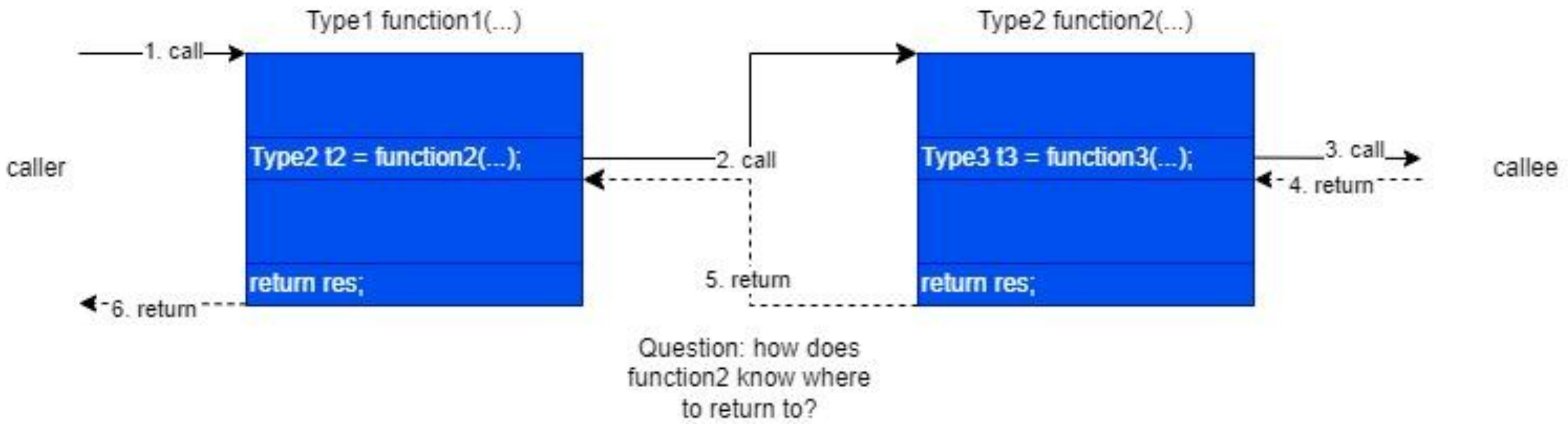
- Costatements allow cooperative, parallel processes to be simulated in a single program.
- Cofunctions allow cooperative processes to be simulated in a single program.
- Slice Statements allow preemptive processes in a single program.
- User Manual: https://hub.digi.com/dp/path=/support/asset/dynamic-c-9-users-manual-rabbit-2000-and-3000-microprocessors/
- Originally developed by Rabbit Semiconductor for its microprocessor-based products (Rabbit 2000 and 3000).
- I discovered Dynamic C in 2002 while doing some research on cooperative multi-tasking for a client that had developed a cooperative scheduler (OS).
- An example of Dynamic C code follows below.

INTRODUCTION TO C++ COROUTINES

1. Function control flow



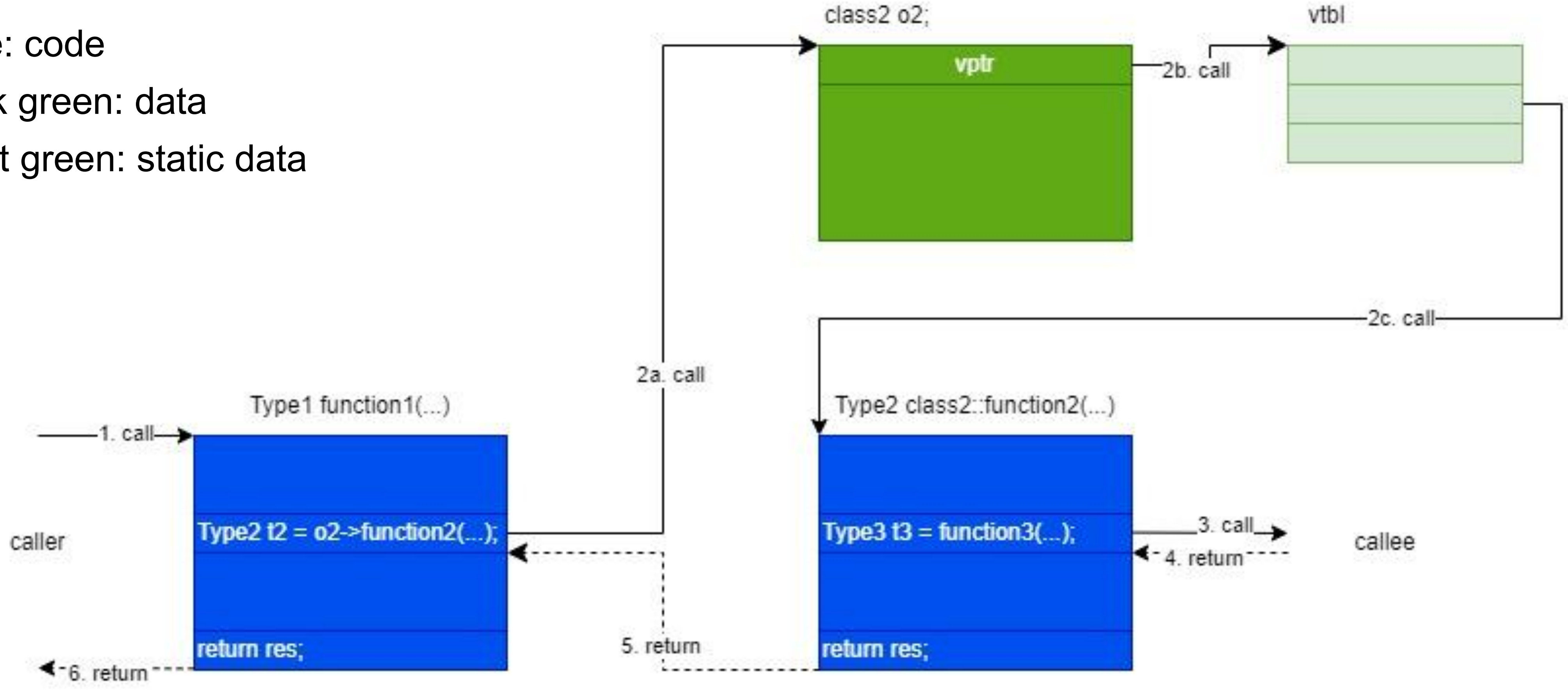
In the rest of the presentation, I will omit the arrows inside the functions and coroutines, as in the picture below.



INTRODUCTION TO C++ COROUTINES

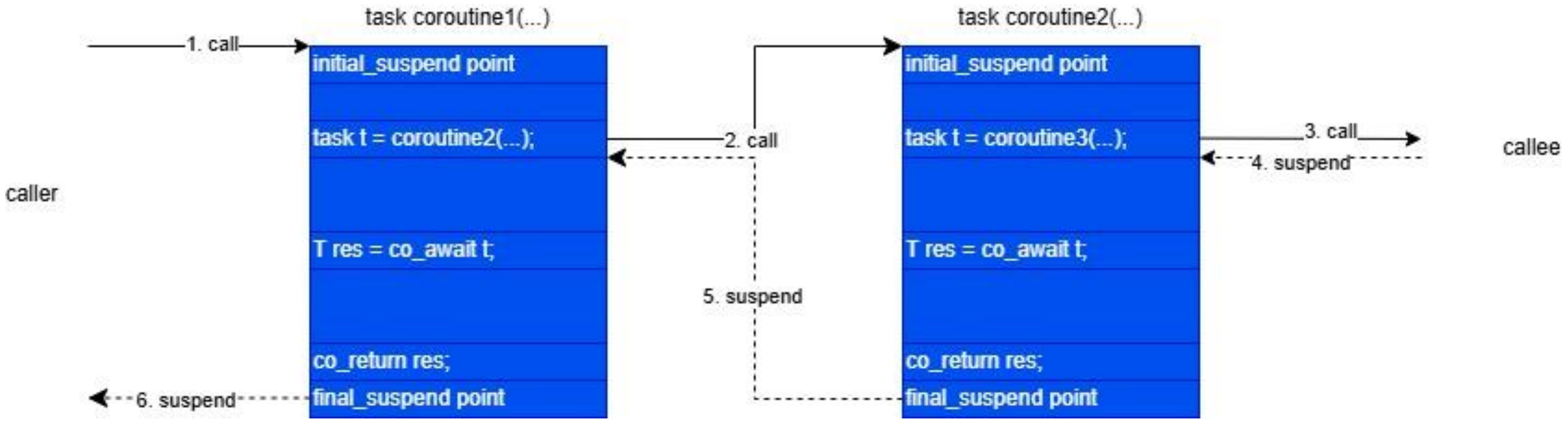
2. Function control flow: virtual functions

- Blue: code
- Dark green: data
- Light green: static data



INTRODUCTION TO C++ COROUTINES

3. Coroutine control flow with synchronous completion

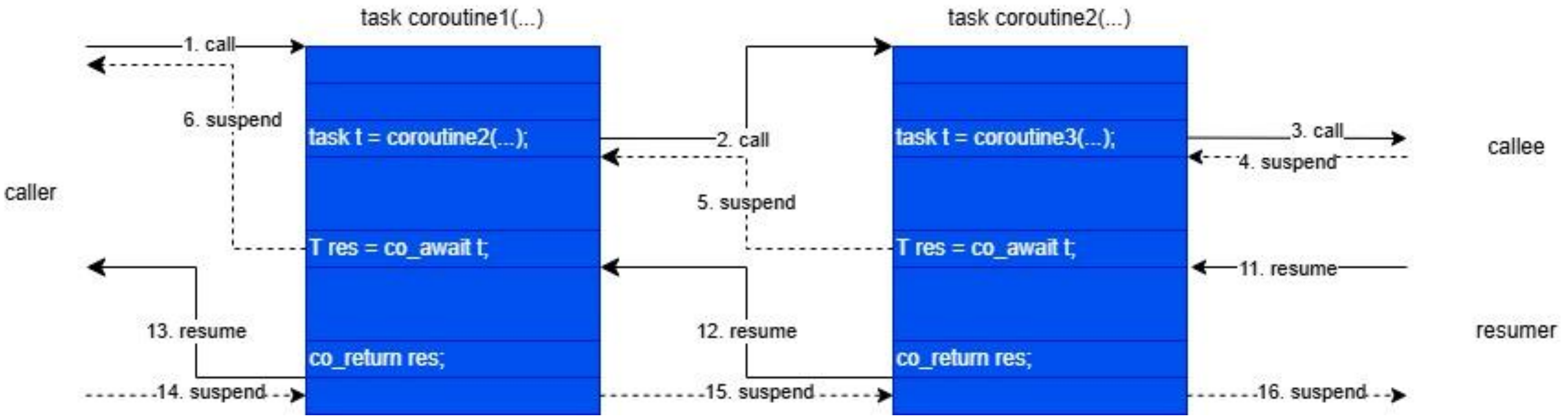


- coroutine1 and coroutine2 run to completion.
- Alternative style: T res = co_await coroutineX(…);
- Notice that the coroutines suspend at the final suspend point (explanation follows below).

- Synchronous completion
  - The leaf coroutine calls only co_return and behaves as an ordinary function.

INTRODUCTION TO C++ COROUTINES

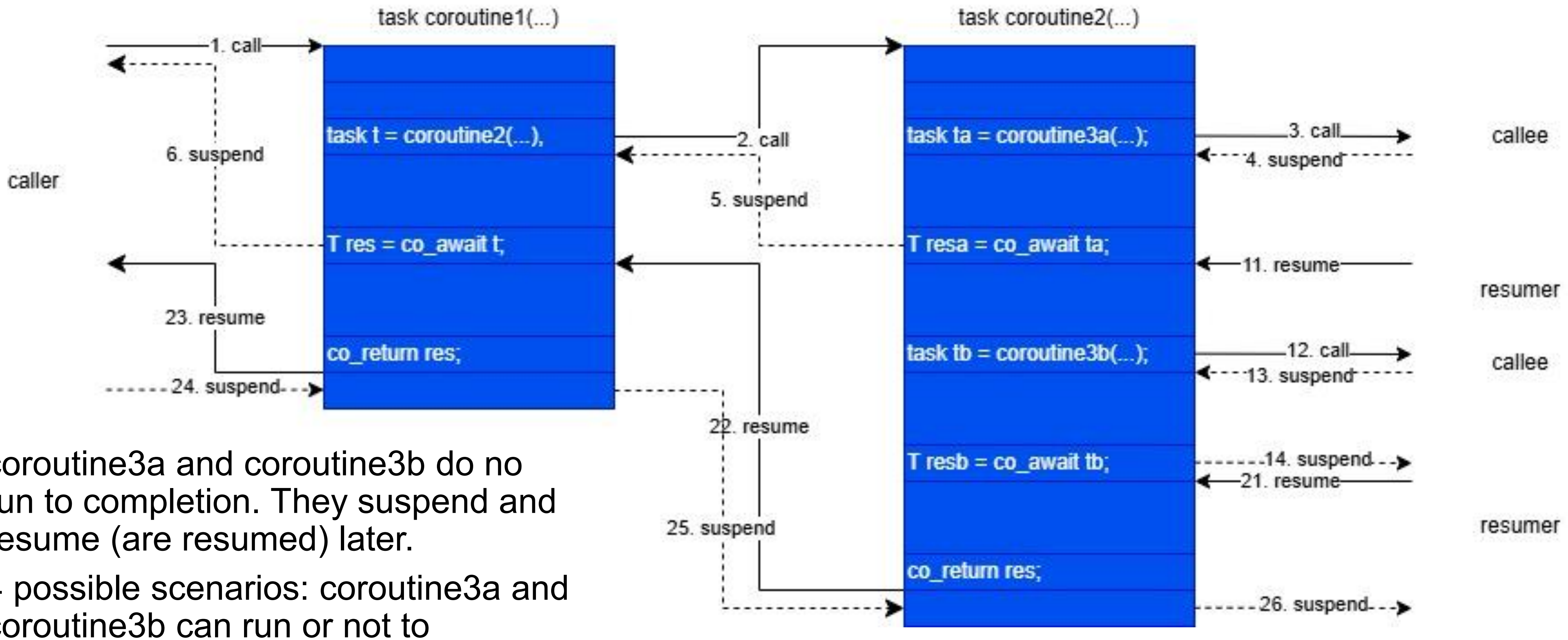
4. Coroutine control flow with asynchronous completion



- coroutine1 and coroutine2 do no run to completion. They suspend and resume (are resumed) later.
- Asynchronous completion
  - The leaf coroutine calls co_await on an awaitable object and will be suspended and then resumed.

INTRODUCTION TO C++ COROUTINES

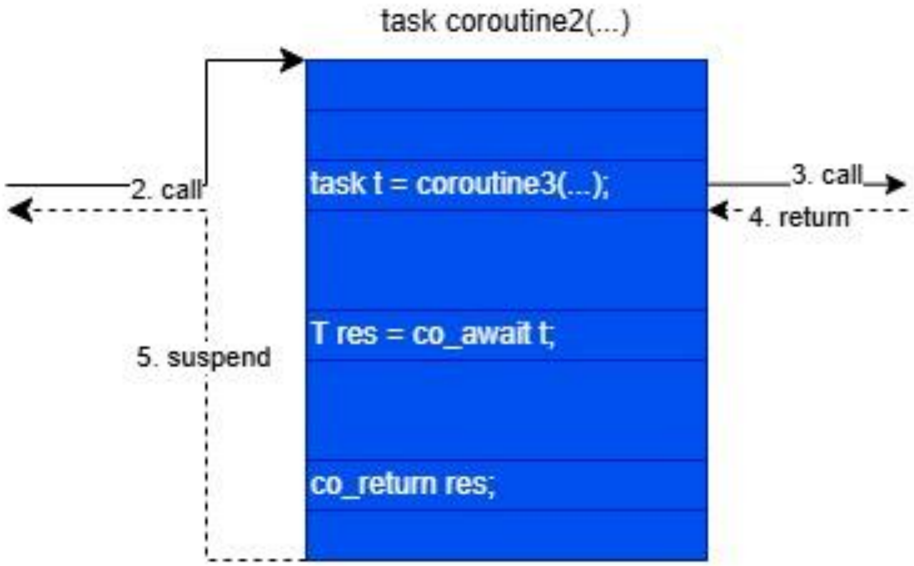
5. Coroutine control flow with 2 x asynchronous completion

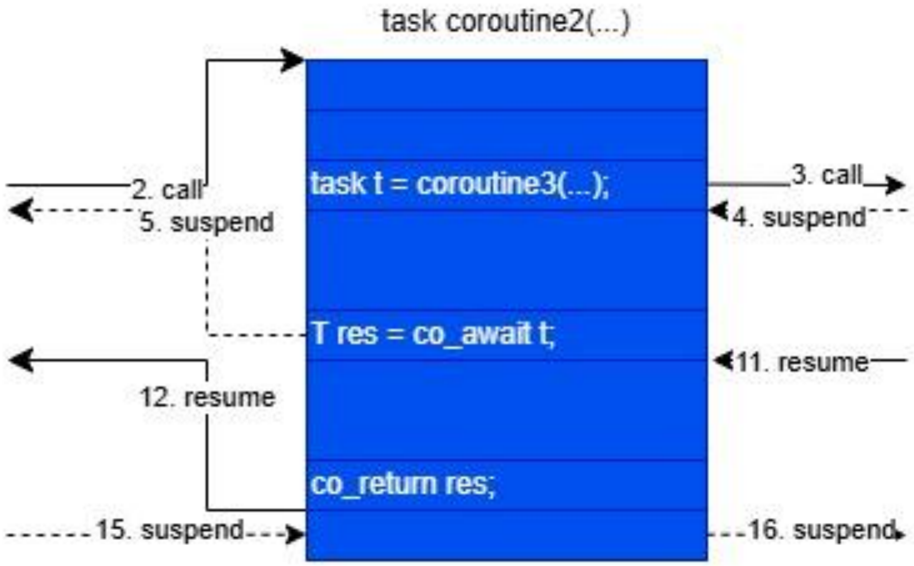


- coroutine3a and coroutine3b do no run to completion. They suspend and resume (are resumed) later.
- 4 possible scenarios: coroutine3a and coroutine3b can run or not to completion.

