Capgemini engineering

COROLIB: DISTRIBUTED PROGRAMMING WITH C++ COROUTINES

Johan Vanslembrouck (johan.vanslembrouck@capgemini.com)

AGENDA

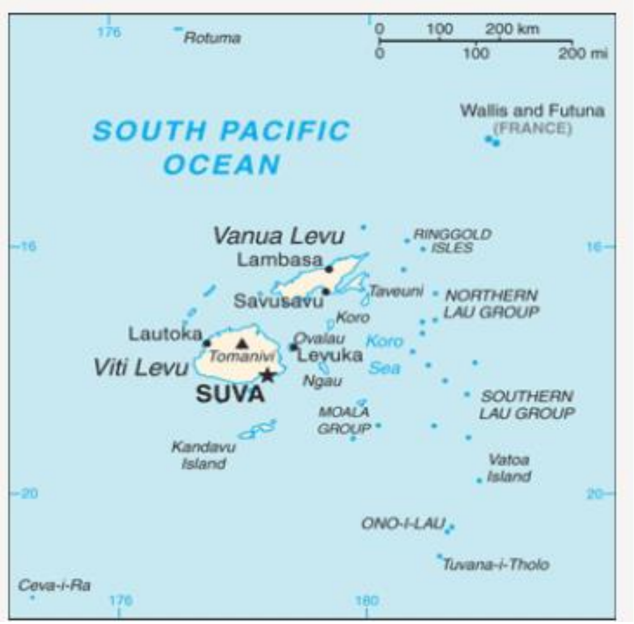
1. What is corolib? (2 slides)
2. Brief introduction to C++ coroutines (4 slides)
3. Brief introduction to (a)synchronous distributed programming (4 slides)
4. **Why use coroutines for distributed programming? (7 overview slides + 20 content slides)**
5. Corolib goals and coding style (2 slides)
6. **Corolib organization (13 slides)**
7. Corolib examples (5 slides)
8. Related work (4 slides)
9. Finalizing (1 slide)

AGENDA

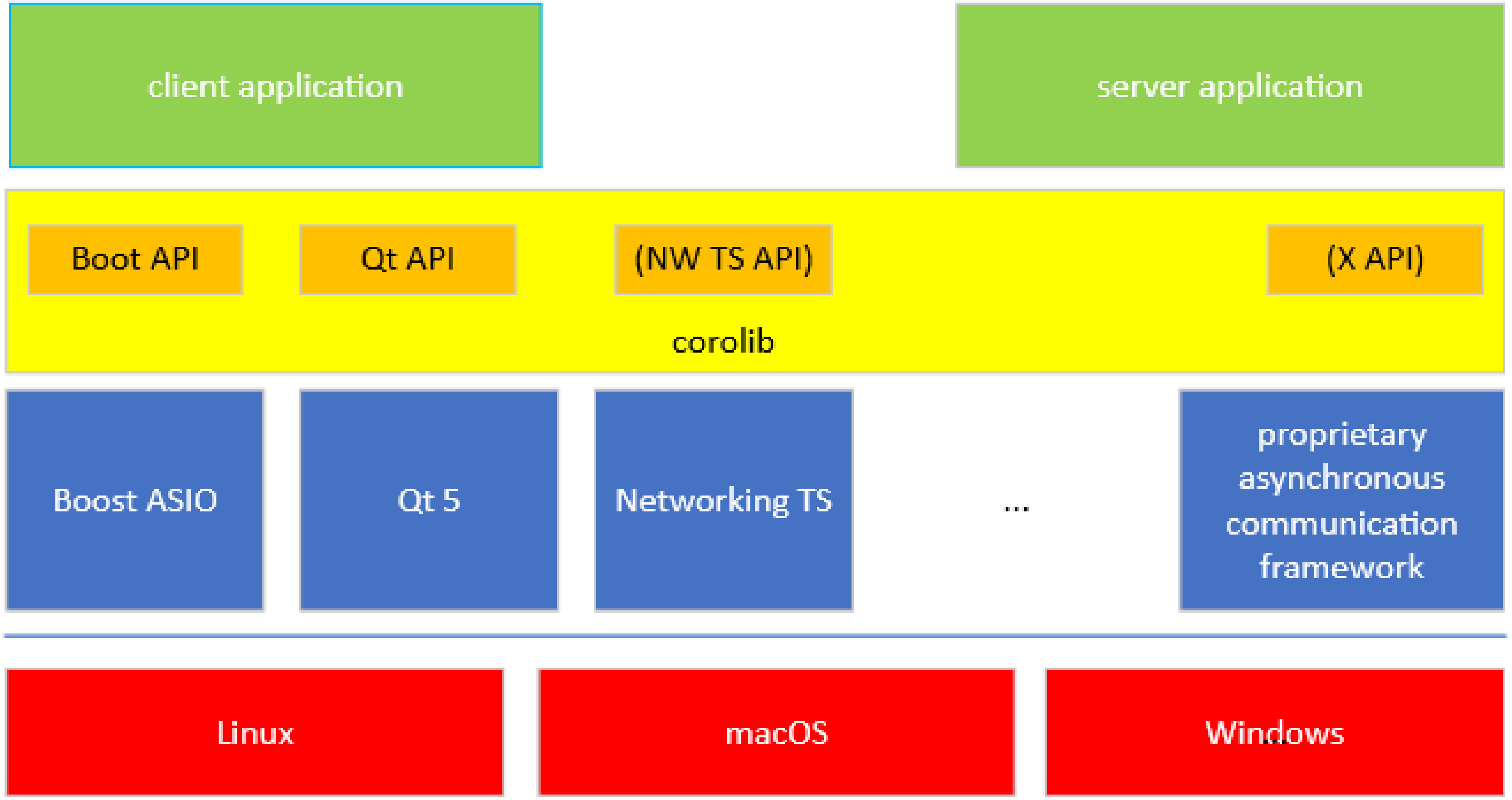
1. **What is corolib?**
2. Brief introduction to C++ coroutines
3. Brief introduction to (a)synchronous distributed programming
4. Why use coroutines for distributed programming?
5. Corolib goals and coding style
6. Corolib organization
7. Corolib examples
8. Related work
9. Finalizing

WHAT IS COROLIB?

- Corolib is an island in Northern Fiji (Oceania) with the region font code of Americas/Western Europe. It is located at an elevation of 20 meters above sea level. Corolib is also known as Goat Islet, Korolevu Island, Korolib.
  - Source: https://www.getamap.net/maps/fiji/northern/_corolib/
- Corolib is a repository on GitHub
  - A C++ coroutine library for developing asynchronous distributed applications
  - Short for “coroutine library”
  - Source: https://github.com/JohanVanslembrouck/corolib
  - Contains all examples and additional documentation for this presentation
  - One-person outside-working-hours hobby project
  - One of my objectives for 2022 for Gapgemini Engineering
- Coro-Lib is another repository on GitHub
  - A repository for deep learning and numerical modelling resources for COVID-19 mitigation
  - Source: https://github.com/anuradhakar49/Coro-Lib



WHAT IS COROLIB?



application layer
(OS independent)

corolib layer
(small coroutine layer on top of asynchronous communcation middleware)

asynchronous communication middleware layer
(available on one or multiple operating sytems)

operating system layer
(including various communication stacks)

AGENDA

1. What is corolib?
2. **Brief introduction to C++ coroutines**
3. Brief introduction to (a)synchronous distributed programming
4. Why use coroutines for distributed programming?
5. Corolib goals and coding style
6. Corolib organization
7. Corolib examples
8. Related work
9. Finalizing

BRIEF INTRODUCTION TO C++ COROUTINES

What is a coroutine?

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

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

- a **co_return** statement: returns from a coroutine (just **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 range-based for loop that uses **co_await**
  - **for co_await (for-range-declaration : expression)**

A coroutine must return an object of a coroutine type

- Cannot return just an int, void, double, etc.