INTRODUCTION TO C++ COROUTINES

6. Summary





coroutine2 runs to completion.

coroutine2 does not run to completion.

It suspend and resumes (is resumed) later.

INTRODUCTION TO C++ COROUTINES

Further information

- Sorry, folks. This is all you can get from me as an introduction to C++ coroutines.
- Luckily, there are excellent introductions to C++ coroutines available on the Internet:
  - C++20's Coroutines for Beginners - Andreas Fertig - Meeting C++ online
  - https://andreasfertig.com/talks/dl/afertig-2024-meeting-cpp-online-cpp20s-coroutines-for-beginners.pdf
- Let's move on to the application domain… (after 4 technical notes slides).

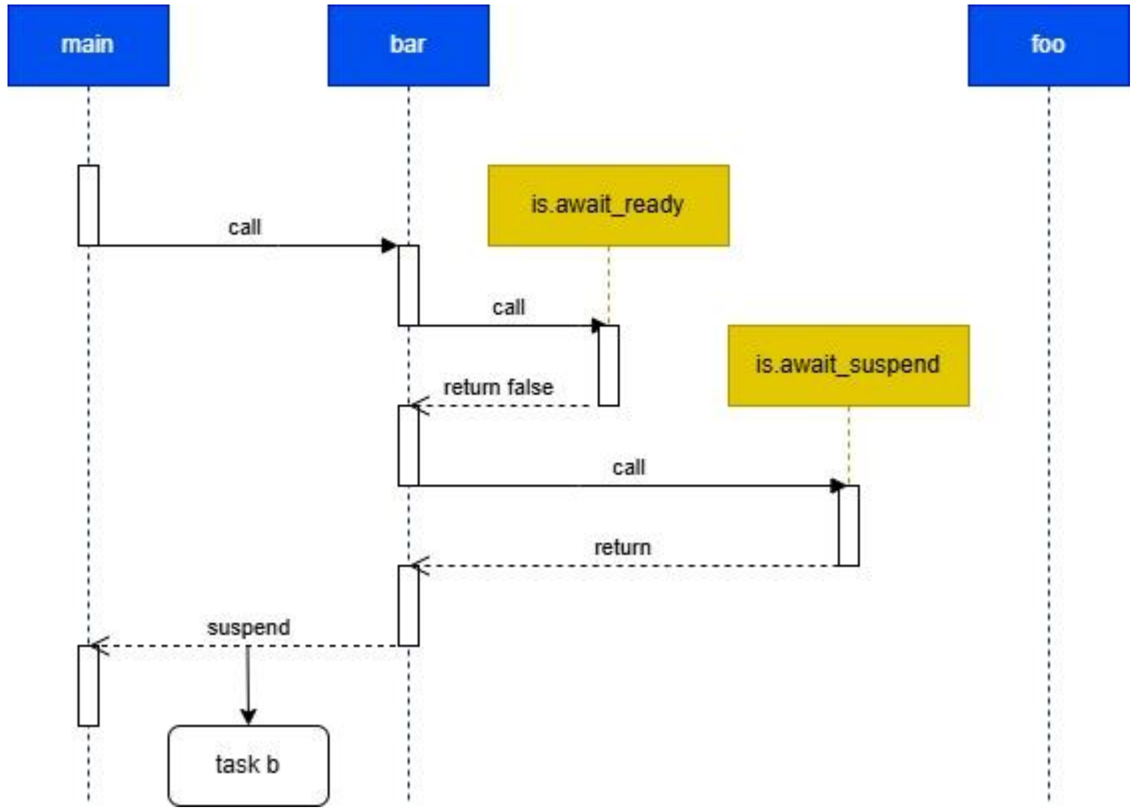
INTRODUCTION TO C++ COROUTINES

Technical note on coroutine start (initial suspend point)

- The scenarios used so far and those that will be used in the rest of the presentation use eager start coroutines.
- Eager start scenarios are easier to understand and illustrate than lazy start scenarios.
- Eager start scenarios allow concentrating more on the control flow inside the body of the coroutine.
- For a more detailed comparison between eager and lazy start coroutines, the reader is referred to the (yet unpublished) presentation “Be lazy or eager? For once, be eager!”

INTRODUCTION TO C++ COROUTINES

Technical note on coroutine start (initial suspend point): lazy start



Using a sequence diagram notation where:

- object names are replaced with function names
- function names on the arrows are replaced with any of 4 operations: call, return, suspend and resume (resume is not in the picture).

INTRODUCTION TO C++ COROUTINES

Technical note on coroutine return (final suspend point)

- The illustrated return flow corresponds to using a final_awaiter type as defined on the left:

```
struct final_awaiter {
    bool await_ready() noexcept {
        return false;
    }
    void await_suspend(coroutine_handle<promise_type> h)
        if (h.promise().continuation)
            h.promise().continuation.resume();
    }
    void await_resume() noexcept {}
};
```

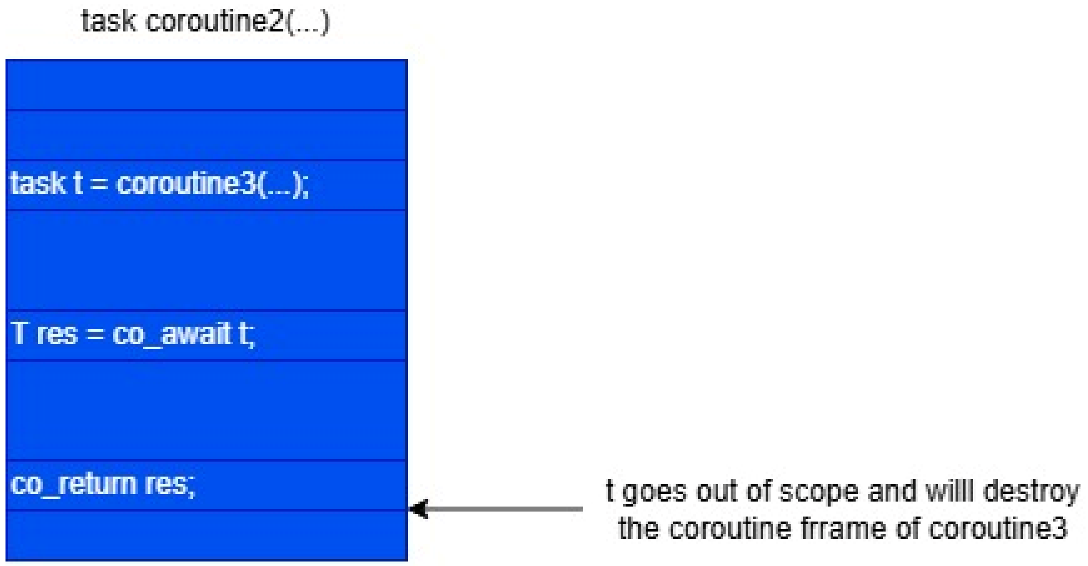
```
struct final_awaiter {
    bool await_ready() noexcept {
        return false;
    }
    coroutine_handle<> await_suspend(coroutine_handle<promise_type> h) noexcept {
        if (h.promise().continuation)
            return h.promise().continuation;
        else
            return std::noop_coroutine();
    }
    void await_resume() noexcept {}
};
```

- With the final_awaiter type defined on the right, each coroutine will be resumed individually from a loop. The latter definition is used in the “symmetric transfer” approach (see understanding_symmetric_transfer).
- This presentation uses the left definition because this control flow corresponds more closely to the asynchronous examples that we will see later in this presentation.
- async_task.h can be compiled to use either of these final_awaiter definitions (and even a 3rd one).
- Use the final_awaiter type on the right if you use lazy start (to avoid possible stack overflow).

INTRODUCTION TO C++ COROUTINES

Technical note on coroutine return (final suspend point)

- With either definition of the final suspend point (see previous slide), a coroutine will always suspend and never return.
- The ‘coroutine state object’ must be destroyed when the ‘task object’ holding a coroutine_handle to the ‘coroutine state object’ goes out of scope:
  - The task object’s destructor must call destroy() on the coroutine_handle.
  - No memory leaks!
- Example (see figure on the right)
  - If coroutine2 holds a task object t to coroutine3 and coroutine2 has reached the end of its body (after a co_return statement), the task object goes out of scope and will destroy coroutine3’s coroutine state object.
  - This happens before coroutine2 reaches the final suspendsection.



AGENDA

1. Brief introduction to C++ coroutines
2. **Brief introduction to (a)synchronous programming (4 slides)**
3. Why use coroutines for asynchronous programming?
4. When not to use coroutines?
5. Summary and conclusions
6. Appendix: brief introduction to corolib

A)SYNCHRONOUS PROGRAMMING / APPLICATIONS

What are the (a)synchronous applications that are the subject of this presentation?

- Applications using
  - (a)synchronous input/output: (async_)open, (async_)connect, (async_)read, (async_)write, (async_)close
  - (a)synchronous remote procedure call (RPC) / remote method invocation (RMI): use the operations listed in the previous point in their implementation
- that are usually long-running
  - by entering an event loop from where they process events from the user, environment and other applications in the system
- and that are single-threaded
  - unless an aggregate application combines several smaller applications running each of these applications on a different thread

(A)SYNCHRONOUS PROGRAMMING / APPLICATIONS

Synchronous versus asynchronous programming

- This presentation deals with (potentially) **long-running** applications with frequent **input/output** (I/O) to a file system or to other applications.
- We want our applications to be **reactive**, i.e., to be able to react quickly to new input from the environment (e.g., the user) or from other applications (in a distributed system).
- A **synchronous** application waits (internally) to the response after every I/O request.
  - Uses open(), connect(), read(), write(), … or a remote procedure call (RPC) that looks (much) like a local procedure call.
  - Advantage: simple style, easy to implement and maintain.
  - Disadvantage: not reactive to new inputs: may be unacceptable for some applications.
- An **asynchronous** application does not wait (immediately) to the response after an I/O request.
  - Uses async_open(), async_connect(), async_read(), async_write, … and a similar style for RPCs.
  - Advantage: reactive to new inputs.
  - Disadvantage: sometimes very fragmented programming style, making it more difficult to implement and maintain.
- Can we combine the benefits of both styles while avoiding the disadvantages?
  - This presentation will (try to) demonstrate that coroutines are the solution.

(A)SYNCHRONOUS PROGRAMMING / APPLICATIONS

How does an asynchronous application await and handle the response to (an) outstanding request(s)?

- Using **callbacks**
  - A callback function is passed as argument to the asynchronous request function.
    - This callback function is then passed further on to the infrastructure (operating system, communication framework).
  - Alternative: a callback function is associated with an API at application startup.
  - The callback function will be called
    - from a dedicated “completion” thread
    - or on the same thread after the application has entered an event loop (see also next slide).
- Using **polling functions**
  - The application calls a function that returns the result (as return value or as output arguments).
  - Polling functions can be blocking or non-blocking.
- Using **events/messages**
  - The application enters an event/message loop from where it passes events/messages to the application.
  - At stated above: callback functions can also be called from an event loop.

(A)SYNCHRONOUS PROGRAMMING / APPLICATIONS

Event loops

- Event/message loops can be local or global:
  - Local: can occur at many places in the application.
  - Global: occurs at a single place in the application.
- Event/message loops can be …
  - never-leaving: once entered, application functions are called from this loop. Typically used in communication frameworks.
  - left after each event/message:
    - the event/message is passed to the application for further processing, or
    - after having processed the event/message in the loop function, e.g., after calling a callback function.
- Notes:
  - Local event loops cannot be never-leaving.
  - Local event loops may restrict the type of events they handle and postpone all other events to the global event loop.
- Conclusion: many different styles are possible in asynchronous programming.
- Can we return to a single style using coroutines, pushing the differences to a library or framework level?

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. **Why use coroutines for asynchronous programming?**
4. When not to use coroutines?
5. Summary and conclusions
6. Appendix: brief introduction to corolib

WHY USE COROUTINES?

Introduction

- This section contains several examples ranging from simple to more complex.
- To obtain even more complex examples, it is possible to combine several examples, such as combining a call stack with sequential RMIs.
- The presentation uses the term **completion handler**:
  - A function that completes an asynchronous operation, i.e., contains its response; often implemented as a callback function.
- The examples do not use any real (asynchronous) **communication framework**
  - … apart from a few examples using Qt5 and gRPC.
- Characteristics of the simulated communication framework:
  - It is single-threaded: the completion handler runs on the same thread as the rest of the application.
  - There is an explicit event loop where the application code returns to, and from where the completion handler is run.
- If the completion handler runs on a different thread, it is possible to return to the illustrated scenarios by
  - introducing an explicit event loop (if not yet present).
  - let the completion handler put its response as an event in the event queue, from where it will be picked up in the next iteration in the event loop.

WHY USE COROUTINES?

Overview: asynchronous I/O and communication examples

Lower level: I/O

- Function/coroutine with (a)synchronous wait, write or read
  - (A)synchronous wait
  - (A)synchronous write with (a)synchronous read

Upper level: RMI/RPC

- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 sequential RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- Function with RMI inside nested loop + segmentation

WHY USE COROUTINES?

Overview: other uses of coroutines

- As an alternative to threads
- In embedded software
- In other programming languages (Python, C#, …)
- Completion on another thread
- Local event loop
- CORBA: synchronous – callback – polling – coroutines
- Communicating finite state machines

WHY USE COROUTINES?

Overview: asynchronous I/O and communication examples

Lower level: I/O

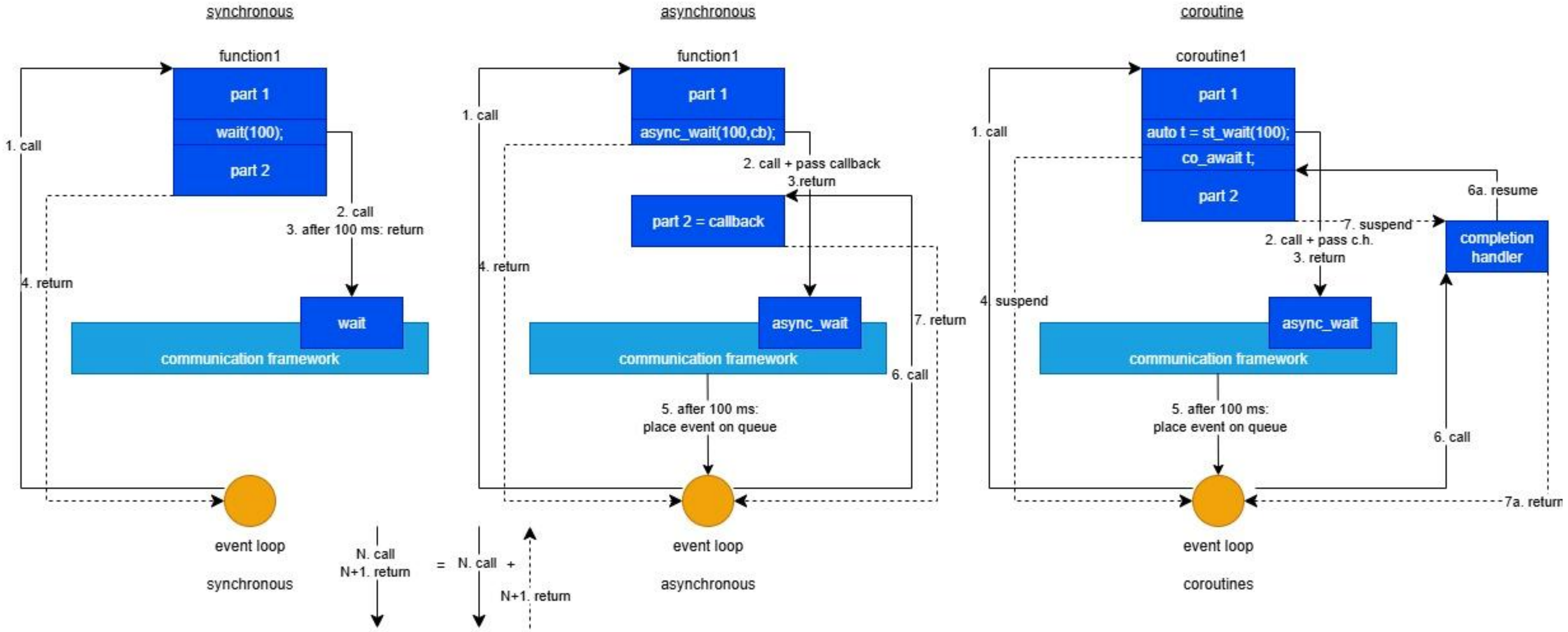
- **Function/coroutine with (a)synchronous wait, write or read**
  - (A)synchronous wait (4 slides)
  - (A)synchronous write with (a)synchronous read (5 slides)

Upper level: RMI/RPC

- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 sequential RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- Function with RMI inside nested loop + segmentation

WHY USE COROUTINES?

Function/coroutine with (a)synchronous wait (1/4)



WHY USE COROUTINES?

Function/coroutine with (a)synchronous wait (2/4)

Synchronous

- function1 has 2 parts with a wait() call in between. Part 2 is executed after a delay of 100 units (e.g., ms) introduced by the call of wait(100).
- The wait() call blocks the thread that calls function1 and the caller(s) of function1 (if any).
  - This thread could be the only thread in the application.

Asynchronous

- function1 has been split into 2 smaller parts. The first part still belongs to function1.
- The second part is placed in a new function whose address is passed as the second argument to async_wait.
- function1 returns control after having called async_wait
  - async_wait and function1 do not block the calling thread.
- The application (re)enters the event loop.
- After 100 units, the timer expires. The communication framework places an event in the event queue.
- The event loop calls the callback function (that contains the second part of function1).

WHY USE COROUTINES?

Function/coroutine with (a)synchronous wait (3/4)

Coroutines

- function1 has been replaced with coroutine1. coroutine1 contains the 2 parts from function1.
- coroutine1 calls st(art)_wait(100), which returns an object t.
  - The implementation of st_wait calls async_wait() and registers a generic completion handler (i.e., a function that is devoid of any application code) with the communication framework.
  - Because async_wait() is a non-blocking call, st_wait() is non-blocking as well.
- coroutine1 co_awaits t.
- Because the timer has not yet expired, the coroutine suspends and returns control to its caller.
- The application (re)enters the event loop.
- After 100 units, the timer expires. The communication framework places an event in the event queue.
- The event loop calls the completion handler. The completion handler resumes coroutine1, which runs to completion.

WHY USE COROUTINES?

Function/coroutine with (a)synchronous wait (4/4)

Comparison

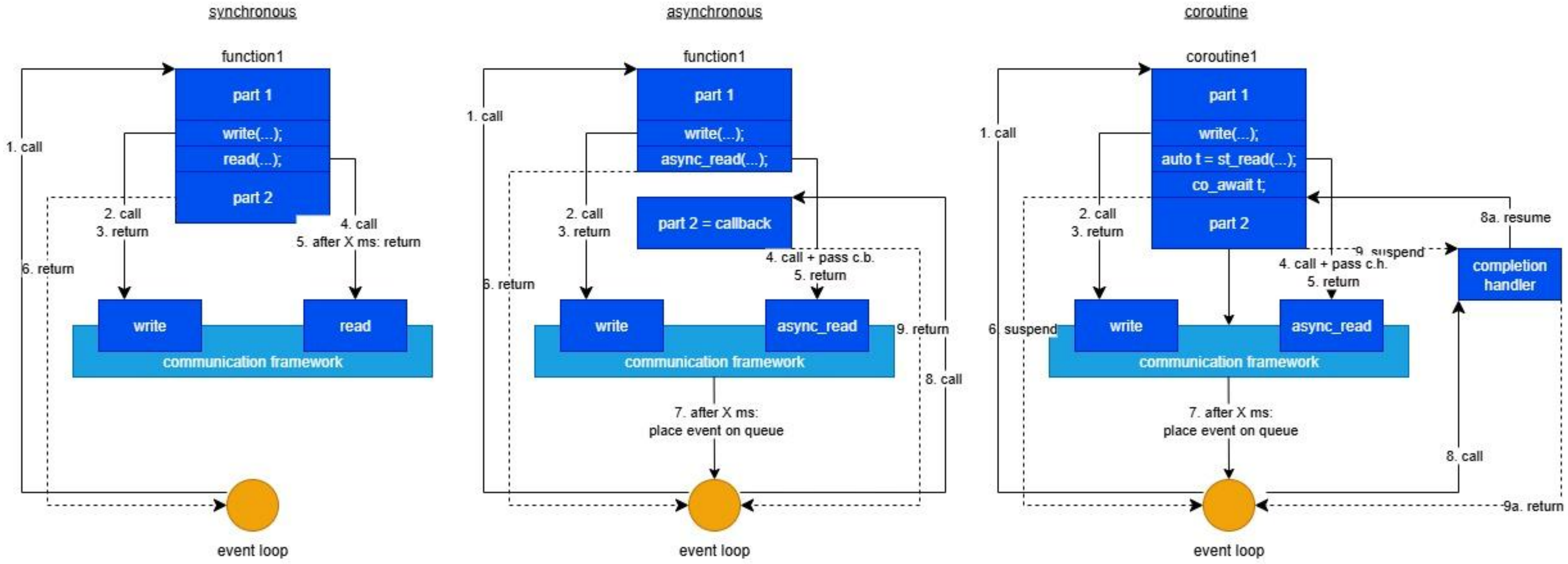
- The synchronous and coroutine version are very close to each other in coding style: there is only one function/coroutine with part 2 following part 1.
- The asynchronous and coroutine version are very close in their control flow, but the coroutine version hides the difficult part from the application developer. The C++ compiler does the difficult work, together with a coroutine library that provides st_wait.

Note

- Because the implementation of an asynchronous wait requires the use of a communication framework (such as Boost ASIO), and most examples in this presentation do not use any real communication framework, there is no source code for this example.

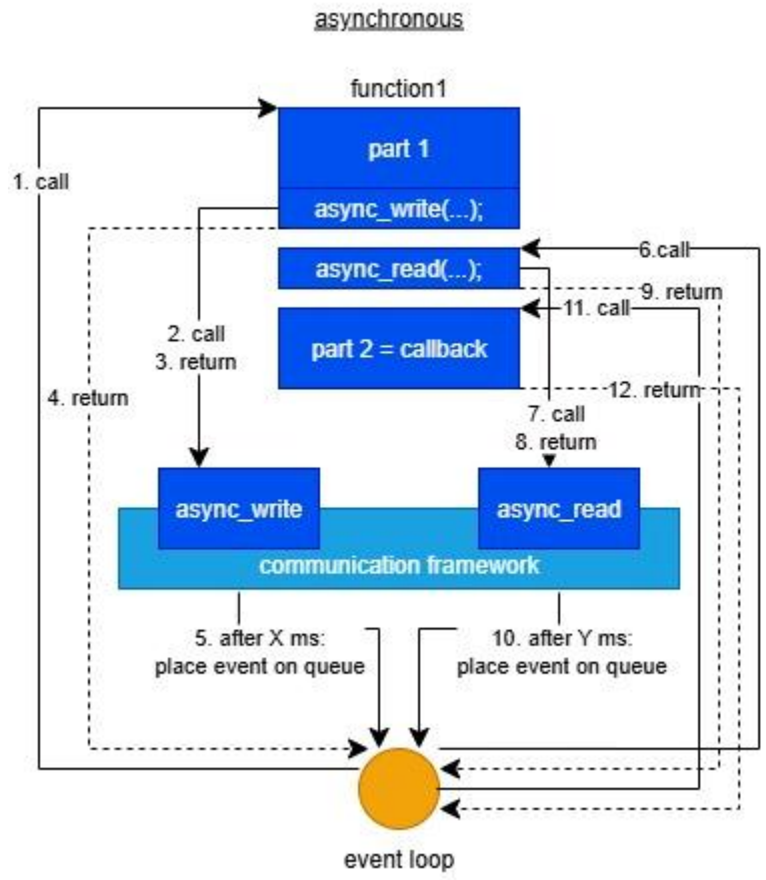
WHY USE COROUTINES?

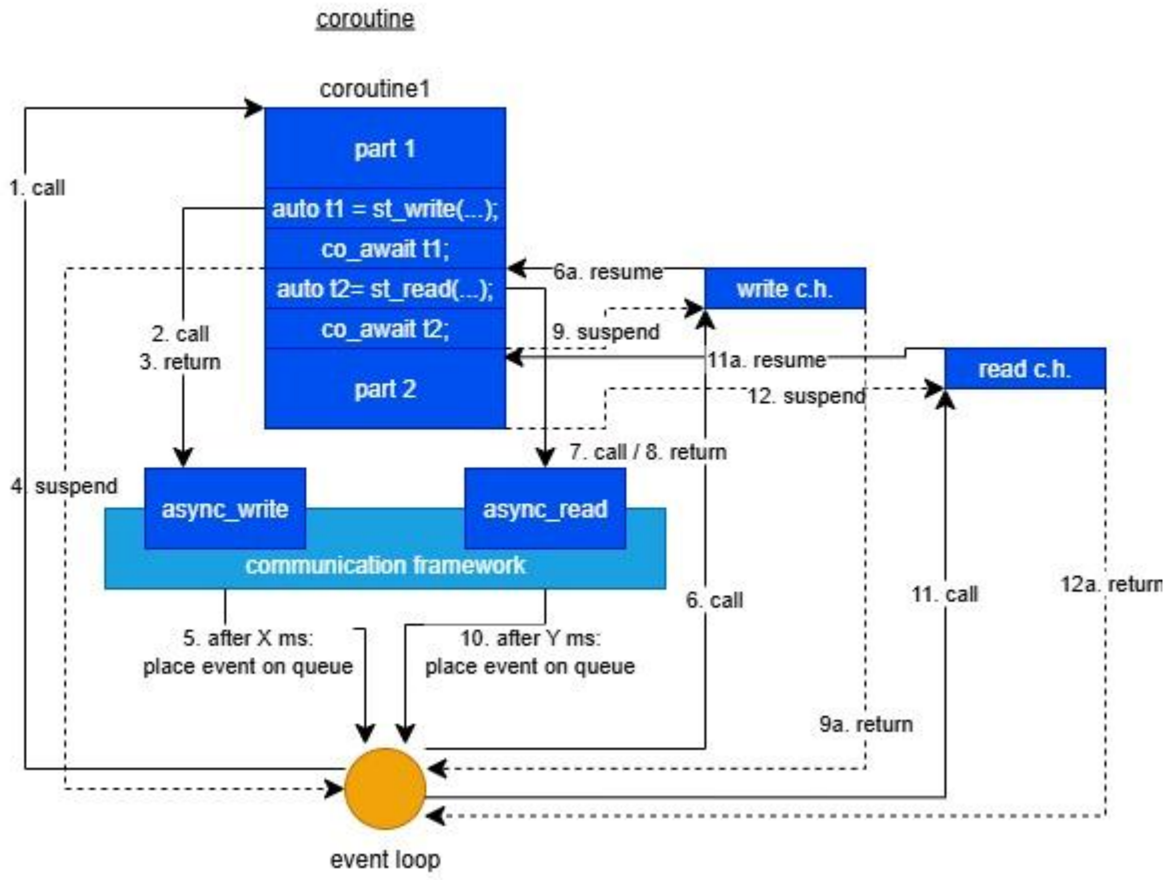
Function/coroutine with synchronous write and asynchronous read (1/5)



WHY USE COROUTINES?

Function/coroutine with asynchronous write and asynchronous read (2/5)





WHY USE COROUTINES?

Function/coroutine with (a)synchronous write and (a)synchronous read (and other) (3/5)

- The control flow of (a)synchronous read and write is similar to the control flow of the (a)synchronous wait. Therefore, the explanation has been omitted (see previous slides).
- The source code examples listed in the next slides use, in addition to (a)synchronous read and write, also (a)synchronous versions of create, open, close and remove.
  - For communication frameworks, connect is used instead of open.
- The examples use simulated read, write, …, operations: there is no standardized asynchronous I/O library.

WHY USE COROUTINES?

Function/coroutine with (a)synchronous write and (a)synchronous read (and other) (4/5)

Synchronous:

- Basic examples: p1000-sync.cpp, p1001-sync.cpp (The second example uses a second write after the read.)
- Running the ‘synchronous’ function on a dedicated thread to simulate 2 parallel accesses: p1010-sync-thread.cpp, p1011-sync-thread.cpp

Asynchronous:

- Basis examples using callback functions: p1020-async.cpp, p1021-async.cpp
- Examples using lambdas in lambda in lambdas, etc.: p1025-async.cpp, p1026-async.cpp
- Examples running the completion handlers on a dedicated thread: p1030-async-thread.cpp, p1031-async-thread.cpp

WHY USE COROUTINES?

Function/coroutine with (a)synchronous write and (a)synchronous read (and other) (5/5)

Coroutines (using a simple coroutine implementation)

- Examples: p1040-coroutine.cpp, p1041-coroutine.cpp, p1042-coroutine.cpp

Coroutines (using corolib):

- Examples: p1060-corolib.cpp, p1061-corolib.cpp, p1062-corolib.cpp

Coroutines (using corolib with the completion handler running on a dedicated thread):

- Examples: p1070-corolib-thread.cpp, p1071-corolib-thread.cpp, p1072-corolib-thread.cpp

Remarks on using a dedicated thread

- When running the callback function of an asynchronous function on a dedicated thread, it is possible to restore the synchronous/sequential style of synchronous applications by synchronizing the thread (1) that has called the asynchronous function with the thread (2) on which the callback function runs.
  - E.g., using a binary semaphore, thread (1) acquires the semaphore which is released by thread (2).
  - However, this introduces local waiting points inside the application.
- Using coroutines, only one thread is needed without introducing local waiting points.

WHY USE COROUTINES?

Following examples

- The following examples will use RMI (remote method invocation).
  - In this context, RMI is equivalent to RPC (remote procedure call).
  - RMI uses more the original object-oriented terminology (Smalltalk, remember).
- On the client side, a RMI involves one or more (a)synchronous writes and one or more (a)synchronous reads.
  - … depending on the number of segmentations and reassemblies.
- A RMI may also involve setting up a connection (connect) before the writes and reads, followed by closing the connection.
- The following examples will only use a single write and a single read.

WHY USE COROUTINES?

Overview: asynchronous I/O and communication examples

Lower level: I/O

- Function/coroutine with (a)synchronous wait, write or read

Upper level: RMI/RPC

- **Function with 1 RMI (5 slides)**
- Call stack + function with 1 RMI
- Function with 3 sequential RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- Function with RMI inside nested loop + segmentation

WHY USE COROUTINES?

Function/coroutine with 1 RMI (1/5): synchronous – asynchronous – coroutine

```
// Synchronous
int function1(int in1, int in2)
{
    int out1 = -1, out2 = -1;
    int ret1 = remoteObj1.op1(in1, in2, out1, out2);
    return in1 + in2 + out1 + out2 + ret1;
}
```

```
// Asynchronous
struct function1_ctxt_t
{
    int in1;
    int in2;
    int* ret;
};

void function1alt2(int in1, int in2, int& ret)
{
    function1_ctxt_t* ctxt = new function1_ctxt_t{ in1, in2, &ret };
    remoteObj1.sendc_op1(in1, in2,
        [ctxt](int out1, int out2, int ret1)
        {
            *ctxt->ret = ctxt->in1 + ctxt->in2 + out1 + out2 + ret1;
            delete ctxt;
        });
}
```

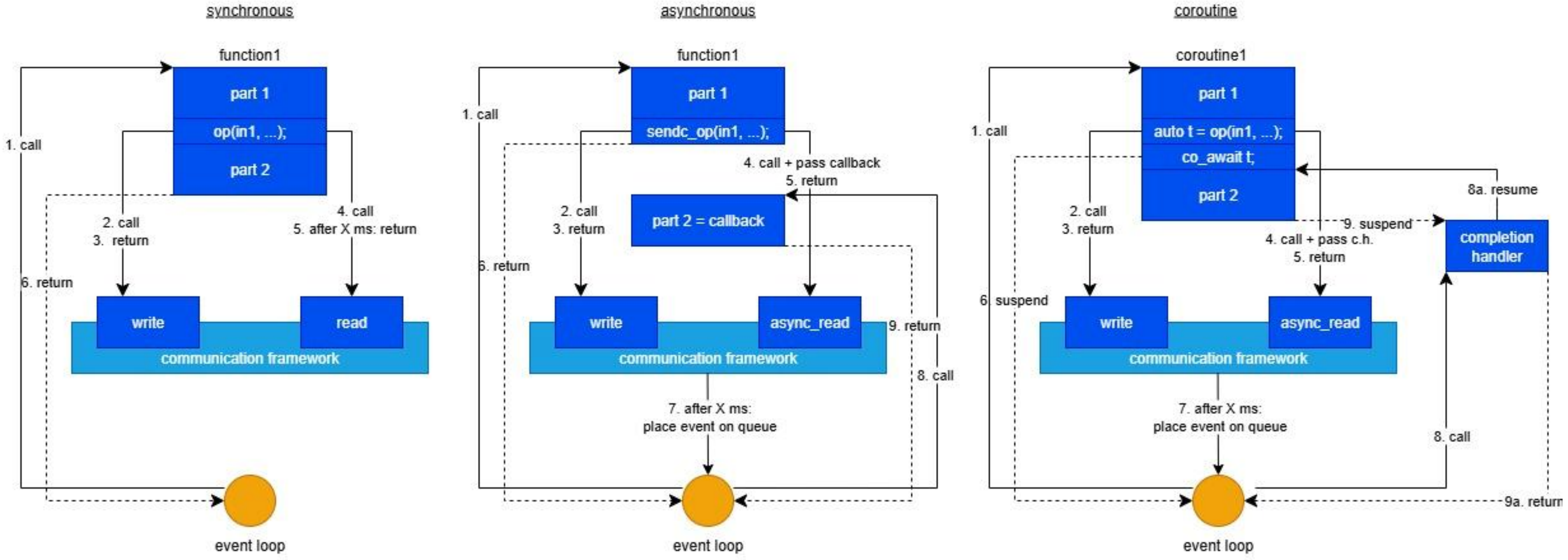
```
// Coroutine
async_task<int> coroutine1(int in1, int in2)
{
    int out1 = -1, out2 = -1;
    int ret1 = co_await remoteObj1co.op1(in1, in2, out1, out2);
    co_return in1 + in2 + out1 + out2 + ret1;
}
```

The synchronous and coroutine examples are syntactically very close to each other and have been placed side by side.

Asynchronous example: many styles are possible, with explicit or anonymous (= lambda) callback functions, or with a single function implemented as a state machine, the way the C++ compiler compiles a coroutine.

WHY USE COROUTINES?