BRIEF INTRODUCTION TO C++ COROUTINES

Definitions

- Stackless coroutine: A coroutine whose state includes variables and temporaries with automatic storage duration in the body of the coroutine and **does not** include the call stack
  - C++ coroutines are stackless
- Stackful coroutine / fiber / user-mode thread: A stackful coroutine state includes the full call stack associated with its execution, enabling suspension from nested stack frames

- Suspend/resume point: A point at which execution of a coroutine can be suspended with a possibility to be resumed at a later time
- Initial suspend/resume point: A suspend/resume point that occurs prior to executing the user-authored body of the coroutine
- Final suspend/resume point: A suspend/resume point that occurs after executing the user-authored body of the coroutine, but before the coroutine state is destroyed

BRIEF INTRODUCTION TO C++ COROUTINES

More definitions

- Awaitable type: type that supports the co_await operator
- Awaiter type: type that implements the three special methods that are called as part of a co_await expression: await_ready(), await_suspend() and await_resume()

- Coroutine state / coroutine frame: a state that is created when a coroutine is first invoked and destroyed once coroutine execution completes
- Coroutine object / coroutine handle / return object of the coroutine: an object returned from the initial invocation of a coroutine
- Coroutine promise: contains library-specific data required for the implementation of a higher-level abstraction exposed by a coroutine

- Generator: a coroutine that provides a sequence of values

BRIEF INTRODUCTION TO C++ COROUTINES

Comparing the behavior of 4 “program entity” types (function, coroutine, thread, process)

question	function	coroutine	thread	process
Separation of memory between entities?	no	no	no	yes
When 1 entity internally waits for input, can other entities proceed?	no	no (coroutine waiting for user input prevents other coroutines from running)	yes	yes
1 stack per entity?	no	no (stackless coroutines) yes (stackful coroutines)	yes	yes
On 1 core: can > 1 entity proceed?	no (only the one on the top of the stack can)	yes (any suspended coroutine can be resumed) Cooperative multi-tasking (hint: see occurrences of when_all and when_any)	yes Preemptive multi-tasking	yes (requires OS)
Local storage available?	static function variables	promise type + compiler generated	thread local storage (via thread control block)	not applicable
Native support in C++?	yes	yes, since C++20	yes, since C++11	yes

AGENDA

1. What is corolib?
2. Brief introduction to C++ coroutines
3. **Brief introduction to (a)synchronous distributed programming**
4. Why use coroutines for distributed programming?
5. Corolib goals and coding style
6. Corolib organization
7. Corolib examples
8. Related work
9. Finalizing

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Distributed systems

- Distributed system = a set of communicating programs running on computers (nodes) in a network
- Communication architectures
  - client-server, e.g. CORBA (Common Object Request Broker Architecture)
  - publish-subscribe, e.g. DDS (Data Distribution Service), ROS (Robot Operating System), uORB
  - peer-to-peer
- Programming styles
  - RMI (Remote Method Invocation), RPC (Remote Procedure Call)
  - Message communication
    - Messages are sent to a mailbox (message queue): applications wait on one or more mailboxes
    - Can be used internally for the implementation of RMIs
    - RMIs are implemented as a request message in one direction, followed by a response message in the other direction
    - Generalization: 0 – 1 request message, N response messages
- Protocol stacks
  - TCP/IP, Bluetooth, USB, …

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Synchronous versus asynchronous communication: explaining the concepts

- (A)synchronicity always requires the presence of two of more entities and the notion of time
  - Entity: process, thread, …
  - At the same time (synchronous), not at the same time (asynchronous)
- Synchronous communication: entity waits internally for information from other entities or from its environment
  - Internally: inside a (library/framework) function called from that entity
  - Waiting for information = synchronization
  - E.g. wait for the response to a request (implementation of RMIs), wait until other application starts processing request
  - (+) natural style for RMIs: remote and local method invocation look (almost) identical
  - (-) not reactive: while waiting internally, the application cannot respond to other inputs
    - For some systems, this behavior is unacceptable, for others, it could only be (slightly) annoying
- Asynchronous communication: entity *cannot* wait internally for information from other entities or environment
  - No (obvious) synchronization between applications in the system
  - Instead, application code has to return control ASAP to a central entry point (event loop) where all information arrives
  - (+) reactive: the application is always ready to accept inputs (either solicited or unsolicited)
  - (-) unnatural implementation style for RMIs

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Synchronous style

- Possible solution to solve the reactivity problem: caller runs every RMI on a dedicated thread
  - Program uses another thread to handle new inputs/requests
  - If these functions/threads share variables, they have to be protected against concurrent access
  - Errors due to incorrect concurrent access may only occur very sporadically and (dis)appear when minor changes are made to the program (Heisenbugs)
  - In other words: errors may be difficult to detect and to correct
  - Threads require a stack of their own, making them resource intensive

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Asynchronous styles: asynchronous method invocation (AMI) + other variants

- Base pattern: the client registers a callback function for every request sent to a remote object
  - N requests sent => N callback functions registered at that time
  - Example: lambda passed to Boost ASIO function
- Alternative 1: all callback functions are registered at start-up
  - Example: signal – slot mechanism of Qt
  - Instead of N functions, just a single function that uses a switch selecting the code corresponding to a response type
  - In case a client program uses the services of M server applications: M (or M x N) functions are registered
- Alternative 2: event-driven system
  - Events (responses, requests) are placed in a mailbox (message queue)
  - (Global) event loop: “get next event”, followed by switch on event type and often a switch on a state variable
  - No distinction between unsolicited event (requests from others) and solicited inputs (responses)
  - Comes down to the use of one big callback function
- Alternative 3: blocking or non-blocking polling (no callback functions used)
- Alternatives 4 – N: (some more alternatives I known and those you know)
- Can they all be replaced by a single “superior” style? (see evolution of asynchronous communication in C#)

AGENDA

1. What is corolib?
2. Brief introduction to C++ coroutines
3. Brief introduction to (a)synchronous distributed programming
4. **Why use coroutines for distributed programming?**
5. Corolib goals and coding style
6. Corolib organization
7. Corolib examples
8. Related work
9. Finalizing

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

C++ coroutines can be used…

- … to write distributed applications using a synchronous style, yet executing/behaving in an efficient responsive asynchronous way
  - Coroutines offer the advantages of both styles while not introducing any new major disadvantages
  - The callback functions can be "hidden" inside the coroutine library
- … as an alternative to threads
  - Several coroutines can run in an interleaved/cooperative way on a single operating system thread

Currently outside the scope of corolib:

- For lazily computed sequences (generators)
- …

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Examples

- **Function with 1 RMI**
- Call stack + function with 1 RMI
- Function with 3 RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- Coroutines as alternative to threads
- Embedded software

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Function with 1 RMI (see next slide for pictures)

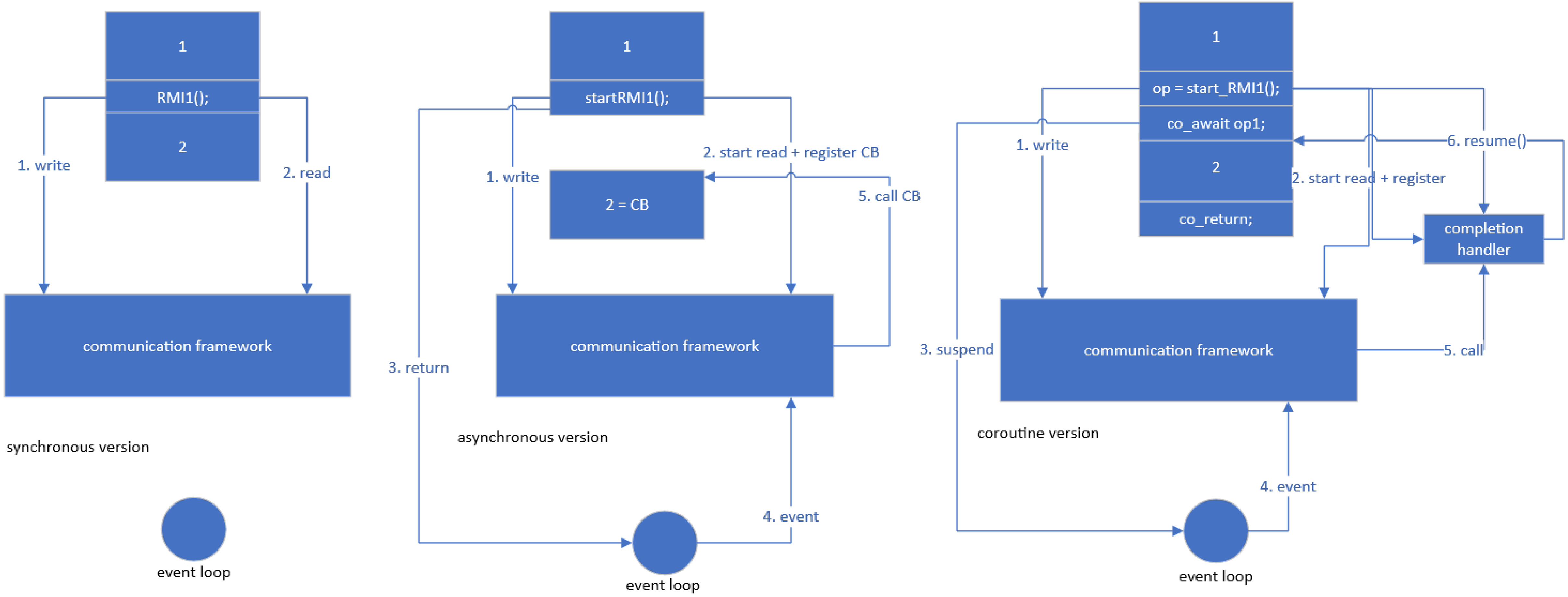
```
// Synchronous
struct Class01s {
    void function1() {
        // Part 1
        ret1 = remoteObj1.op1(in11, in12, out11, out12);
        // Part 2
    }
};
```

```
// Asynchronous
struct Class01a {
    void function1() {
        // Part 1
        remoteObj1.sendc_op1(in11, in12,
            [this](int out1, int out2, int ret1) {
                // Part 2
            });
    }
};
```

```
// Coroutine
struct Class01c {
    async_task<void> coroutine1() {
        // Part 1
        ret1 = co_await remoteObj1.op1(in11, in12, out11, out12);
        // Part 2
    }
};
```


WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Function with 1 RMI (see next slides for explanation)



WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Function with 1 RMI: synchronous style

- RMI at the lowest level is
  - writing the request as a series of bits (onto the connection) to the server
  - and reading the response bit stream from the server
- While waiting for the response to arrive, the application cannot process any other inputs
  - It does not return to the global event loop
- This type of application is not reactive/responsive
  - Already mentioned before
  - Does not matter if function1 call is deeply nested on the call stack

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Function with 1 RMI: asynchronous style

- Split the original function into two functions
- The first function
  - contains the original code up till the point of the RMI
  - sends a request 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
- The second function (completion handler, implemented as lambda in this example)
  - handles the output arguments and return value of the RMI
  - contains the code that followed the RMI in the original function
- When the completion event arrives, the completion handler is called back

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Function with 1 RMI: coroutines (corolib)

- The operation invocation only contains the input arguments
- The output arguments and the return value of the original function have to be placed in new struct
- The coroutine library registers a completion handler with the communication framework
  - This completion handler is application-independent
- At the co_await, 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 to the awaitable, which will resume the coroutine
- The original code does not have to be restructured
- Everything runs on the same thread

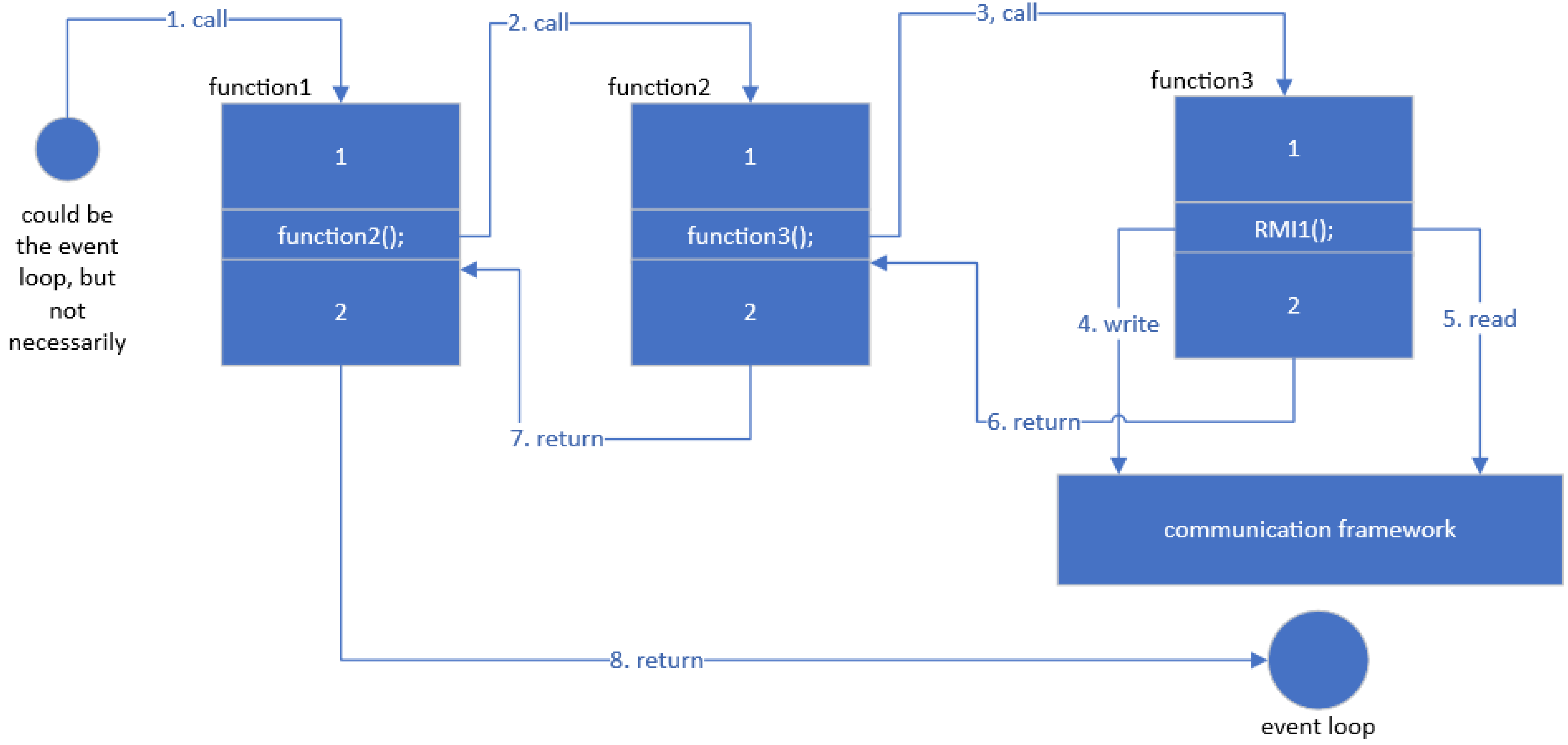
WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Examples

- Function with 1 RMI
- **Call stack + function with 1 RMI**
  - Call stack can be: application call stack, protocol call stack, device driver call stack
- Function with 3 RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- Coroutines as alternative to threads
- Embedded software

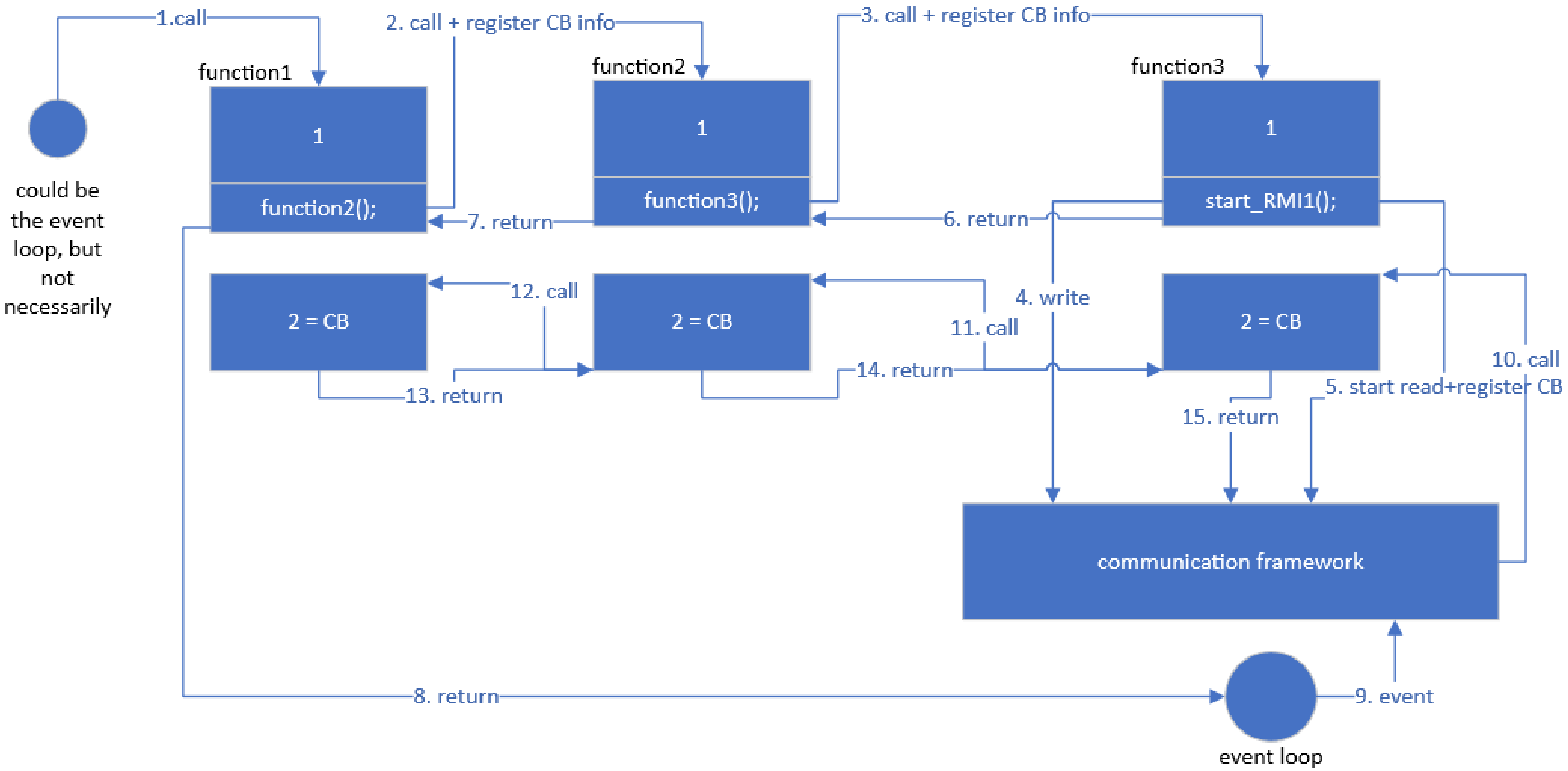
WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Call stack + function with 1 RMI: synchronous style