Function/coroutine with 1 RMI (2/5) (see next slides for explanation)



WHY USE COROUTINES?

Function/coroutine with 1 RMI (3/5): synchronous style

- The RMI op(in1, …) is …
  - writing the request as a series of bytes/bits (onto the connection) to the server
  - and reading the response byte/bit stream from the server.
- While waiting for the response to arrive, the application cannot process any other inputs.
  - function1 does not return to the global event loop.
- This type of application is not reactive.
- Source code: p1000-sync-1rmi.cpp , p1050-sync-1rmi.cpp (with code to simulate write and read operations)

- When possible, run independent RMIs on different threads to improve the throughput.
- Source code: p1002-sync+thread-1rmi.cpp, p1004-sync+thread-1rmi.cpp

WHY USE COROUTINES?

Function/coroutine with 1 RMI (4/5): asynchronous style

- The synchronous function (see previous slide) is split into two functions.
- The first function …
  - contains the original code up till the point of the RMI
  - sends a request sendc_op(in1, …) with the input arguments of the RMI
  - registers a second function (see next point) with the communication framework
  - returns control to the global event loop to handle the response (if completion is on the same thread).
- The second function (completion handler, here implemented as a lambda and used as a callback function) …
  - handles the output arguments and return value of the RMI
  - contains the code that followed the RMI in the synchronous function.
- When the completion event arrives, the completion handler is called back.
- Many styles are possible, including using a single function implemented as a state machine.
- Source code: completion on the same thread: p1010-async-1rmi.cpp, p1060-async-1rmi.cpp
- Source code: completion on a dedicated thread: p1015-async-thread-1rmi.cpp (synchronous application!)

WHY USE COROUTINES?

Function/coroutine with 1 RMI (5/5): coroutines (corolib)

- The implementation of op1(in1, …) registers a completion handler with the communication framework.
  - This completion handler is application-independent (i.e., it does not contain any application-specific code).
- At the co_await statement, the coroutine suspends itself and returns control to the main event loop (if the response has not arrived yet).
- When the completion event arrives, the completion handler is called back.
- The completion handler passes the response and resumes the coroutine.
- The original code does not have to be restructured.
- Everything runs on the same thread (if a single-threaded asynchronous communication framework is used).
- Source code: p1020-coroutines-1rmi.cpp, p1070-coroutines-1rmi.cpp

WHY USE COROUTINES?

Overview: asynchronous I/O and communication examples

Lower level: I/O

- Function with (a)synchronous wait, write and read

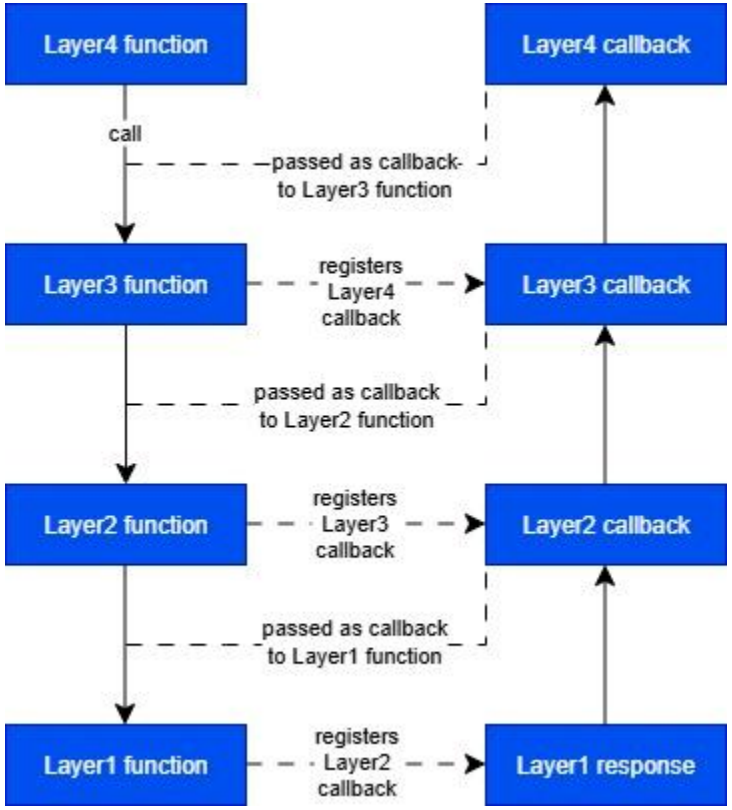
Upper level: RMI/RPC

- Function with 1 RMI
- **Call stack + function with 1 RMI (7 slides)**
  - The call stack can be an application call stack, a protocol call stack, a device driver call stack
- Function with 3 RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- Function with RMI inside nested loop + segmentation

WHY USE COROUTINES?

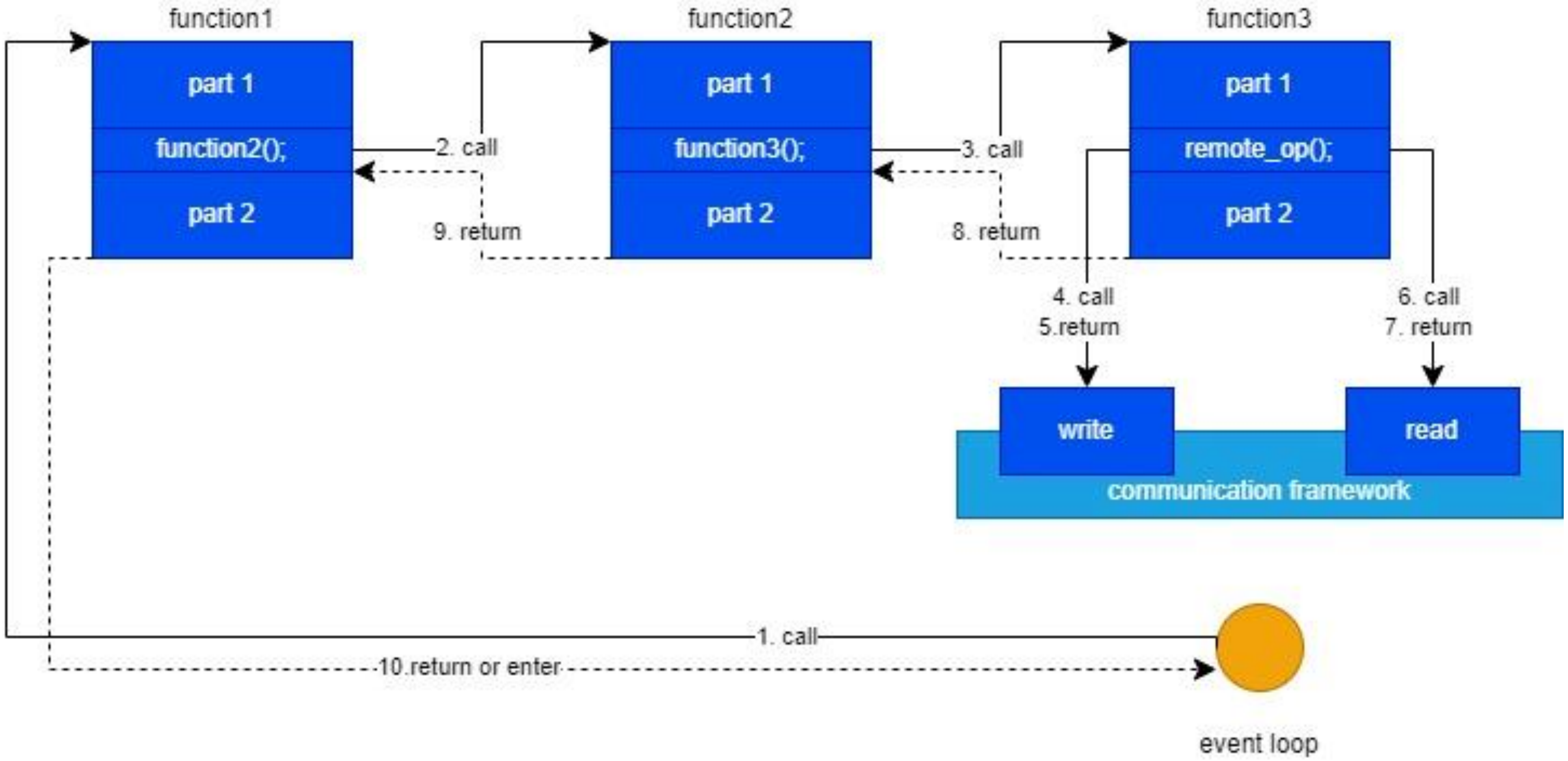
Call stack + function with 1 RMI (1/7): real world example

- ISO/OSI (International Organization for Standardization / Open Systems Interconnection) based protocol stack.
- Example: BLE (Bluetooth Low Energy) stack implemented in C.
- Function at layer N calls a function at the lower layer N-1 and passes a callback function to that layer N-1 function.
- Several layer N functions can call the same layer N-1 function.
  - E.g., TCP and UDP are at layer 4, IP at layer 3.
- The layer 1 function writes the request onto the physical medium but does not wait for a response.
- The response is handled by the registered callback functions in the opposite direction.
- The callback functions must find their way back to the correct top layer.
- Other example: device driver stack.



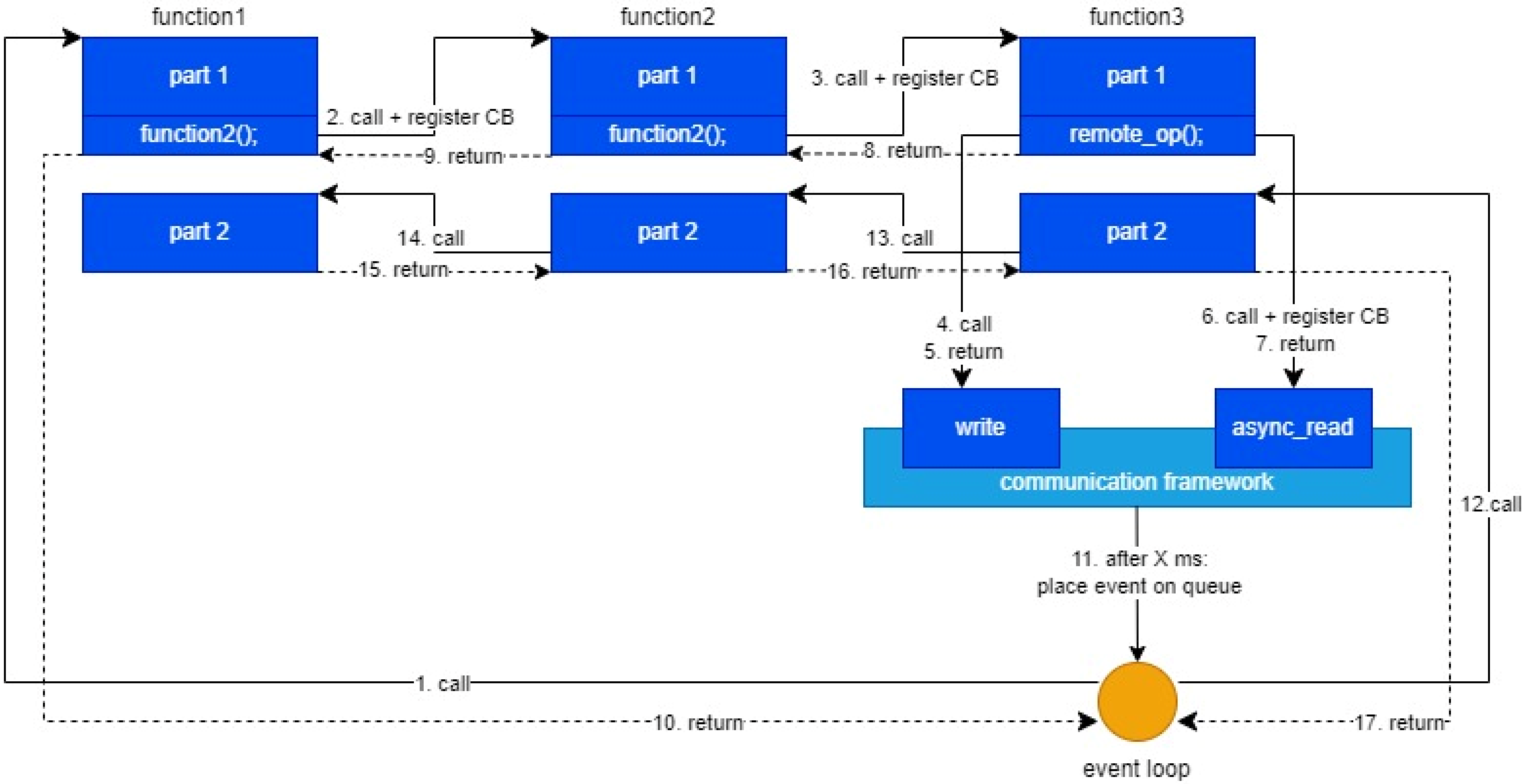
WHY USE COROUTINES?

Call stack + function with 1 RMI (2/7): synchronous style



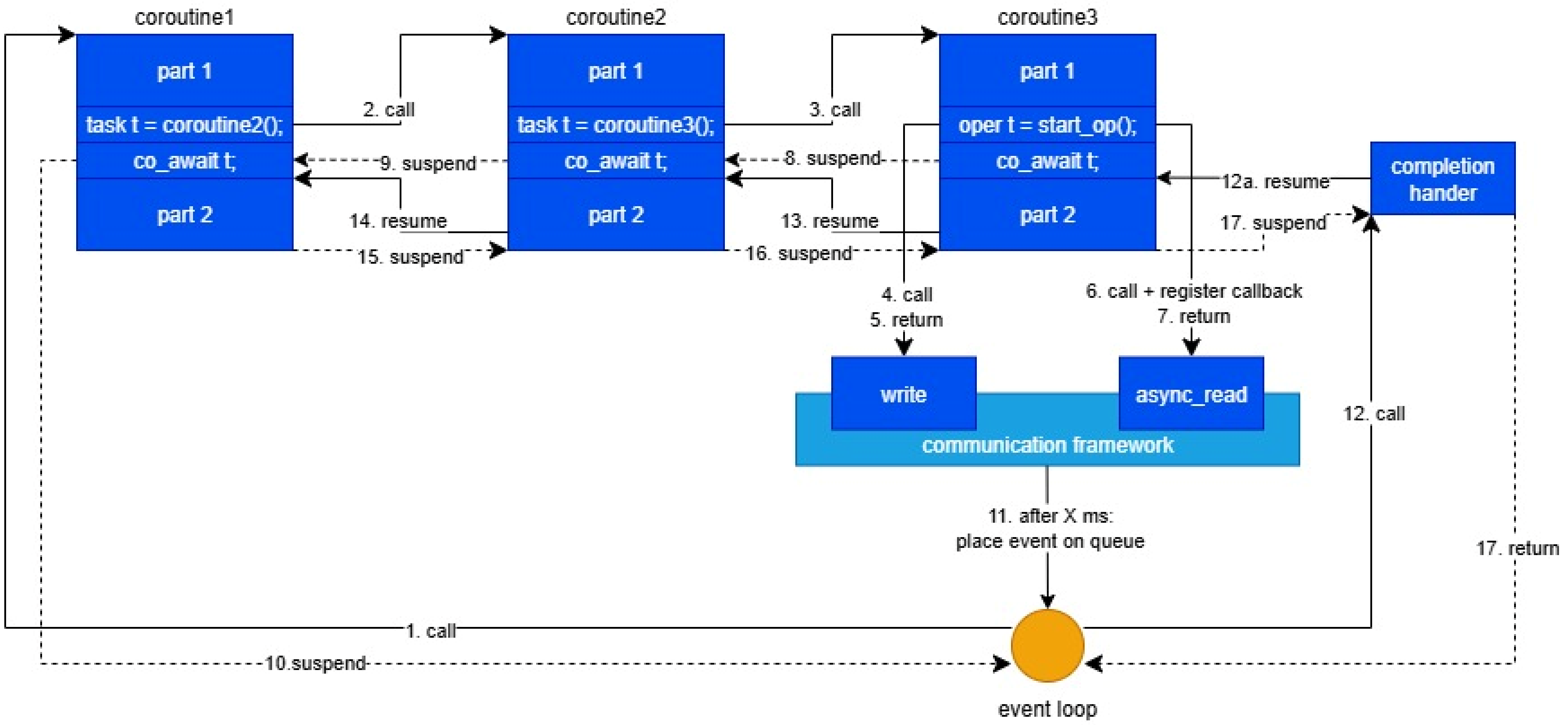
WHY USE COROUTINES?

Call stack + function with 1 RMI (3/7): asynchronous style



WHY USE COROUTINES?

Call stack + function with 1 RMI (4/7): coroutines



WHY USE COROUTINES?

Call stack + function with 1 RMI (5/7) (see previous slides for pictures)

Synchronous

- Natural style, but not reactive.
- Source code: p1100-sync-callstack-1rmi.cpp, p1150-sync-callstack-1rmi.cpp

Asynchronous

- The original functions must be split into a “forward” function and a “backward” function.
- The backward functions form a chain of callback functions traversed in reverse order.
- Note: the stack of the forward functions has unrolled when the backward functions are called and cannot be used to store information for the backward direction: this information is typically stored in “context” objects allocated on the heap.
- Reactive application, but it is more difficult to follow the control flow.
- Source code: p1110-async-callstack-1rmi.cpp, p1160-async-callstack-1rmi.cpp
- Source code: p1112-async-callstack-1rmi-cs.cpp

WHY USE COROUTINES?

Call stack + function with 1 RMI (6/7) (see previous slides for pictures)

Coroutines

- Natural style, reactive again.
- Same flow as in the asynchronous case.
- The compiler and the coroutine library do all the hard work.
- Source code: p1120-coroutines-callstack-1rmi.cpp, p1170-coroutines-callstack-1rmi.cpp
- Source code: p1122-coroutines-callstack-1rmi.cpp, p1124-coroutines-callstack-1rmi.cpp, p1126-coroutines-callstack-1rmi.cpp

Coroutines and asynchronous

- What if you have already implemented an asynchronous call stack but want to switch to coroutines?
- It is possible to add a coroutine layer on top of an asynchronous call stack implementation with minor modifications to the top layer.
- Source code: p1130-coroutines-async-callstack-1rmi.cpp, p1132-coroutines-async-callstack-1rmi-cs.cpp

WHY USE COROUTINES?

Call stack + function with 1 RMI (7/7): summary

Synchronous

- Natural function call style, but not reactive: the read operation can block the application for a longer time waiting for the response to arrive.

Asynchronous

- Lots of manual work involved to rewrite a non-reactive synchronous program into a reactive one: introduction of callback functions, passing them on the call stack, …

Coroutines

- Synchronous style program can be turned into a reactive asynchronous style program by using co_await, co_return and a dedicated coroutine return type.
- The control flow is very similar to the asynchronous case. The compiler is doing the hard work.

WHY USE COROUTINES?

Overview: asynchronous I/O and communication examples

Lower level: I/O

- Function with (a)synchronous wait, write or read

Upper level: RMI/RPC

- Function with 1 RMI
- Call stack + function with 1 RMI
- **Function with 3 sequential RMIs (7 slides)**
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop

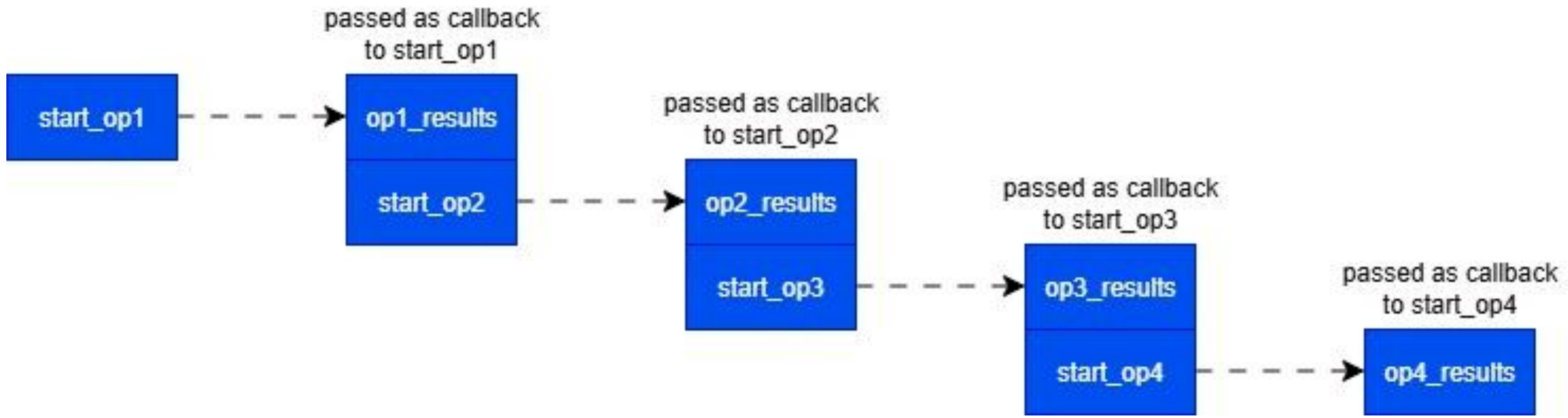
WHY USE COROUTINES?

Function with 3 sequential RMIs (1/7): real world example

- In 2005, a client company decided to rewrite their software using a self-developed asynchronous communication framework (ACF).
- Motivation to introduce a new approach:
  - Avoid problems with multi-threaded applications (imperfect protection of shared variables).
  - The previous system supported asynchronous operations, but the application writer had to explicitly check the completion of an operation using polling. This was sometimes forgotten or ignored; not doing this usually did not cause any problems.
  - Applications mixed business logic with TCP/IP communication (including setting up and closing connections).
- New applications must all be single-threaded, communicating with each other using the ACF.
  - The ACF hides the TCP/IP communication between applications.
- The new approach led to a specific programming style, that can be described as a “chain of callback functions.”
  - See next slide for an illustration.
- All applications use a single, global event loop, from where the callback functions are called.
- I have encountered this programming style at many other clients.

WHY USE COROUTINES?

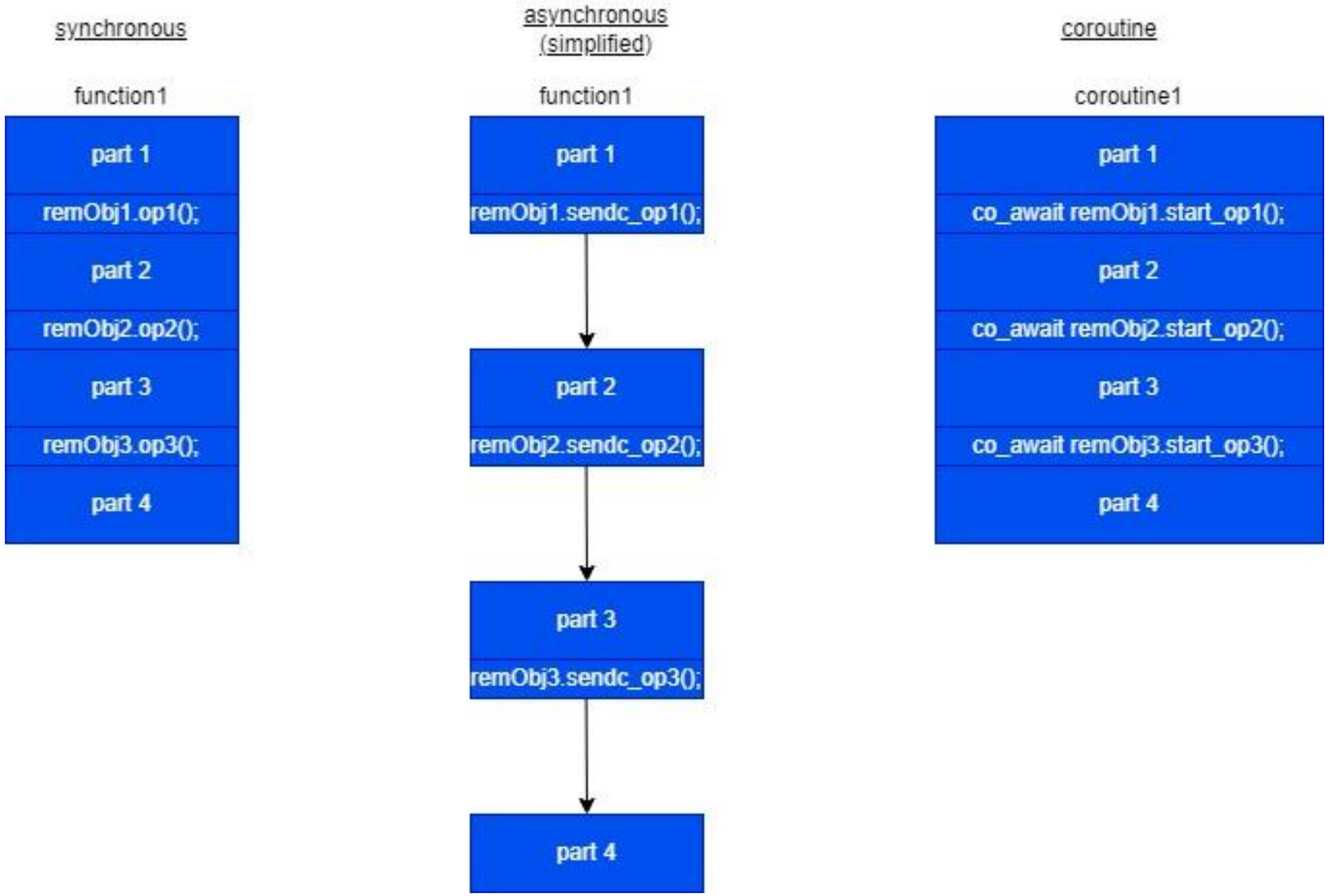
Function with 3 sequential RMIs (2/7): real world example



- A callback function first handles the response of an operation (first part) and then starts one or more new operations (second part).
- This new operation(s) may seem unrelated when just looking at the content of the callback function.
- Usually, a context variable is passed containing information that is used in several parts of the chain.
- This style of programming was my main motivation to start studying C++ coroutines.

WHY USE COROUTINES?

Function with 3 sequential RMIs (3/7)



WHY USE COROUTINES?

Function with 3 sequential RMIs (4/7): synchronous – coroutine

```
int function1(int in1, int in2, int testval)
{
    int out1 = -1, out2 = -1;
    int ret1 = remoteObj1.op1(in1, in2, out1, out2);
    // 1 Do stuff
    if (ret1 == testval) {
        int out3 = -1;
        int ret2 = remoteObj2.op2(in1, in2, out3);
        // 2 Do stuff
        return ret2;
    }
    else {
        int out4 = -1, out5 = -1;
        int ret3 = remoteObj3.op3(in1, out4, out5);
        // 3 Do stuff
        return ret3;
    }
}
```

```
async_task<int> coroutine1(int in1, int in2, int testval)
{
    int out1 = -1, out2 = -1;
    int ret1 = co_await remoteObj1co.op1(in1, in2, out1, out2);
    // 1 Do stuff
    if (ret1 == testval) {
        int out3 = -1;
        int ret2 = co_await remoteObj2co.op2(in1, in2, out3);
        // 2 Do stuff
        co_return ret2;
    }
    else {
        int out4 = -1, out5 = -1;
        int ret3 = co_await remoteObj3co.op3(in1, out4, out5);
        // 3 Do stuff
        co_return ret3;
    }
}
```

The synchronous and coroutine examples are syntactically very close to each other and have been placed side by side. The asynchronous example is more complex, see next slide.

WHY USE COROUTINES?

Function with 3 sequential RMIs (5/7): async.

```
struct function1_ctxt_t
{
    int in1;
    int in2;
    int testval;
};

void function1(int in1, int in2, int testval, int& ret)
{
    function1_ctxt_t* ctxt = new function1_ctxt_t{in1, in2,
                                                   testval};
    remoteObj1.sendc_op1(in1, in2,
        [this, ctxt, &ret](int out1, int out2, int ret1) {
            this->function1a(ctxt, out1, out2, ret1, ret);
        });
    // 1a Do stuff that doesn't need the result of the RMI
}
```

```
void function1a(function1_ctxt_t* ctxt, int out1, int out2, int ret1,
                int& ret)
{
    // 1b Do stuff that needs the result of the RMI
    if (ret1 == ctxt->testval) {
        remoteObj2.sendc_op2(ctxt->in1, ctxt->in2,
            [this, &ret](int out1, int ret1) {
                this->function1b(out1, ret1, ret);
            });
        // 2a Do stuff that doesn't need the result of the RMI
    }
    else {
        remoteObj3.sendc_op3(ctxt->in1,
            [this, &ret](int out1, int out2, int ret1) {
                this->function1c(out1, out2, ret1, ret);
            });
        // 3a Do stuff that doesn't need the result of the RMI
    }
    delete ctxt;
}

void function1b(int out3, int ret2, int& ret)
{
    // 2b Do stuff that needs the result of the RMI
    ret = ret2;
}

void function1c(int out4, int out5, int ret3, int& ret)
{
    // 3b Do stuff that needs the result of the RMI
    ret = ret3;
}
```

WHY USE COROUTINES?

Function with 3 sequential RMIs (6/7): asynchronous (alternative with lambdas)

```
void function1alt(int in1, int in2, int testval, int& ret)
{
    function1_ctxt_t* ctxt = new function1_ctxt_t{ in1, in2, testval };
    remoteObj1.sendc_op1(in1, in2,
        [this, ctxt, &ret](int out1, int out2, int ret1) {
            // 1b Do stuff that needs the result of the RMI
            if (ret1 == ctxt->testval) {
                remoteObj2.sendc_op2(ctxt->in1, ctxt->in2,
                    [this, &ret](int out3, int ret2) {
                        // 2b Do stuff that needs the result of the RMI
                        ret = ret2;
                    });
                // 2a Do stuff that doesn't need the result of the RMI
            }
            else {
                remoteObj3.sendc_op3(ctxt->in1,
                    [this, &ret](int out4, int out5, int ret3) {
                        // 3b Do stuff that needs the result of the RMI
                        ret = ret3;
                    });
                // 3a Do stuff that doesn't need the result of the RMI
            }
            delete ctxt;
        });
    // 1a Do stuff that doesn't need the result of the RMI
}
```

WHY USE COROUTINES?

Function with 3 sequential RMIs (7/7): source code

Synchronous

- Source code: p1200-sync-3rmis.cpp

Asynchronous

- Source code: p1210-async-3rmis.cpp
- Source code: p1212-async-3rmis-local-event-loop.cpp
- Using Boost ASIO (connect -> write -> read): clientserver0/client1.cpp, clientserver0/client2.cpp

Coroutines

- Source code: p1220-coroutines-3rmis.cpp
- Source code: p1222-coroutines-3rmis-generichandler.cpp

WHY USE COROUTINES?

Overview: asynchronous I/O and communication examples

Lower level: I/O

- Function with (a)synchronous wait, write or read

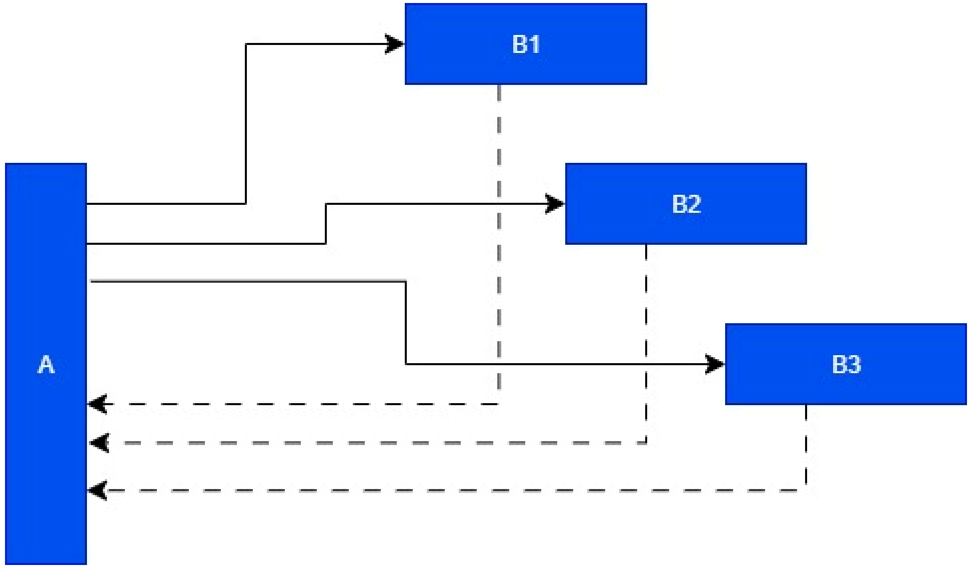
Upper level: RMI/RPC

- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 sequential RMIs
- **Function with 3 “parallel” RMIs (6 slides)**
- Function with RMI inside nested loop
- Function with RMI inside nested loop + segmentation

WHY USE COROUTINES?

Function with 3 “parallel” RMIs (1/6): real world examples

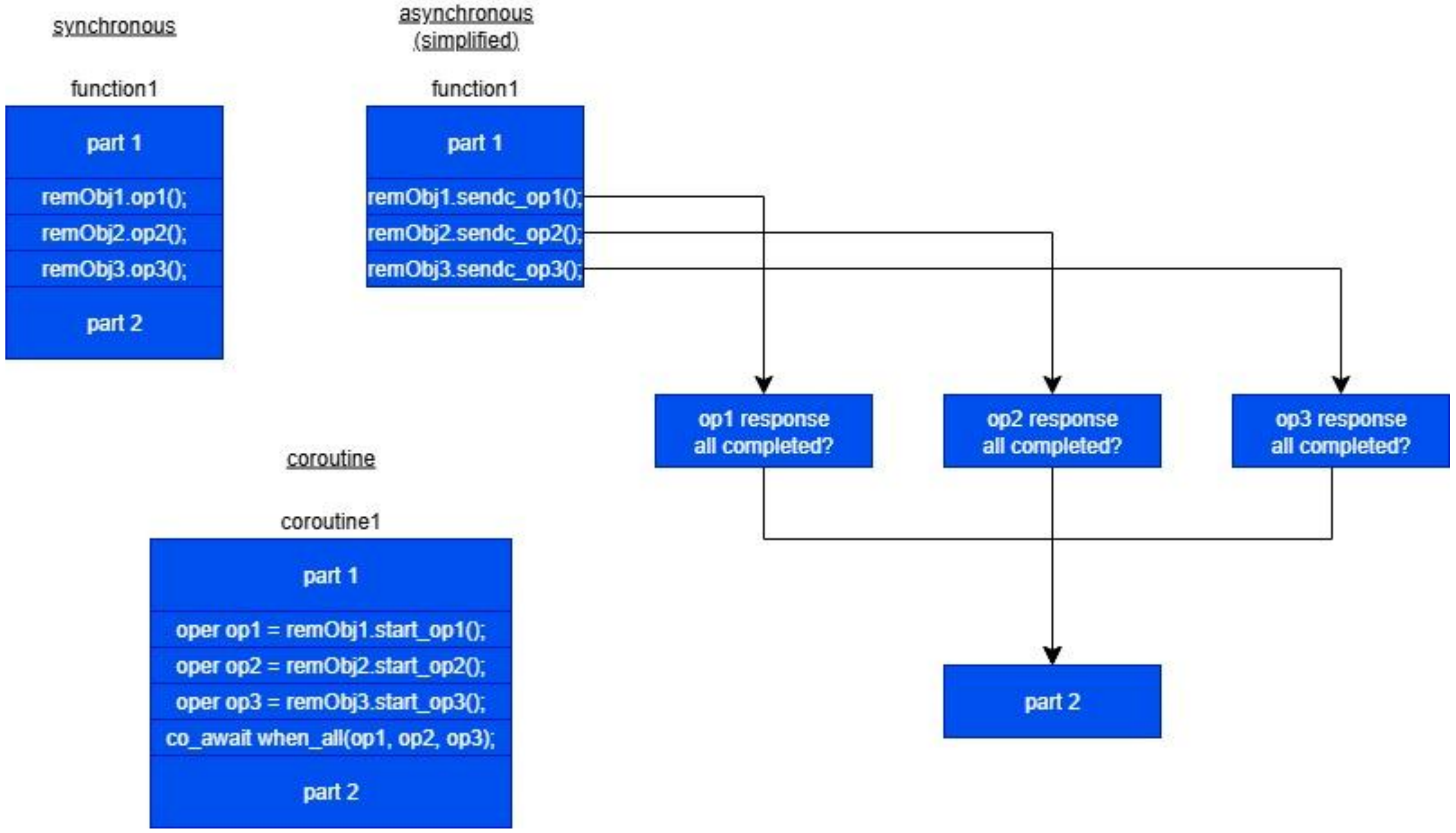
- In a message-based system, process A sends a message to N other processes B1, … requesting for information.
- Every process Bi sends a response to process A.
- Process A collects the responses and proceeds when all responses have been received.
- Processes are structured as finite state machines (FSMs)
  - In every state only a subset of the messages will/can be handled.
  - A message that is received in a “wrong” state indicates a (logical) error.
- An alternative architecture for this type of system could be publish/subscribe.