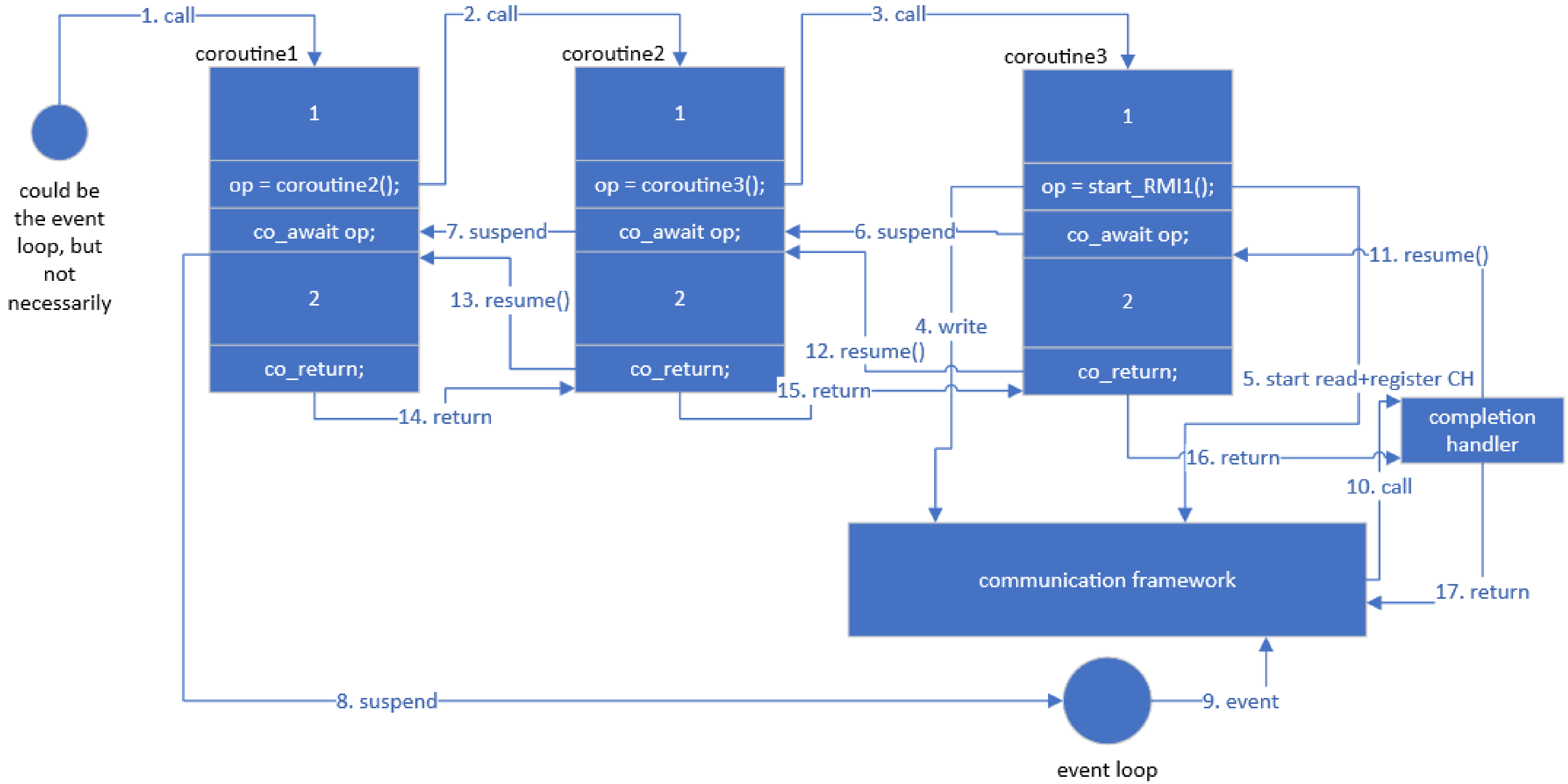
WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Call stack + function with 1 RMI: asynchronous style



WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Call stack + function with 1 RMI: coroutines



WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Call stack + function with 1 RMI (see previous 3 slides for pictures)

Synchronous

- Natural style, but not reactive

Asynchronous

- Original functions have to be split into a “forward” function and a “backward” function
- The backward functions form a chain of callback functions traversed in reversed order
  - The stack of the forward functions has unrolled at the moment the backward functions are called and cannot be used to store information for the backward direction
- Example: use of IRPs (Input/Output Request Packets) in WDM (Windows Driver Model)
- Reactive application, but it is more difficult to follow the flow

Coroutines

- Natural style, reactive again
- Same flow as in the asynchronous case
- The compiler and the coroutine library do all the hard work

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Examples

- Function with 1 RMI
- Call stack + function with 1 RMI
- **Function with 3 RMIs**
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- Coroutines as alternative to threads
- Embedded software

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Synchronous and asynchronous style

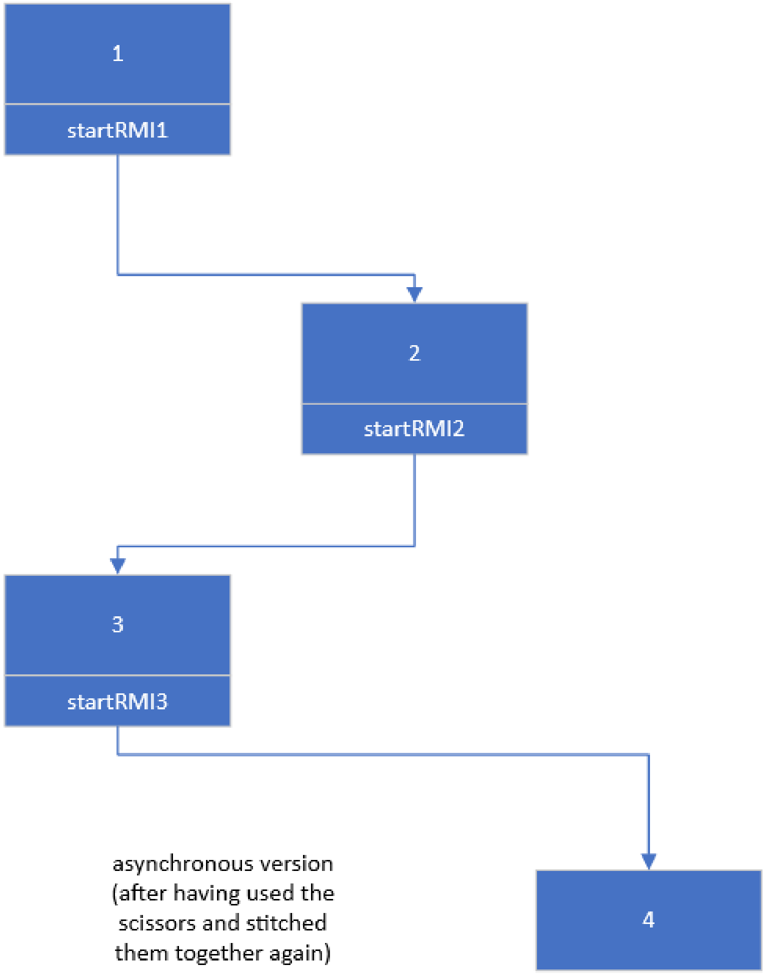
```
struct Class01 {
    void function1() {
        ret1 = remoteObj1.op1(in11, in12, out11, out12);
        // 1 Do stuff
        if (ret1 == val1) {
            ret2 = remoteObj2.op2(in21, in22, out21);
            // 2 Do stuff
        }
        else {
            ret3 = remoteObj3.op3(in31, out31, out32);
            // 3 Do stuff
        }
    }
    void function2() { }
};
```

```
struct Class03 {
    void function1() {
        remoteObj1.sendc_op1(in11, in12,
            [this](int out1, int out2, int ret1)
                { this->function1a(out1, out2, ret1); });
        // 1a Do stuff that doesn't need the result of the RMI
    }

    void function1a(int out11, int out12, int ret1) {
        // 1b Do stuff that needs the result of the RMI
        if (ret1 == val1) {
            remoteObj2.sendc_op2(in21, in22,
                [this](int out1, int ret1)
                    { this->function1b(out1, ret1); });
            // 2a Do stuff that doesn't need the result of the RMI
        }
        else {
            remoteObj3.sendc_op3(in31,
                [this](int out1, int out2, int ret1)
                    { this->function1c(out1, out2, ret1); });
            // 3a Do stuff that doesn't need the result of the RMI
        }
    }

    void function1b(int out21, int ret2) {
        // 2b Do stuff that needs the result of the RMI
    }

    void function1c(int out31, int out32, int ret3) {
        // 3b Do stuff that needs the result of the RMI
    }

    void function2() { }
};
```