- In the “2005 system,” it was sometimes more efficient to start independent operations on multiple servers in parallel, gather the responses and then proceed, instead of calling operations in a sequential way.
- This is not possible using a synchronous style.

WHY USE COROUTINES?

Function with 3 “parallel” RMIs (2/6)



WHY USE COROUTINES?

Function with 3 “parallel” RMIs (3/6): synchronous – coroutine

```
int function1(int in1, int in2)
{
    int out11 = -1, out12 = -1;
    int out21 = -1, out22 = -1;
    int out31 = -1, out32 = -1;

    int ret1 = remoteObj1.op1(in1, in2, out11, out12
    int ret2 = remoteObj2.op1(in1, in2, out21, out22
    int ret3 = remoteObj3.op1(in1, in2, out31, out32
    int result = ret1 + ret2 + ret3;
    return result;
}
```

```
async_task<int> coroutine1(int in1, int in2)
{
    int out11 = -1, out12 = -1;
    int out21 = -1, out22 = -1;
    int out31 = -1, out32 = -1;

    async_task<int> op1 = remoteObj1co.op1(in1, in2, out11, out12);
    async_task<int> op2 = remoteObj2co.op1(in1, in2, out21, out22);
    async_task<int> op3 = remoteObj3co.op1(in1, in2, out31, out32);
    co_await when_all(op1, op2, op3);
    int result = op1.get_result() + op2.get_result() + op3.get_result();
    co_return result;
}
```

The synchronous and coroutine examples are syntactically very close to each other and have been placed side by side. The asynchronous example is more complex, see next slide.

WHY USE COROUTINES?

Function with 3 “parallel” RMIs (4/6): asynchronous style

```
struct function1_ctxt_t
{
    bool callfinished[3]{ false, false, false };
    int result[3]{ 0, 0, 0 };
};

void function1(int in1, int in2, int& ret)
{
    function1_ctxt_t* ctxt = new function1_ctxt_t;

    remoteObj1.sendc_op1(in1, in2,
        [this, ctxt, &ret](int out1, int out2, int ret1)
                          { this->function1a(ctxt, 0, out1, out2, ret1, ret); });
    remoteObj2.sendc_op1(in1, in2,
        [this, ctxt, &ret](int out1, int out2, int ret1)
                          { this->function1a(ctxt, 1, out1, out2, ret1, ret); });
    remoteObj3.sendc_op1(in1, in2,
        [this, ctxt, &ret](int out1, int out2, int ret1)
                          { this->function1a(ctxt, 2, out1, out2, ret1, ret); });
}

void function1a(function1_ctxt_t* ctxt, int index, int out1, int out2, int ret1, int& ret)
{
    ctxt->callfinished[index] = true;
    ctxt->result[index] = ret1;
    if (ctxt->callfinished[0] && ctxt->callfinished[1] && ctxt->callfinished[2]) {
        ret = ctxt->result[0] + ctxt->result[1] + ctxt->result[2];
    }
}
```

WHY USE COROUTINES?

Function with 3 “parallel” RMIs (5/6): asynchronous style (see a previous slide for a picture)

- The synchronous implementation uses 3 RMIs that can, in theory, run in parallel.
  - The synchronous implementation is inefficient because of the sequential execution.
  - Once the last RMI has returned, the function continues with part 2.
- In the asynchronous implementation, the 3 RMIs can be started one after the other (without waiting for the response).
  - The responses can arrive in any order and are handled by callback functions.
  - Each callback function 1) retrieves the out arguments and return value of the original operations and 2) checks if the other callbacks have run.
    - If no, do nothing.
    - If yes, call the part 2 function.

WHY USE COROUTINES?

Function with 3 “parallel” RMIs (6/6): source code

Synchronous

- Source code: p1500-sync-3-parallel-rmis.cpp

Asynchronous

- Source code: p1510-async-3-parallel-rmis.cpp

Coroutines

- Source code: p1520-coroutines-3-parallel-rmis.cpp

WHY USE COROUTINES?

Overview: asynchronous I/O and communication examples

Lower level: I/O

- Function with (a)synchronous wait, write or read

Upper level: RMI/RPC

- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 sequential RMIs
- Function with 3 “parallel” RMIs
- **Function with RMI inside nested loop (4 slides)**
- Function with RMI inside nested loop + segmentation

WHY USE COROUTINES?

Function with RMI inside nested loop (1/4): real world examples

Measuring the round-trip delay of messages (protocol data units) transferred onto a serial line.

- Outer loop: construct messages with lengths between a minimum length l1 and a maximum length l2.
  - Smallest message: e.g., STX – empty body – CRC – ETX.
- Inner loop: send and receive each same-length message several times and calculate the average.
- The program should not block waiting for a send/receive interaction to have completed.

Changing the frequency on a transmitter or receiver with variable capacitors from f1 to f2.

- Outer loop: change the frequency in small steps between f1 and f2.
- Inner loop: calculate and adjust position of each motor driving a variable capacitor.
- The program should not block waiting for all motors to have reached their target positions.

Discussion

- Both algorithms can run on a dedicated worker thread, with the main thread always ready to receive user input (such as “stop” or requesting the current status).
- The main thread and the worker thread must share some information.

WHY USE COROUTINES?

Function with RMI inside nested loop (2/4): synchronous - coroutine

```
void function1()
{
    for (int i = 0; i < MAX_MSG_LENGTH; i++)
    {
        Msg msg(i * 10);
        for (int j = 0; j < NR_MSGS_TO_SEND; j++)
        {
            int ret1 = remoteObj1.op1(msg);
            (void)ret1;
        }
    }
}
```

```
async_task<void> coroutine1()
{
    for (int i = 0; i < MAX_MSG_LENGTH; i++)
    {
        Msg msg(i * 10);
        for (int j = 0; j < NR_MSGS_TO_SEND; j++)
        {
            int ret1 = co_await remoteObj1co.start_op1(msg);
            (void)ret1;
        }
    }
}
```

The synchronous and coroutine examples are syntactically very close to each other and have been placed side by side. The asynchronous example is more complex, see next slide.

WHY USE COROUTINES?

Function with RMI inside nested loop (3/4): asynchronous

```
struct function1_ctxt_t
{
    Msg msg;
    int i = 0;
    int j = 0;
};

void function1()
{
    function1_ctxt_t* ctxt = new function1_ctxt_t;
    ctxt->msg = Msg(0);
    remoteObj1.sendc_op1(ctxt->msg,
        [this, ctxt]() {
            this->function1a(ctxt);
        });
}
```

```
void function1a(function1_ctxt_t* ctxt)
{
    if (ctxt->j < NR_MSGS_TO_SEND) {
        remoteObj1.sendc_op1(ctxt->msg,
                        [this, ctxt]() {
                            this->function1a(ctxt);
                        });
        ctxt->j++;
    }
    else {
        ctxt->j = 0;
        ctxt->i++;
        if (ctxt->i < MAX_MSG_LENGTH) {
            ctxt->msg = Msg(ctxt->i * 10);
            remoteObj1.sendc_op1(ctxt->msg,
                            [this, ctxt]() {
                                this->function1a(ctxt);
                            });
            ctxt->j++;
        }
        else {
            // End of inner and outer loop
            delete ctxt;
        }
    }
}
```


WHY USE COROUTINES?

Function with RMI inside nested loop (4/4): source code

Synchronous

- Source code: p1300-sync-nested-loop.cpp, p1350-sync-nested-loop.cpp

Asynchronous

- Source code: p1310-async-nested-loop.cpp, p1360-async-nested-loop.cpp
- Source code using Qt5: clientserver10/tcpclient00.cpp

Coroutines

- Source code: p1320-coroutines-nested-loop.cpp, p1370-coroutines-nested-loop.cpp

WHY USE COROUTINES?

Overview: asynchronous I/O and communication examples

Lower level: I/O

- Function with (a)synchronous wait, write or read

Upper level: RMI/RPC

- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 sequential RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- **Function with RMI inside nested loop + segmentation (2 slides)**

WHY USE COROUTINES?

Function with RMI inside nested loop + segmentation (1/2): real world example

- In the nested loop example, I “forgot” that the lowest layer can only handle messages of a certain maximum size.
- Messages larger than the MTU (maximum transfer unit) will have to be split into chunks of the maximum size and reassembled at the receiver side.
- Can we add this segmentation + reassembly in all 3 cases (synchronous, asynchronous, coroutines) without any (or with minimal) impact on the existing code?

WHY USE COROUTINES?

Function with RMI inside nested loop + segmentation (2/2): source code

Synchronous

- Source code: p1400-sync-segmentation.cpp

Asynchronous

- Source code: p1410-async-segmentation.cpp

Coroutines

- Source code: p1420-coroutines-segmentation.cpp

WHY USE COROUTINES?

Overview: other uses of coroutines

- **As an alternative to threads (7 slides, 2 examples)**
- In embedded software
- In other programming languages (Python, C#, …)
- Completion on another thread
- Local event loop
- CORBA: synchronous – callback – polling – coroutines
- Communicating finite state machines

WHY USE COROUTINES?

Coroutines as an alternative to threads (real world example 1) (1/7)

- Example: asynchronous distributed system using message communication.
- The system contains tens of processors communicating over an internal network
- Only one active/standby processor pair has access to a SCSI disk.
- Applications on other processors use an asynchronous file system interface to read/write data from/to disk.
  - The equivalent of async_open, async_read, async_write, async_close, but with messages.
- To separate the file access flow from the application's "business" flow, the file access flow runs on a dedicated thread.
- The "business flow thread" communicates with the "file access thread" using an internal queue, posting the data to be written to a file on the disk.
- Next slide: source code of a (simplified) coroutine implementation.
- Full source code: p1920-async_queue-async_file.cpp, p1922-async_queue_eq-async_file.cpp

WHY USE COROUTINES?

Coroutines as an alternative to threads (based upon real world example 1) (2/7)

```
async_task<void> filewriter()
{
    while (true) {
        int ret = -1;
        std::string str = co_await queue.pop();
        if (str.size() == 0)
            break;

        ret = co_await file.start_open("alarms.txt");
        (void)ret;

        ret = co_await file.start_write(str);
        (void)ret;

        ret = co_await file.start_close();
        (void)ret;
    }
    co_return;
}
```

```
async_task<void> producer(int nr_operations)
{
    for (int counter = 0; counter < nr_operations; counter++) {
        co_await dummy_op();

        std::string str = "This is string " + std::to_string(counter)
            + " to save";
        co_await queue.push(str);
    }
    // Signal end of producer
    co_await queue.push("");
    co_return;
}

async_task<void> mainflow(int nr_operations)
{
    int ret = co_await file.start_create("alarms.txt");
    (void)ret;

    async_task<void> fw = filewriter();
    async_task<void> pr = producer(nr_operations);
    co_await when_all(fw, pr);

    co_await file.start_remove("alarms.txt");
    co_return;
}
```


WHY USE COROUTINES?

Coroutines as an alternative to threads (example 2) (3/7)

- The code on the next slide uses two measurement loops (see above) that can run in parallel to optimize throughput.
- Instead of running them on a separate thread, the example uses coroutines instead, both for the measurement functionality and for “spawning” two or more measurements.

WHY USE COROUTINES?

Coroutines as an alternative to threads (example 2) (4/7)

```
async_task<int> TcpClient02::measurementLoop44()
{
    qDebug() << Q_FUNC_INFO << "begin";
    async_task<int> t1 = measurementLoop40(m_tcpClient1);
    async_task<int> t2 = measurementLoop40(m_tcpClient2);
    when_all wa({ &t1, &t2 });
    co_await wa;
    qDebug() << Q_FUNC_INFO << "end";
    co_return 0;
}
```

Note: this example use Qt5.

```
async_task<int> TcpClient02::measurementLoop40(TcpClientCo& tcpClient)
{
    qDebug() << Q_FUNC_INFO << "begin";
    int msgLength = 0;
    for (int selection = 0; selection < nr_message_lengths; selection++)
    {
        std::chrono::high_resolution_clock::time_point start =
            chrono::high_resolution_clock::now();
        for (int i = 0; i < configuration.m_numberTransactions; i++)
        {
            QByteArray data = prepareMessage(selection);
            msgLength = data.length();
            tcpClient.sendMessage(data);
            async_operation<QByteArray> op = tcpClient.start_reading();
            QByteArray dataOut = co_await op;

            qInfo() << dataOut.length() << ":" << dataOut;
        }
        calculateElapsedTime(start, msgLength);
    }
    qDebug() << Q_FUNC_INFO << "end";
    co_return 0;
}
```


WHY USE COROUTINES?

Coroutines as an alternative to threads (example 2) (5/7)

- Full source code: examples/clientserver11/tcpclient02.cpp
- measurementLoop44() starts async_task<int> t1 by calling measurementLoop40(m_tcpClient1).
- measurementLoop40(m_tcpClient1) runs until QByteArray dataOut = co_wait op;
- The reading operation has not yet completed: measurementLoop40() suspends and returns control to measurementLoop44()
  - To complete the operation, the event loop must run, which is not yet the case.
- Repeat the previous three steps for t2 and m_tcpClient2.
- Since t1 and t2 have not co_return-ed, measurementLoop44() suspends at the co_await line and returns control to its calling function/coroutine, etc., until we reach the event loop (not in the code fragments).
- Either of the operations will complete, which will make the corresponding measurementLoop40 coroutine run till the next co_await (next iteration in the double loop).
- This process continues until we leave the double loop and measurementLoop40() calls co_return.
- When both t1 and t2 have completed, measurementLoop44() resumes at co_await wa and calls co_return.

WHY USE COROUTINES?

Coroutines as an alternative to threads (6/7)

- Threads support pre-emptive multitasking.
  - The OS can de-schedule one thread and schedule another thread, e.g. when one thread blocks on I/O or when it has used its maximum allotted time (time slice).
- Coroutines support cooperative multitasking.
  - One coroutine must voluntarily release control (suspend) to allow another coroutine to resume (on the same thread).
- Thread synchronization:
  - Uses condition variables (CVs), (binary) semaphores, mutexes, latches, …
  - A thread that cannot proceed because it needs information (that is not available yet) acquires a semaphore or CV.
  - Another thread that supplies the information releases the semaphore or CV the first thread has acquired.
- Coroutine “synchronization:”
  - A coroutine that cannot proceed because it needs information (that is not available yet) suspends.
  - When the information becomes available, the code (completion handler) that makes the information available, resumes the coroutine. This code can run on the same thread.

WHY USE COROUTINES?

Coroutines as an alternative to threads (7/7)

- Threads can be (de)scheduled at any time.
  - Shared data must be protected using mutexes.
- Several coroutines can run on a single thread.
  - An application may only have one thread (the main thread).
- Coroutines use a disciplined way to pass control to each other.
- IMO, we should avoid that the code of a single coroutine runs on different threads.
- Notice: I am not against threads, but we should use them in a very disciplined way:
  - One variable, object, coroutine can only be accessed from a single thread.
  - Threads communicate by posting information on a queue (“mailbox”) which is read by a single thread but can be written to by all other threads. Only the mailbox needs to use thread synchronization, not the application.
  - When one thread needs the services of an object owned by another thread, it posts a functor onto the latter thread’s mailbox, which will execute the functor on behalf of the first thread.

WHY USE COROUTINES?

Overview: other uses of coroutines

- As an alternative to threads
- **In embedded software (1 slide)**
- In other programming languages (Python, C#, …)
- Completion on another thread
- Local event loop
- CORBA: synchronous – callback – polling – coroutines
- Communicating finite state machines

WHY USE COROUTINES?