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Asynchronous style example with a sequence of RMIs (see next slide for explanation)

synchronous version



asynchronous version (after having used the scissors and stitched them together again)

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Solution with coroutines (two styles)

```
struct Class01
{
    async_task<void> coroutine1()
    {
        ret1 = co_await remoteObj1.op1(in11, in12, out11, out12);
        // 1 Do stuff
        if (ret1 == val1) {
            ret2 = co_await remoteObj2.op2(in21, in22, out21);
            // 2 Do stuff
        }
        else {
            ret3 = co_await remoteObj3.op3(in31, out31, out32);
            // 3 Do stuff
        }
    }
};
```

```
struct Class01
{
    async_task<void> coroutine1()
    {
        async_task<int> op1 = remoteObj1.op1(in11, in12, out11, out12);
        // 1a Do some stuff that doesn't need the result of the RMI
        ret1 = co_await op1;
        // 1b Do stuff that needs the result of the RMI
        if (ret1 == val1) {
            async_task<int> op2 = remoteObj2.op2(in21, in22, out21);
            // 2a Do some stuff that doesn't need the result of the RMI
            ret2 = co_await op2;
            // 2b Do stuff that needs the result of the RMI
        }
        else {
            async_task<int> op3 = remoteObj3.op3(in31, out31, out32);
            // 3a Do some stuff that doesn't need the result of the RMI
            ret3 = co_await op3;
            // 3b Do stuff that needs the result of the RMI
        }
    }
};
```

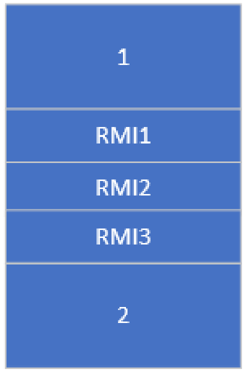
WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Examples

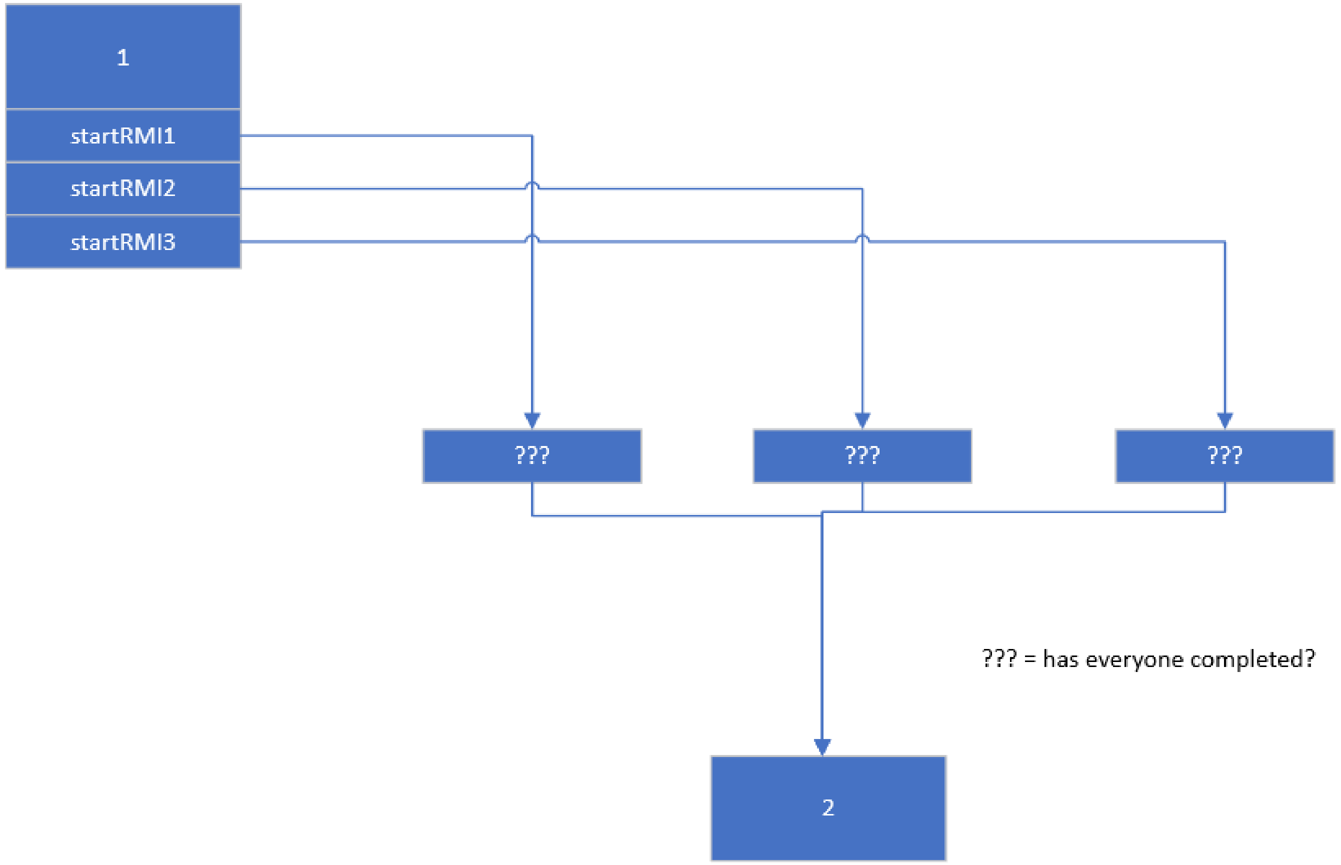
- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 RMIs
- **Function with 3 “parallel” RMIs**
- Function with RMI inside nested loop
- Coroutines as alternative to threads
- Embedded software

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Asynchronous style example with a sequence of RMIs (see next slide for explanation)



synchronous version



asynchronous version

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Asynchronous style example with a sequence of RMIs (see 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 FOR DISTRIBUTED PROGRAMMING?

Examples

- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 RMIs
- Function with 3 “parallel” RMIs
- **Function with RMI inside nested loop**
- Coroutines as alternative to threads
- Embedded software

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Solution with C++ coroutines and corolib

```
struct Class04 {
    void function1() {
        int counter = 0;
        start_time = get_current_time();
        for (int i = 0; i < max_msg_length; i++) {
            Msg msg(i);
            for (int j = 0; j < nr_msgs_to_send; j++) {
                ret1 = remoteObj1.op1(msg);
            }
        }
        elapsed_time = get_current_time() - start_time;
    }
    void function2() { }
};
```

```
struct Class03
{
    async_task<void> coroutine1()
    {
        int counter = 0;
        start_time = get_current_time();
        for (int i = 0; i < max_msg_length; i++) {
            Msg msg(i);
            for (int j = 0; j < nr_msgs_to_send; j++) {
                op1_ret_t res = co_await remoteObject2a.op1(in11, in12);
            }
        }
        elapsed_time = get_current_time() - start_time;
    }
};
```


WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Examples

- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- **Coroutines as alternative to threads**
- Embedded software

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Coroutines as an alternative to threads (see next slide for explanation)

```
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<async_task<int>> wa({ &t1, &t2 });
    co_await wa;
    qDebug() << Q_FUNC_INFO << "end";
    m_timerStartSending.start(100);
    m_selection = nr_message_lengths;
    co_return 0;
}
```

```
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 FOR DISTRIBUTED PROGRAMMING?

Coroutines as an alternative to threads (see previous slide for source code)

- Full source code: corolib/tree/master/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 the case yet
- Repeat the previous three steps for t2 and m_tcpClient2
- Since t1 and t2 have not co_return-ed, measurementLoop44() suspends at the co_wait 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 FOR DISTRIBUTED PROGRAMMING?

Examples

- Function with 1 RMI
- Call stack + function with 1 RMI
- Function with 3 RMIs
- Function with 3 “parallel” RMIs
- Function with RMI inside nested loop
- Coroutines as alternative to threads
- **Embedded software**

WHY USE COROUTINES FOR DISTRIBUTED PROGRAMMING?

Embedded software

- Size: 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)
  - 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
- Coroutines can be used as an alternative to RTK/RTOS threads
  - Use several event queues, each for a different thread priority
  - Interrupt service routines (ISRs) create an event and place it in an event queue
- This approach works as long as the long-running coroutines voluntary return control by calling co_await
  - This could even be using an “artificial” asynchronous operation if no others are available
    - 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 FOR DISTRIBUTED PROGRAMMING?

Conclusion

- 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 overhead (-) Thread communication overhead
asynchronous	(-) More difficult to develop and maintain (+) Reactive	(-) More difficult to develop and maintain (+) Reactive

AGENDA

1. What is corolib?
2. Brief introduction to C++ coroutines
3. Brief introduction to (a)synchronous distributed programming
4. Why use coroutines in distributed applications?
5. **Corolib goals and coding style**
6. Corolib organization
7. Corolib examples
8. Related work
9. Finalizing

COROLIB GOALS

- Implement a C++ coroutine library that can be used to write asynchronous distributed applications
- Demonstrate that synchronous style programs can be executed in a responsive, asynchronous way with only minor modifications to the original program
- Demonstrate that C++ coroutines can be used with existing asynchronous communication frameworks without modifications to their code
  - Boost ASIO, without using any of Boost's stackful or stackless coroutine implementations
  - Qt 5
- Demonstrate that coroutines can be used to replace threads, running the whole application on a single thread (e.g. the main thread)
- Keep the library simple to make it also useable for learning C++ coroutines
- Try to accomplish as much coroutine functionality as possible with a minimum of code

COROLIB CODING STYLE

Preferred coding style

```
// use

    // Start an asynchronous operation on a (remote) object.
    async_operation<aType> retObj = proxy_to_object.start_operation(in1, in2);
    // Do some other things that do not rely on the result of the operation.
    // Finally co_await the result if nothing else can be done in this coroutine.
    aType returnVal = co_await retObj;

// instead of
    aType returnVal = co_await proxy_to_object.start_operation(in1, in2);
```

- While the remote object is processing the request and preparing the response, the application can proceed with actions that do not rely on the response.
  - Requires eager start (defined in the initial suspend/resume point in case of a coroutine)
- When learning coroutines, the first approach shows more clearly what is going on. It shows the return type of the asynchronous operation or coroutine, making it easier to find and examine the implementation.
- corolib allows reusing the return object (retObj in this case) for several asynchronous operation invocations that return an object of the same type. Therefore, the object must be declared explicitly.

AGENDA

1. What is corolib?
2. Brief introduction to C++ coroutines
3. Brief introduction to (a)synchronous distributed programming
4. Why use coroutines in distributed applications?
5. Corolib goals and coding style
6. **Corolib organization**
7. Corolib examples
8. Related work
9. Finalizing

COROLIB ORGANIZATION

corolib/tree/master/include/corolib/

Major classes (independent of Boost or Qt 5)

- async_operation<TYPE>
  - Does not have a promise_type, defines operator co_await()
  - Used as return type of non-coroutine functions to coroutine functions
  - async_operation<TYPE> objects are placed in a list, can be accessed using an index to that list
    - See slides below for an explanation
  - Contains member functions that can be called from completion routines (of the communication framework) to resume the coroutine that has called co_await on the async_operation<TYPE> object
  - Derived from async_operation_base (used internally)
  - Used as the return type of asynchronous (Boost, Qt 5, …) operations implemented inside corolib
- async_task<TYPE>
  - Has a promise_type, defines operator co_await()
  - Used as return type of coroutines (that are part of the application)
  - Can be returned to main() or to another non-coroutine function

COROLIB ORGANIZATION

corolib/tree/master/include/corolib/

Major classes (independent of Boost or Qt 5)

- oneway_task
  - Has a promise_type, does not define operator co_await()
  - “Spawns” a coroutine that will then run “freely”
  - The coroutine can not be co_awaited-for by its launcher
  - Used for example by a server application to spawn a coroutine for every connected client
  - Original implementation in cppcoro

COROLIB ORGANIZATION

corolib/tree/master/include/corolib/

	(no operation co_await)	operator co_await
(no promise_type)	(any struct/class not related to coroutines)	async_operation<TYPE>
promise_type	oneway_task	async_task<TYPE>

COROLIB ORGANIZATION

corolib/tree/master/include/corolib/

Major classes (independent of Boost or Qt 5)

- when_all<TYPE>
  - Allows waiting for all async_operation<TYPE> or async_task<TYPE> objects in its initialization list to be completed
- when_any<TYPE>
  - Allows waiting for any of the async_operation<TYPE> or async_task<TYPE> objects in its initialization list to be completed
  - Returns the index in the list of awaitables that is passed to when_any<TYPE> to find the object that completed
  - For example: select between
    - the response from an operation invocation
    - the expiry of a timer that guards the execution of that operation

COROLIB ORGANIZATION

corolib/tree/master/include/corolib/

Secondary classes (independent of Boost or Qt 5)

- when_all_counter, when_any_one
  - Auxiliary classes used in the implementation of when_all<TYPE>, when_any<TYPE>, async_operation<TYPE> and async_task<TYPE>
- auto_reset_event
  - Does not have a promise type, defines operator co_await()
  - Simplified version of async_operation<TYPE>
  - Can be co_waited for multiple times; must be resumed each time
  - Coroutine way of implementing a semaphore
- Semaphore
  - Binary semaphore used with threads: thread 1 awaits the completion of the semaphore, thread 2 signals the completion
- print
  - Dedicated print that prints a logical thread id as its first output
  - Can be used to trace the control flow in the application and in the corolib library

COROLIB ORGANIZATION

corolib/tree/master/include/corolib/

Communication classes (independent of Boost and Qt 5)

- CommService
  - Used as a base class for the classes on the next slides
  - Contains an array of pointers to async_operation_base derived-class objects

COROLIB ORGANIZATION

corolib/tree/master/include/corolib/

Communication classes (Boost)

- CommCore
  - Contains functionality that is common for client and server:
    - async_operation<void> start_writing(const char* str, int size);
    - async_operation<std::string> start_reading(const char ch = '\n');
    - async_operation<void> start_timer(steady_timer& timer, int ms);
- CommClient
  - Contains client-specific functionality:
    - async_operation<void> start_connecting();
- CommServer
  - Contains server-specific functionality:
    - async_operation<void> start_accepting(spCommCore commRWT);

COROLIB ORGANIZATION

corolib/tree/master/examples/common-qt

Qt 5 – auxiliary classes independent of coroutines

- TcpClient
  - Major functions:
    - bool connectToServer(QString& serverIpAddress, quint16 port);
    - void sendMessage(QByteArray& message);
- TcpServer
  - Major functions:
    - void startListening(quint16 port);
    - void sendMessage(QTcpSocket* sock, QByteArray& message);

COROLIB ORGANIZATION

corolib/tree/master/examples/common-qt

Qt 5 – coroutine related classes

- TcpClientCo
  - Major functions:
    - void sendMessage(QByteArray& message);
    - async_operation<QByteArray> start_reading(bool doDisconnect = true);
    - async_operation<void> start_timer(QTimer& timer, int ms);
    - async_operation<void> start_connecting(QString& serverIpAddress, quint16 port);

COROLIB

Basic usage pattern (1/4)

```
async_operation<std::string>
    CommCore::start_reading(const char ch)
{
    int index = get_free_index();
    async_operation<std::string> ret{ this, index };
    start_reading_impl(index, ch);
    return ret;
}
```

```
void CommCore::start_reading_impl(const int idx, const char ch)
{
    m_input_buffer = "";
    m_bytes = 0;

    boost::asio::async_read_until(
        m_socket,
        boost::asio::dynamic_buffer(m_input_buffer), ch,
        [this, idx](const boost::system::error_code& error,
                    std::size_t bytes)
        {
            async_operation_base* om_async_operation = m_async_operations[idx];
            async_operation<std::string>* om_async_operation_t =
                dynamic_cast<async_operation<std::string>*>(om_async_operation);
            if (!error)
            {
                m_bytes = bytes;
                m_read_buffer = m_input_buffer;
                if (om_async_operation_t)
                {
                    om_async_operation_t->set_result(m_read_buffer);
                    om_async_operation_t->completed();
                }
            }
            else
            {
                // Removed for this slide
            }
        });
}
```

COROLIB

Basic usage pattern (2/4)

async_operation<std::string> CommCore::start_reading(const char ch)

- Prepares an object ‘ret’ of async_operation<std::string> to be returned to a coroutine
- Calls start_reading_impl(index, ch);
  - We want to start the reading operation immediately
- Returns ‘ret’ to the coroutine

void CommCore::start_reading_impl(const int idx, const char ch)

- Starts the asynchronous operation, passing a callback function (a lambda):