Embedded software – without operating system

- Size of embedded software: a few KB till a few hundred KB (or even more).
- Single executable:
  - No traditional operating system is used or necessary to schedule multiple processes (there is only one process).
  - Often a real-time kernel (RTK) or real-time operating system (RTOS) is integrated in the application to provide threads with different priorities, e.g., to separate long-running background tasks from tasks that need immediate response.
- Embedded systems usually do a lot of (asynchronous) I/O (serial bus, USB, I²C, …)
- Coroutines can be used as an alternative to RTK/RTOS threads.
  - Use several event queues, one queue per thread priority.
  - Interrupt service routines (ISRs) create an event and place it in an event queue.
- This approach works well as long as long-running coroutines voluntarily return control by calling co_await.
  - This mechanism could be implemented using an “artificial” asynchronous operation: the operation posts its “response” in the corresponding low-priority event queue.
  - No pre-emption or time-sliced scheduling mechanism is necessary.
  - However, this approach may lead to unnecessary “yields” to allow the event loop to run.

WHY USE COROUTINES?

Overview: other applications of coroutines

- Use coroutines as an alternative to threads
- Use coroutines in embedded software
- **Coroutine use in other programming languages (Python, C#, …) (1 slide, 1 example)**
- Completion on another thread
- Local event loop
- CORBA: synchronous – callback – polling – coroutines
- Communicating finite state machines

WHY USE COROUTINES?

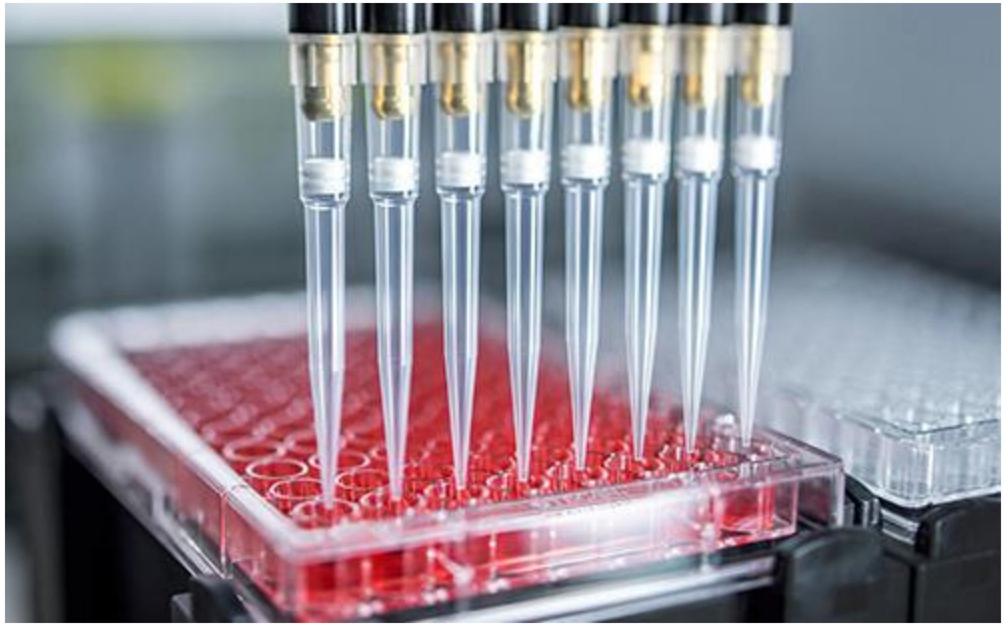
Python

```
async def liquid_handling_sequence():
    await lh.pick_up_tips(tip_rack["A1:H1"])
    await lh.aspirate(plate["A1:H1"],
                      vols=100,
                      flow_rates=100,
                      end_delay=0.5,
                      offsets=Coordinate(1, 2, 3)
                      )
    await lh.dispense(plate["A2:H2"], vols=100)
    await lh.return_tips()
    await asyncio.gather(
        liquid_handling_sequence(),
        backend.refill_pump()
    )
```

- Each of the 4 steps (tip pick up, aspirate, dispense, tip return) take a few seconds.
- While waiting for a step to complete, the async procedure does not block the rest of the application.

Liquid handler software

- PyLabRobot:
- https://www.cell.com/device/pdf/S2666-9986(23)00170-9.pdf
- https://www.biorxiv.org/content/10.1101/2023.07.10.547733v1.full.pdf
- https://github.com/PyLabRobot/pylabrobot



WHY USE COROUTINES?

Overview: other uses of coroutines

- As an alternative to threads
- In embedded software
- In other programming languages (Python, C#, …)
- **Completion on another thread (8 slides, 3 examples)**
- Local event loop
- CORBA: synchronous – callback – polling – coroutines
- Communicating finite state machines

WHY USE COROUTINES?

Completion on another thread (1/8): example without coroutines

```
class RemoteObject1Thr
{
public:
    void op1(int in1, int in2, int& out1, int& out2, int& ret1, std::latch& latch_)
    {
        m_remoteObject.startthr_op1(in1, in2,
            [this, &latch_, &out1, &out2, &ret1](int out1a, int out2a, int ret1a)
            {
                out1 = out1a;
                out2 = out2a;
                ret1 = ret1a;
                latch_.count_down();
            });
    }
};

int function1(int in1, int in2)
{
    std::latch mainLatch{ 3 };
    remoteObj1thr.op1(in1, in2, out11, out12, ret1, mainLatch);
    remoteObj1thr.op1(in1, in2, out21, out22, ret2, mainLatch);
    remoteObj1thr.op1(in1, in2, out31, out32, ret3, mainLatch);
    mainLatch.wait();
    int result = ret1 + ret2 + ret3;
    return result;
}
```

- The completion handler (lambda) runs on a dedicated thread and notifies the main thread using a C++20 std::latch.
- The programming style is sequential, although we use an asynchronous API.
- However, function1 blocks the main thread until all 3 responses have arrived.

WHY USE COROUTINES?

Completion on another thread (2/8): example with coroutines

```
class RemoteObject1Co : public CommService
{
public:
    async_task<int> op1(int in1, int in2, int& out1, int& out2)
    {
        async_operation<op1_ret_t> op1 = start_op1(in1, in2);
        op1_ret_t res = co_await op1;
        out1 = res.out1;
        out2 = res.out2;
        co_return res.ret;
    }
};


async_task<int> coroutine1(int in1, int in2)
{
    printf("Class01::coroutine1()\n");
    int out11 = -1, out12 = -1;
    int out21 = -1, out22 = -1;
    int out31 = -1, out32 = -1;
    async_task<int> op1 = remoteObj1co.op1(in1, in2, out11, out12);
    async_task<int> op2 = remoteObj2co.op1(in1, in2, out21, out22);
    async_task<int> op3 = remoteObj3co.op1(in1, in2, out31, out32);
    when_all wa(op1, op2, op3);
    co_await wa;
    int result = op1.get_result() + op2.get_result() + op3.get_result();
    co_return result;
}
```

- Blocking problem has been solved using coroutines.
- There is no need for completion on another thread.

WHY USE COROUTINES?

Completion on another thread (3/8): gRPC example 1: original example without coroutines

```
std::string SayHello(const std::string& user) {
  HelloRequest request;
  request.set_name(user);
  HelloReply reply;
  ClientContext context;
  std::mutex mu;
  std::condition_variable cv;
  bool done = false;
  Status status;
  print(PRI1, "SayHello: pre\n");
  stub_->async()->SayHello(&context, &request, &reply,
                           [&mu, &cv, &done, &status](Status s) {
                             status = std::move(s);
                             std::lock_guard<std::mutex> lock(mu);
                             done = true;
                             cv.notify_one();// looks like handle.resume()
                           });

  std::unique_lock<std::mutex> lock(mu);
  while (!done) {
    cv.wait(lock);           // looks like a co_await
  }
  if (status.ok()) {
    return reply.message();
  } else {
    return "RPC failed";
  }
}
```

- The completion handler (lambda) runs on a dedicated thread and notifies the main thread using a condition variable.
- The programming style is sequential, although we use an asynchronous API.
- The thread on which SayHello runs is blocked until the response has arrived.

WHY USE COROUTINES?

Completion on another thread (4/8): gRPC example 1: example with coroutines

```
async_operation<Status> start_SayHello(ClientContext* pcontext,
                                        HelloRequest& request,
                                        HelloReply& reply) {
  int index = get_free_index();
  async_operation<Status> ret{ this, index };
  stub_->async()->SayHello(pcontext, &request, &reply,
      [index, this](Status s) {
          Status status = std::move(s);
          completionHandler<Status>(index, status);
      });
  return ret;
}

async_task<std::string> SayHelloCo(const std::string& user) {
  HelloRequest request;
  request.set_name(user);
  HelloReply reply;
  ClientContext context;
  Status status;
  status = co_await start_SayHello(&context, request, reply);
  if (status.ok()) {
      co_return reply.message();
  }
  else {
      co_return "RPC failed";
  }
}
```


WHY USE COROUTINES?

Completion on another thread (5/8): gRPC example 2: original example without coroutines

```
// Request to a Greeter service
hello_request.set_name("user");
helloworld::Greeter::NewStub(channel_)->async()->SayHello(
    &hello_context, &hello_request, &hello_response,
    [&](Status status) {
        std::lock_guard<std::mutex> lock(mu);
        done_count++;
        hello_status = std::move(status);
        cv.notify_all();             // looks like handle.resume()
    });
// Request to a RouteGuide service
feature_request.set_latitude(50);
feature_request.set_longitude(100);
routeguide::RouteGuide::NewStub(channel_)->async()->GetFeature(
    &feature_context, &feature_request, &feature_response,
    [&](Status status) {
        std::lock_guard<std::mutex> lock(mu);
        done_count++;
        feature_status = std::move(status);
        cv.notify_all();             // looks like handle.resume()
    });
// Wait for both requests to finish
cv.wait(lock, [&]() { return done_count == 2; });    // looks like co_await when_all
if (hello_status.ok()) {
    // ...
}
if (feature_status.ok()) {
   // ...
}
```

Starting two asynchronous operations one after the other.

Waiting until both have finished.

WHY USE COROUTINES?

Completion on another thread (6/8): gRPC example 2: example with coroutines

```
async_operation<Status> start_GetFeature(ClientContext* pcontext, routeguide::Point& request,
                                         routeguide::Feature& reply) {
    int index = get_free_index();
    async_operation<Status> ret{ this, index };
    routeguide::RouteGuide::NewStub(channel_)->async()->GetFeature(pcontext, &request, &reply,
        [index, this](Status s) {
            Status status = std::move(s);
            completionHandler<Status>(index, status);
        });
    return ret;
}
async_task<std::string> GetFeatureCo() {
    // Code removed to save some space
    Status feature_status =
        co_await start_GetFeature(&feature_context, feature_request, feature_response);
    std::stringstream strstr;
    // Code removed to save some space
    co_return strstr.str();
}
async_task<void> SayHello_GetFeatureCo_when_all() {
    async_task<std::string> t1 = SayHelloCo();
    async_task<std::string> t2 = GetFeatureCo();
    co_await when_all(t1, t2);
    std::cout << t1.get_result();
    std::cout << t2.get_result();
    co_return;
}
```


WHY USE COROUTINES?

Completion on another thread (7/8): comparison and conclusions

- The original examples and the coroutine examples both have a sequential programming style.
- In the non-coroutine examples, the function that calls cv.wait(…) (or an equivalent synchronization primitive) is (usually) blocked and blocks (recursively) its calling function(s).
  - This is not the case with coroutines: when a coroutine suspends, it passes control to its caller / resumer.
- A solution is to run the top-level function on a dedicated thread (e.g., one thread for every received event).
- Comparison (see table below):

Action	Using threads	Using coroutines
waiting	cv.wait(lock);	co_await …;
continuation	cv.notify_one();	handle.resume();

WHY USE COROUTINES?

Completion on another thread (8/8): source code

Example:

- Without coroutines: p1200thr.h, p1515-async+thread-3-parallel-rmis.cpp
- With coroutines: p1200co.h, p1520-coroutines-3-parallel-rmis.cpp

gRPC example 1:

- Original example without coroutines: greeter_callback_client.cc
- With coroutines: greeter_cb_coroutine_client2.cc

gRPC example 2:

- Original example without coroutines: multiplex_client2.cc
- With coroutines: multiplex_coroutine_client3-when_all.cc

WHY USE COROUTINES?

Overview: other uses of coroutines

- As an alternative to threads (2 examples)
- In embedded software
- In other programming languages (Python, C#, …)
- Completion on another thread
- **Local event loop (3 slides)**
- CORBA: synchronous – callback – polling – coroutines
- Communicating finite state machines

WHY USE COROUTINES?

Local event loop (1/3): without coroutines

```
void function1(int in1, int in2, int testval, int& lret)
{
    int lret1 = -1;
    remoteObj1.sendc_op1(in1, in2,
        [this, &lret1](int out1, int out2, int ret1) {
            lret1 = this->callback1(out1, out2, ret1);
        });
    // 1a Do some stuff that doesn't need the result of the RMI
    eventQueue.run();
    // 1b Do stuff that needs the result of the RMI
    if (lret1 == testval) {
        remoteObj2.sendc_op2(in1, in2,
            [this, &lret](int out1, int ret1) {
                lret = this->callback2(out1, ret1);
            });
        // 2a Do some stuff that doesn't need the result of the RMI
        eventQueue.run();
        // 2b Do stuff that needs the result of the RMI
    }
    else {
        remoteObj3.sendc_op3(in1,
            [this, &lret](int out1, int out2, int ret1) {
                lret = this->callback3(out1, out2, ret1);
            });
        // 3a Do some stuff that doesn't need the result of the RMI
        eventQueue.run();
        // 3b Do stuff that needs the result of the RMI
    }
}
```

- The completion handler (lambda) runs on the same thread.
- The programming style is sequential, although we use an asynchronous API.
- The thread on which function1 runs is blocked inside eventQueue.run() until the response has arrived.

WHY USE COROUTINES?

Local event loop (2/3): with coroutines

```
async_task<int> coroutine1a(int in1, int in2, int testval)
{
    int out1 = -1, out2 = -1;
    async_task<int> op1 = remoteObj1co.op1(in1, in2, out1, out2);
    // 1a Do some stuff that doesn't need the result of the RMI
    int ret1 = co_await op1;
    // 1b Do stuff that needs the result of the RMI
    if (ret1 == testval) {
        int out3 = -1;
        async_task<int> op2 = remoteObj2co.op2(in1, in2, out3);
        // 2a Do some stuff that doesn't need the result of the RMI
        int ret2 = co_await op2;
        // 2b Do stuff that needs the result of the RMI
        co_return ret2;
    }
    else {
        int out4 = -1, out5 = -1;
        async_task<int> op3 = remoteObj3co.op3(in1, out4, out5);
        // 3a Do some stuff that doesn't need the result of the RMI
        int ret3 = co_await op3;
        // 3b Do stuff that needs the result of the RMI
        co_return ret3;
    }
}
```

WHY USE COROUTINES?

Local event loop (3/3): discussion and source code

- Both pieces of code use an asynchronous API and have a very similar structure, but the control flow is different.
- The code not using coroutines blocks the function and its calling functions, while the coroutine passes control to its caller (or resumer) and enters eventually a global event loop.
- The non-coroutine code is used in CORBA AMI (Common Object Request Broker Architecture - Asynchronous Method Invocation), callback variant.
  - The polling variant is similar but hides the local event loop in the polling function.
- The local event loop should only accept the response corresponding to the sent request.
  - It should defer all other events.
  - These events will be handled by the global event loop (that will be entered after leaving the function).
- Source code: without coroutines: p1212-async-3rmis-local-event-loop.cpp
- Source code: with coroutines: p1220-coroutines-3rmis.cpp, p1222-coroutines-3rmis-generichandler.cpp

WHY USE COROUTINES?

Completion on another thread versus local event loop versus coroutines

	Completion on another thread	Local event loop	Coroutines
Asynchronous API used?	Yes	Yes	Yes
Synchronous / sequential programming style?	Yes	Yes	Yes
Completion handler runs on	A dedicated thread	Same thread	Dedicated or same thread
“await” realized by	Thread synchronization mechanisms	Local event loop	co_await
Awaiting function blocked?	Yes	Yes	No
Real-world use case	gRPC	CORBA AMI	All

WHY USE COROUTINES?

- As an alternative to threads
- In embedded software
- In other programming languages (Python, C#, …)
- Completion on another thread
- Local event loop
- **CORBA: synchronous – callback – polling – coroutines (2 slides)**
- Communicating finite state machines

WHY USE COROUTINES?

CORBA: synchronous – callback – polling – coroutines (1/2)

```
void synchronous1()
{
    CORBA::Long in1 = 1;
    CORBA::Double inout1 = 1;
    CORBA::Short out1 = 0;
    CORBA::Short ret1;

    ret1 = interfaceAObj.operation1(in1, inout1, out1);
    printf("ret1 = %d, inout1 = %f, out1 = %d\n", ret1, inout1, out1);
}
```

```
async_task<void> coroutine1()
{
    CORBA::Long in1 = 1;
    CORBA::Double inout1 = 1;

    async_operation<operation1_result> a =
        interfaceACoObj.start_operation1(in1, inout1);
    operation1_result result = co_await a;
    printf("ret1 = %d, inout1 = %f, out1 = %d\n",
        result.m_ret_val, result.m_inout_val, result.m_out_val);
    co_return;
}
```

```
void asynchronous_callback1()
{
    CORBA::Long in1 = 1;
    CORBA::Double inout1 = 1;
    CORBA::Short out1 = 0;
    CORBA::Short ret1 = 0;

    interfaceAHandler_impl_ptr handler = new interfaceAHandler_impl;
    interfaceAObj.sendc_operation1(handler, in1, inout1);
    eventqueue.run();
    if (handler->finished()) {
        handler->operation1(ret1, inout1, out1);
        printf("ret1 = %d, inout1 = %f, out1 = %d\n",
                ret1, inout1, out1);
    }
    delete handler;
}
```

```
void asynchronous_polling1()
{
    CORBA::Long in1 = 1;
    CORBA::Double inout1 = 1;
    CORBA::Short out1 = 0;
    CORBA::Short ret1 = 0;

    PollerID pollerId;
    CORBA::Boolean completed = false;
    CORBA::Boolean blocking = false;
    pollerId = interfaceAObj.sendp_operation1(in1, inout1);
    while (!completed)
        completed = interfaceAObj.operation1Poller(
            pollerId, blocking, ret1, inout1, out1);
    printf("ret1 = %d, inout1 = %f, out1 = %d\n", ret1, inout1, out1);
}
```


WHY USE COROUTINES?