```
boost::asio::async_read_until(m_socket, boost::asio::dynamic_buffer(m_input_buffer), ch,
        [this, idx](const boost::system::error_code& error, std::size_t bytes)  {… } );
```

- The lambda is independent of any application logic!
  - The lambda must just inform the async_operation<std::string> object that the operation has completed
- async_operation<std::string> will resume the coroutine that called co_await on it

However…

COROLIB

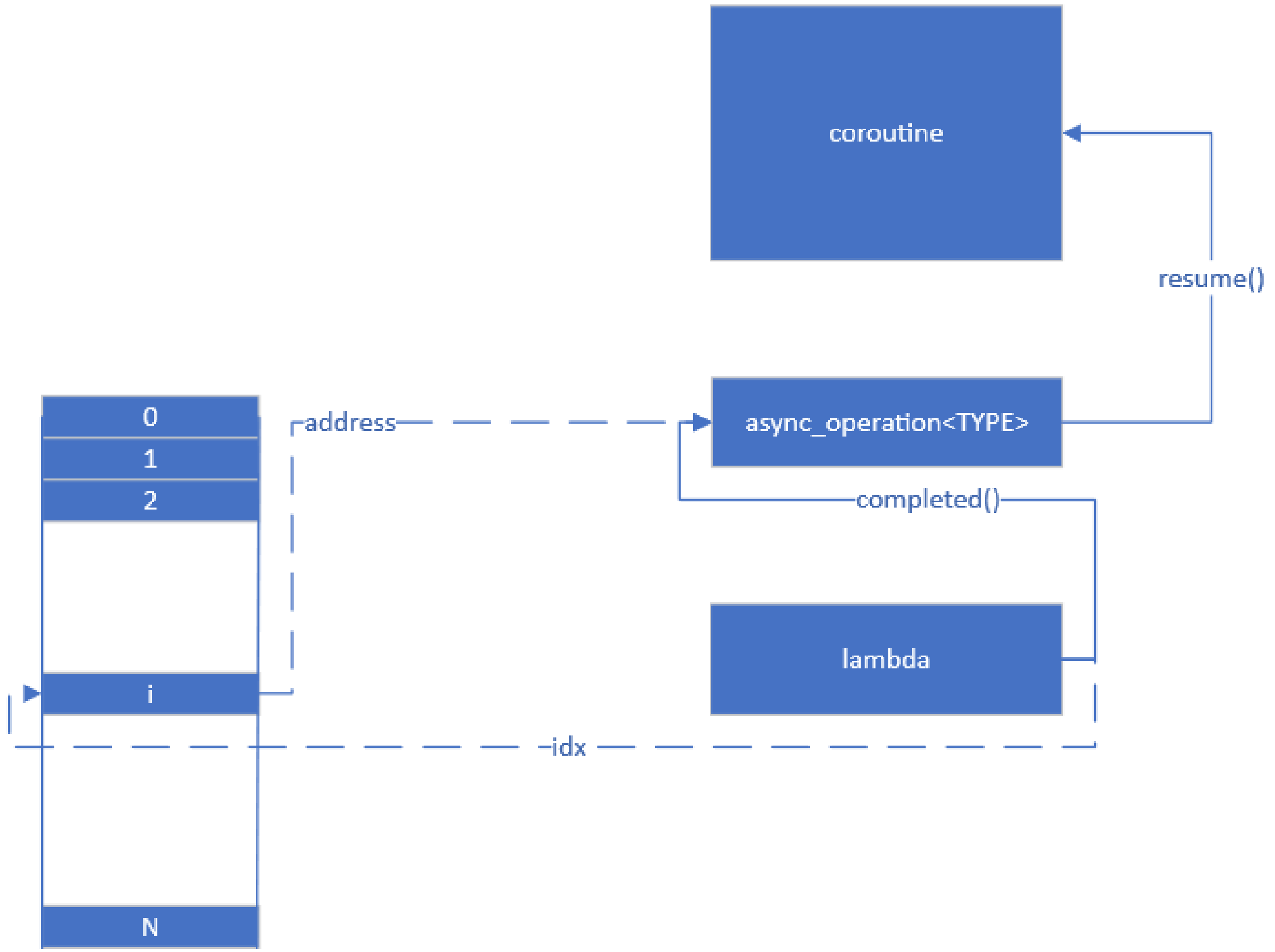
Basic usage pattern (3/4)

However…

- At the moment start_reading calls start_reading_impl, start_reading has not returned an async_operation< std::string> object to its calling coroutine
  - This is still the case at the moment start_reading_impl calls boost::asio::async_read_until and passes the lambda
- There is no final address of the async_operation<std::string> object that we can pass to the lambda
- How can we pass a valid “pointer” to the async_operation<std::string> object at this place in the code?
- Solution: use an index into an array of pointers to async_operation_base objects
  - Remember: async_operation_base is a base class of async_operation<TYPE>
- Every async_operation<TYPE> object writes its address at an index into this array upon construction, also when it is moved

COROLIB

Basic usage pattern (4/4)



AGENDA

1. What is corolib?
2. Brief introduction to C++ coroutines
3. Brief introduction to (a)synchronous distributed programming
4. Why use coroutines in distributed aplications?
5. Corolib goals and coding style
6. Corolib organization
7. **Corolib examples**
8. Other work
9. Finalizing

COROLIB EXAMPLES

corolib/tree/master/examples

Client-server applications using Boost

- clientserver1/
  - Three tier client-server: client, client-server, server
- clientserver2/
  - Two tier client-server application with cancellable actions
- clientserver3/
  - Two tier client-server application with layer of abstraction at the server side to dispatch input requests onto operation invocations
- clientserver4/
  - Two tier client-server application with ROS-like cancellable actions

Client-server applications using Qt

- clientserver11/
  - Two tier client-server application using Qt

COROLIB EXAMPLES

corolib/tree/master/examples

Applications not related to the client-server architecture

- various-boost/
  - At the moment this folder contains only an example using timers in combination with coroutines
- various-qt/
  - At the moment this folder contains only an example using timers in combination with coroutines

Smaller examples not using Boost or Qt

- tutorial/
  - Various introductory examples to the corolib library
  - Contains many of the programs I wrote while learning C++ coroutines
- why-coroutines/
  - Various examples that explain the advantage of C++ coroutines for writing (distributed) applications, see section 4

COROLIB EXAMPLES

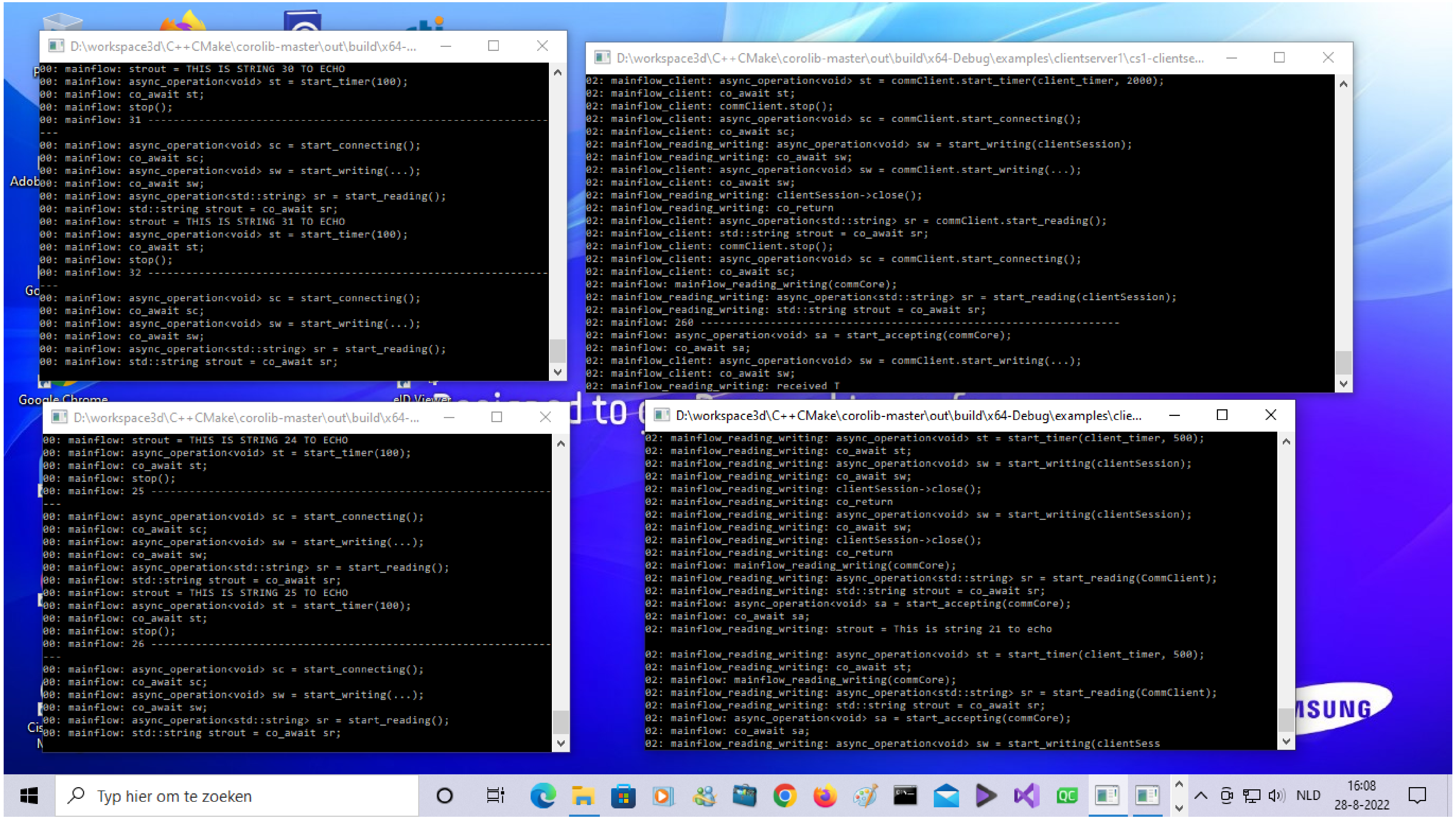
corolib/tree/master/examples

Examples prepared for the Belgian C++ Users Group presentation of 29 Jan 2020

- corolab/
  - “coroutine laboratory”
  - All code for one program is in single file: you just need to open one file to study the whole application
  - Contains examples with generators (co_yield): current absent in corolib because not yet needed
  - Contains examples showing what code could be generated by the compiler to support C++ coroutines
  - Was the basis for the development of corolib
  - Contains variants and experiments that are not useful in the context of corolib
  - Contains examples with and without Boost
  - Unchanged since 1 Feb 2020

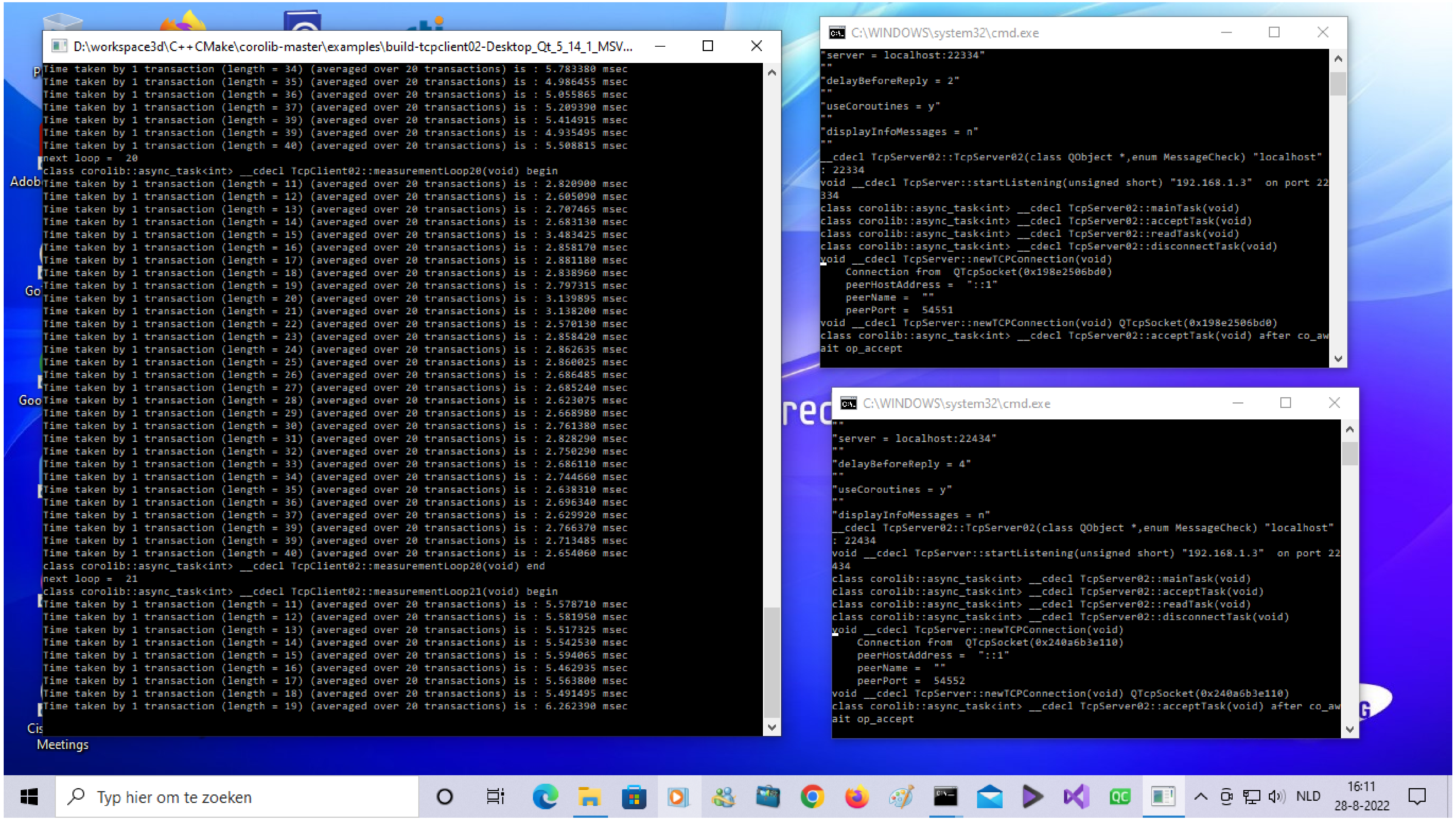
COROLIB @ WORK

clientserver1



COROLIB @ WORK

clientserver11



AGENDA

1. What is corolib?
2. Brief introduction to C++ coroutines
3. Brief introduction to (a)synchronous distributed programming
4. Why use coroutines in distributed applications?
5. Corolib goals and coding style
6. Corolib organization
7. Corolib examples
8. **Related work**
9. Finalizing

RELATED WORK

Comparison cppcoro - corolib

cppcoro	corolib
https://github.com/lewissbaker/cppcoro	https://github.com/JohanVanslembrouck/corolib
Main purpose was to evaluate the implementation of C++ coroutines during the standardization process	Main purpose is to demonstrate the use and usefulness of C++ coroutines for writing distributed applications
	Secondary purpose is to allow newcomers to learn C++ coroutines (tutorial examples, print function)
Uses Win32 overlapped I/O for asynchronous operations	Uses Boost (without its coroutine implementations) and Qt 5 because both libraries are operating system independent
Uses dedicated classes for every asynchronous operation and operation implementation	Uses member functions for every asynchronous operation (returning async_operation<TYPE>) and operation implementation
Uses lazy start: operation object must be co_await-ed before operation is started	Uses eager start: operation started immediately without having to call co_await

RELATED WORK

Comparison Boost ASIO - corolib

Boost ASIO	corolib
https://www.boost.org/users/history/version_1_80_0.html	https://github.com/JohanVanslembrouck/corolib
Stackless coroutines are part of Boost ASIO as overloaded functions: difficult to distinguish the coroutines from the many other functions with the same name	Corolib uses the non-coroutine Boost ASIO functions, and uses a different naming convention for clearer separation: start_operation instead of async_operation
Not possible to split T result = co_await async_operation(…); into two statements because of o.a. private constructors	Split into two statements is my preferred style and natural in case of eager start (see next point)
Uses lazy start: operation must be co_await-ed before operation is started	Uses eager start: operation started immediately without having to call co_await
Difficult to use with other asynchronous communication frameworks	Easy to use with other asynchronous communication frameworks, such as Qt, a proprietary framework (many companies have developed one)
Difficult to read/understand because of the use of a lot of macros to support many C++ versions and OSs	Only has to support C++20

RELATED WORK

Previous work

Presentation at the Belgian C++ Users Group on 29 January 2020

- http://becpp.org/blog/wp-content/uploads/2020/02/Johan-Vanslembrouck-Coroutines-in-C20.zip
- PowerPoint presentation + self-contained source code examples
- (See before)

RELATED WORK

References

Articles (lots of them)

- https://lewissbaker.github.io/
- …

Books

- C++20 for Programmers – An Objects-Natural Approach
  Chapter 18 C++ 20 Coroutines
  Paul Deitel - Harvey Deitel
  Pearson Education Inc., 2022
- C++20: Get the Details
  Section 6.1 Coroutines
  Rainer Grimm
  2021
  https://leanpub.com/c20

AGENDA

1. What is corolib?
2. Brief introduction to C++ coroutines
3. Brief introduction to (a)synchronous distributed programming
4. Why use coroutines in distributed applications?
5. Corolib goals and coding style
6. Corolib organization
7. Corolib examples
8. Related work
9. **Finalizing**

FINALIZING

Conclusions, experiences, thoughts, …

- Coroutines are very useful to write distributed (and many other) applications
  - Minimal impact on the original algorithms/specifications: 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
- Coroutines can reduce the need for threads
- Coroutines can be used as an alternative for threads

- Coroutines are fun!
- IMO coroutines are an essential part of the future of programming
  - And they should have been there already a long time ago… when I started writing software
- Everybody should learn and use coroutines!

FINALIZING

Thank you!

Was everything clear enough?

Do you still have some doubts?