CORBA: synchronous – callback – polling – coroutines (2/2)

- CORBA: Common Object Request Broker Architecture
- Source code: corba_client_app.cpp
- Also contains examples with operations placed in a loop (or two loops) and the use of when_all and when_any.

WHY USE COROUTINES?

Overview: other uses of coroutines

- As an alternative to threads
- In embedded software
- In other programming languages (Python, C#, …)
- Completion on another thread
- Local event loop
- CORBA: synchronous – callback – polling – coroutines
- **Communicating finite state machines (5 slides)**

WHY USE COROUTINES?

Communicating finite state machines (1/5)

- Real-world application: communication protocols with two (or more) parties.

Original system

- Consider a system of processes communicating using message passing.
- A process is organized as a state machine using a double selection:
  - Outer selection on state
  - Inner selection on message (id)
  - Or vice-versa, or both if the programming language allows this or you can concatenate state and message id.
- A process uses a potentially infinite loop.
- At the “top” of the loop, it waits on incoming messages and then enters the state machine, depending on the state and the message id.
- The state machine does not block/wait on the response on messages it has sent: the response message will be handled by the event loop.
- See below for code fragment.

WHY USE COROUTINES?

Communicating finite state machines (2/5)

Compiling C++ coroutines to pre-C++20 code

- A C++ computer typically compiles each coroutine as a state machine.
- At the first coroutine call, the code enters the compiled coroutine at the top.
- Each suspend/resume point corresponds to a different state (id).
- On re-entry of the coroutine (due to a resume), the coroutine jumps to the resume point indicated by the current state.

Coroutine alternative to CFSMs

- Each process can be rewritten using coroutines using a single “global” event loop.
- A new request leads to calling a dedicated coroutine.
  - In the original code, the same state machine is/can be used. Coroutines will have to “communicate” using variables.
- Sending a request from that coroutine corresponds to invoking an asynchronous operation.
- Waiting for the response message to that request corresponds to co_await-ing the response message.
- Subsequent requests/responses can be handled by the same coroutine.

WHY USE COROUTINES?

Finite state machines

```
bool process_message()
{
    MessageId msgid1 = m_message_queue1.get().value_or(MessageId::NullMsg);
    if (msgid1 == MessageId::NullMsg)
        return false;
    switch (m_state) {
    case State::State1:
        switch (msgid1) {
        case MessageId::Message001_req:
            m_message_queue2.push(MessageId::Message101_req);
            m_state = State::State2;
            break;
        default:
            break;
        }
        break;
    case State::State2:
        switch (msgid1) {
        case MessageId::Message101_resp:
            m_message_queue2.push(MessageId::Message102_req);
            m_state = State::State3;
            break;
        default:
            break;
        }
        break;
    case State::State3:
        // Some lines deleted to save some space
        break;
    default:
        break;
    }
    return true;
}
```

Corolib coroutine

```
async_task<void> mainflow()
{
    co_await start_operation1();
    co_await start_operation2(4);
    co_return;
}

bool process_message(async_task<void>& t1)
{
    MessageId msgid1 = m_message_queue1.get().value_or(MessageId::Nul
    if (msgid1 == MessageId::NullMsg)
        return false;

    if (msgid1 == MessageId::Message001_req) {
        t1 = mainflow();
    }
    else
        m_messagemapper.find(msgid1);
    return true;
}
```


WHY USE COROUTINES?

Dynamic C: state machine

```
task1state = 1;                    // initialization:
while(1){
   switch(task1state){

      case 1:
         if( buttonpushed() ){
            task1state=2;   turnondevice1();
            timer1 = time;          // time incremented every second
         }
         break;

      case 2:
         if( (time-timer1) >= 60L){
            task1state=3;   turnondevice2();
            timer2=time;
         }
         break;

      case 3:
         if( (time-timer2) >= 60L){
            task1state=1;   turnoffdevice1();
            turnoffdevice2();
         }
         break;
   }
   /* other tasks or state machines */
}
```

Dynamic C: cofunctions

```
while(1){
   costate{ ... }                          // task 1

   costate{                                // task 2
      waitfor( buttonpushed() );
      turnondevice1();
      waitfor( DelaySec(60L) );
      turnondevice2();
      waitfor( DelaySec(60L) );
      turnoffdevice1();
      turnoffdevice2();
   }

   costate{ ... }                          // task n
}
```

Source:
https://hub.digi.com/dp/path=/support/asset/dynamic-c-9-users-manual-rabbit-2000-and-3000-microprocessors/

WHY USE COROUTINES?

Communicating finite state machines (5/5)

Source code:

- FSM (no coroutines): p1000-cfsm1.h
- With coroutines: p1010-cfsm1-co.h, p1012-cfsm1-co.h, p1014-cfsm1-co.h

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Why use coroutines for asynchronous (distributed) programming?
4. **When not to use coroutines? (3 slides)**
    1. When all you need is a function
    2. In simple applications where blocking is not harmful
    3. When there is no asynchronous I/O API
    4. In batch processing applications
    5. In long running algorithms with little user interaction
    6. In publish/subscribe applications
5. Summary and conclusions
6. Appendix: brief introduction to corolib

WHEN NOT TO USE COROUTINES?

Examples 1 + 2

- Introduction: the C++ compiler generates a lot of code for coroutines (compared to ordinary functions).
  - Likewise, you should not use virtual functions if you can do with non-virtual functions.
  - Virtual functions have 3 to 4 indirections (pointer to object, location of vptr (if not at the beginning of the object), pointer to virtual function table, pointer to function).

- If you just need a function.
  - The leaf coroutines in a call tree must co_await an awaitable object instead of just calling co_return.
  - And the awaitable object cannot provide the response immediately.
  - In the case of just co_return, the coroutine will behave as a function: no suspend-resume involves (when using eager start coroutines).
- In simple applications that have nothing else to do than to wait for a response.
  - If no response arrives (in time), <CTRL>+C is an acceptable option.

WHEN NOT TO USE COROUTINES?

Examples 3 + 4

- When there is no asynchronous I/O API
  - Run synchronous operations on a dedicated thread to avoid blocking the main thread.
  - In an application I developed in 1999, there were 18 threads (19 when run as a Windows service).
  - Most threads were controlling equipment connected to a serial bus (RS485 / RS422): one thread per bus.
  - Communication with the equipment used synchronous writes and reads (the application was the “master”).
  - Although it would be relatively easy to replace all threads by coroutines, the first step would be to replace the synchronous calls by asynchronous calls. This first step would probably take most of the development time.
- In batch processing jobs
  - Read information from an input file, process that information, write result to one or more output files.
  - Repeat this process for other input files.

WHEN NOT TO USE COROUTINES?

Examples 5 + 6

- In long running algorithms with little user interaction
  - Consider a calibration or tuning algorithm that runs for many seconds (even tens of seconds).
  - While this algorithm is running, there is little that a user can do, apart from querying the state or stopping the algorithm.
  - The algorithm can run on a separate thread and use only synchronous I/O.
  - The user thread can accept user input at any time and communicate with the auxiliary thread using a simple queue or shared variables.
- Publish/subscribe applications
  - Publishers publish information to a topic.
  - Subscribers subscribe to a topic and consume the published information.
  - A publisher does not know its subscribers (if any), a subscriber does not know its publishers (if any).
  - Produce and consume times are or may be unrelated.
  - However, if also RPC is supported (built on top of the pub/sub infrastructure), then coroutines come into scope again.
    - Example: ROS actions.

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous programming
3. Why use coroutines for asynchronous programming?
4. When not to use coroutines?
5. **Summary and conclusions (4 slides)**
6. Appendix: brief introduction to corolib

SUMMARY AND CONCLUSIONS

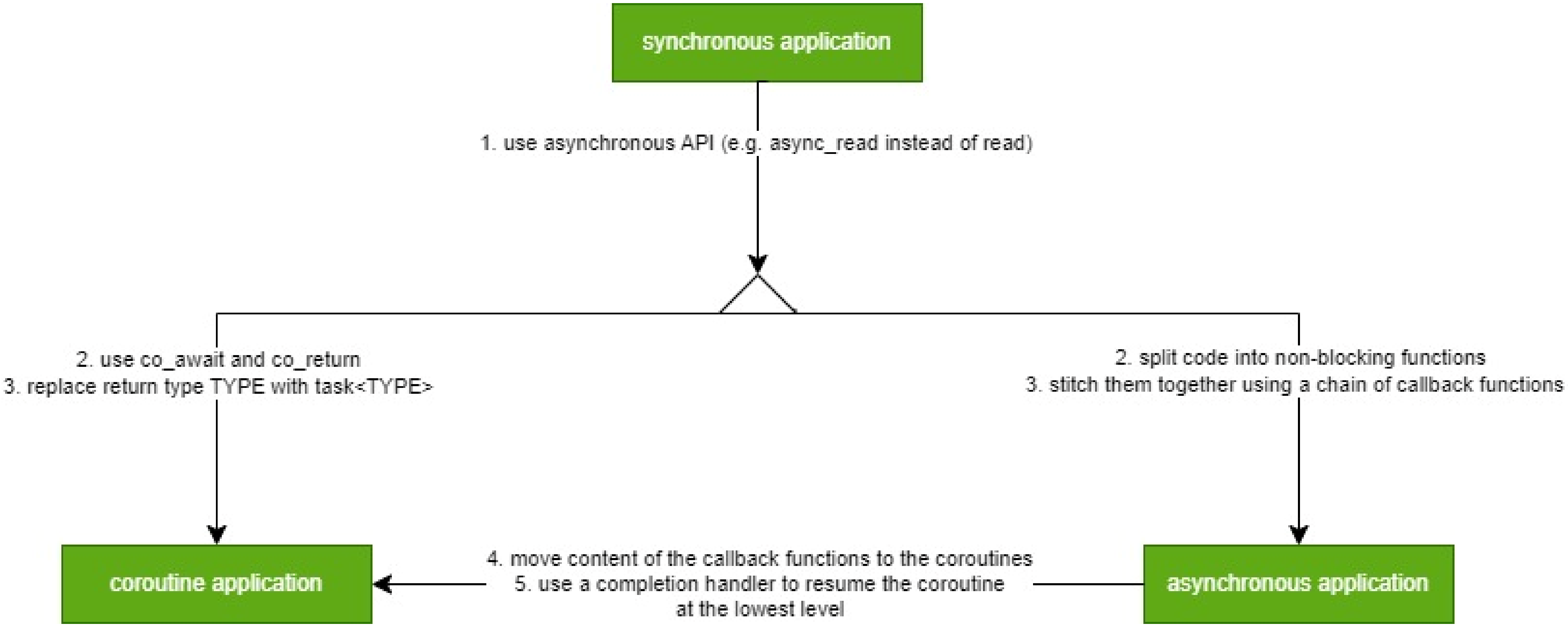
Comparison

- We can use coroutines to make a synchronous-style / single-threaded program behave like an asynchronous / multi-threaded program without making structural modifications to the original program!
- The table below compares 4 styles in the absence of coroutines.
- Coroutines combine the advantages (+) without the disadvantages (-)

	single-threaded	**multi-threaded**
synchronous	(+) Very easy to develop and maintain (-) Not reactive	(+) Easy to develop and maintain (depends on the number of threads and inter-thread communication) (+) Reactive (-) Thread (communication) overhead (-) Shared variables, …
asynchronous	(-) More difficult to develop and maintain (+) Reactive	(-) More difficult to develop and maintain (+) Reactive (-) There is no need to combine both mechanisms

SUMMARY AND CONCLUSIONS

Transformation from synchronous to asynchronous and to coroutines



SUMMARY AND CONCLUSIONS

Coroutines can be used …

- to write distributed applications using a synchronous style yet executing/behaving in an efficient reactive asynchronous way
  - Coroutines offer the advantages of both styles while not introducing any new major disadvantages.
- as an alternative to threads
  - Several coroutines can run in an interleaved/cooperative way on a single operating system thread.
  - Coroutines can avoid or reduce the use of threads.
- for lazily computed sequences (generators)
  - Not discussed in this presentation.

SUMMARY AND CONCLUSIONS

Conclusions

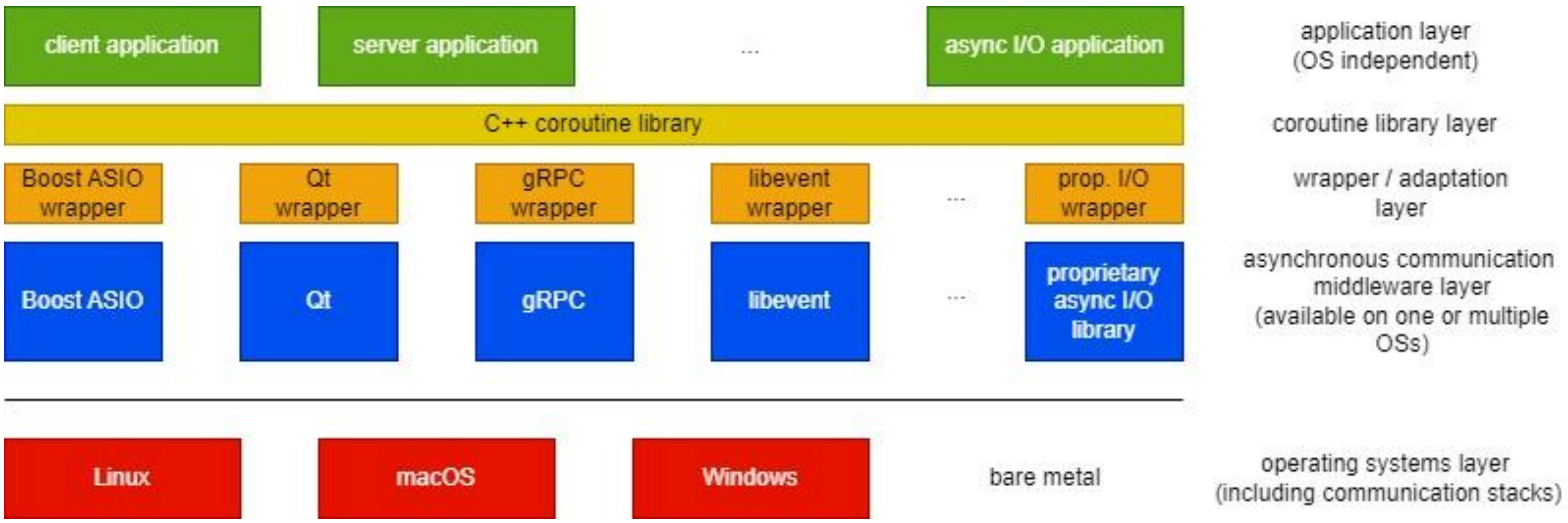
- Coroutines are very useful to write distributed (and many other types of) applications.
  - There is a minimal impact on the original algorithms/specifications: there is no need to cut the algorithm into a chain of callback functions.
- Coroutines can lead to a more uniform coding style among different types of applications:
  - Stand-alone applications performing long-running/complex algorithms
  - Distributed applications with communicating processes
  - Reactive real-time and embedded applications

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous programming
3. Why use coroutines for asynchronous programming?
4. When can you avoid using coroutines?
5. Summary and conclusions
6. **Appendix: brief introduction to corolib (4 slides)**

APPENDIX: BRIEF INTRODUCTION TO COROLIB

Organization of corolib (https://github.com/JohanVanslembrouck/corolib)



APPENDIX: BRIEF INTRODUCTION TO COROLIB

Directory structure (1/2)

```
corolib
├── docs
├── examples
│   ├── boost (uses Boost ASIO, but not the Boost coroutine implementation)
│   ├── cppcoro (integration of the coroutine library of Lewis Baker)
│   ├── curl
│   ├── grpc
│   ├── libevent
│   ├── qt5
│   ├── ros2_ws
│   ├── tao
│   └── tutorial (does not use any communication framework)
```

APPENDIX: BRIEF INTRODUCTION TO COROLIB

Directory structure (2/2)

```
├── include
│   └── corolib (header files of the coroutine library)
├── lib (.cpp files for the .h files in include/corolib)
├── studies
│   ├── cfsms (contains source code used in this presentation)
│   ├── corba
│   ├── corolab (contains the (somewhat modified) code of the 2020 presentation)
│   ├── final_suspend
│   ├── initial_suspend
│   ├── rvo
│   ├── transform
│   ├── why-coroutines (contains source code used in this presentation)
│   └── why-coroutines2 (contains source code used in this presentation)
└── tests (uses GoogleTest)
```

APPENDIX: BRIEF INTRODUCTION TO COROLIB

Notes

- Hobby project
  - Developed in my free time or between client projects.
  - First commit to GitHub in June 2020.
- Two main classes:
  - async_task (eager start) or async_ltask (lazy start)
  - async_operation
- corolib classes have been tested in numerous examples (with and without communication frameworks)
- Current status and future work:
  - Some data members of the main classes are not used or needed in all contexts.
  - A split into smaller classes could be beneficial to improve performance: only use data and code you need in any given context.

Thank you!

