Capgemini engineering
DISTRIBUTED PROGRAMMING WITH COROUTINES AND ASYNCHRONOUS COMMUNICATION FRAMEWORKS
Johan Vanslembrouck

DISTRIBUTED PROGRAMMING WITH COROUTINES AND ACFS

Goal of this presentation

- Demonstrate how several asynchronous communication frameworks (ACFs), such as
  - Boost ASIO
  - Qt5
  - gRPC
  - Windows overlapped I/O
  - ROS 2 (Robot Operating System 2)
  - CORBA AMI (Common Object Request Broker Architecture - Asynchronous Method Invocation)
  - libevent
  - curl
  - proprietary communication frameworks
  - …
- can be used with a single coroutine library to write responsive, distributed applications using a uniform synchronous/sequential programming style
- without having to change anything to these ACFs or to the coroutine library.

Public © Capgemini 2025. All rights reserved |

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.
  - Many applications were distributed and embedded. (Distributed applications are my favourite kind of applications.)
- 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 (May 2026).
  - During this 41+ years, I have seen a lot of code that could have benefited from the use of (C++) coroutines… if only they had been available at that time.
    - See next slides for a small example.
  - Still, I haven’t had the chance to use coroutines in client projects (although I did see some opportunities).

Public © Capgemini 2025. All rights reserved |

DISTRIBUTED PROGRAMMING WITH COROUTINES AND ACFS

Appetizer example: synchronous style 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.

Public © Capgemini 2025. All rights reserved |

DISTRIBUTED PROGRAMMING WITH COROUTINES AND ACFS

Appetizer example: asynchronous style

```
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.

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;
}
```


Public © Capgemini 2025. All rights reserved |

DISTRIBUTED PROGRAMMING WITH COROUTINES AND ACFS

Appetizer example: source code

Synchronous

- p1200-sync-3rmis.cpp

Asynchronous

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

Coroutines

- p1220-coroutines-3rmis.cpp
- p1222-coroutines-3rmis-generichandler.cpp

Public © Capgemini 2025. All rights reserved |

AGENDA

1. **Brief introduction to C++ coroutines (11 slides)**
2. Brief introduction to (a)synchronous distributed computing
3. Using a single coroutine library with several asynchronous communication frameworks
4. Summary and conclusions
5. Brief introduction to corolib
6. Other coroutine libraries

Public © Capgemini 2025. All rights reserved |

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) 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.

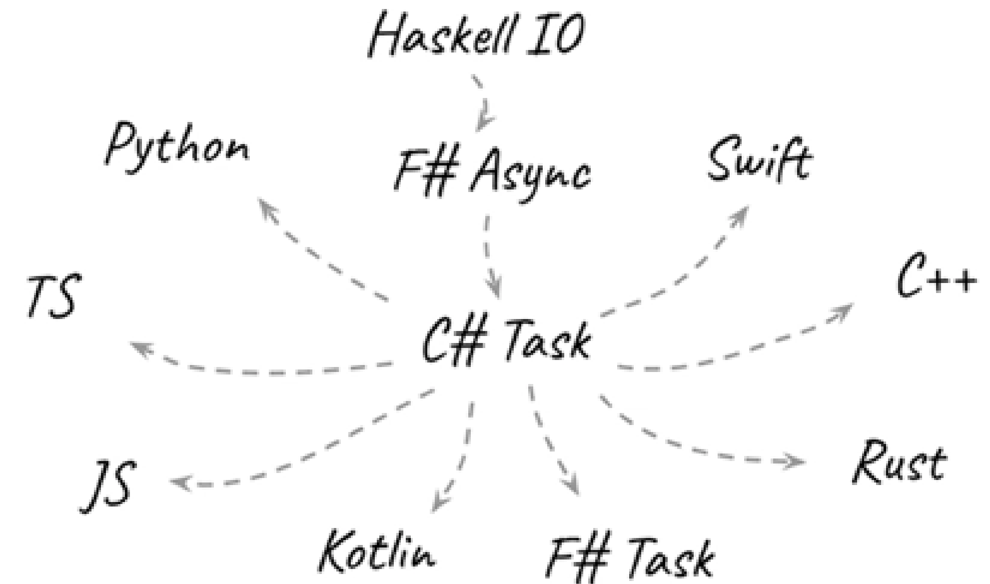
```
PROGRAM SUBDEM
REAL A,B,C,SUM,SUMSQ
CALL INPUT( + A,B,C)
CALL CALC(A,B,C,SUM,SUMSQ)
CALL OUTPUT(SUM,SUMSQ)
END

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
```


Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

A short history of async/await and coroutines

- 2007 – F# 2
- 2012 – C# 5
- 2015 – Python 3.5, TypeScript 1.7
- 2017 – ECMAScript 2017
- 2018 – Kotlin 1.3
- 2019 – Rust
- 2020 – C++ 20
- 2021 – Swift



Source: 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.

Note: In contrast to the other programming languages in this list, C++ 20 defines only low-level primitives; there is no standardized coroutine library yet.

Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

Dynamic C: costatements, cofunctions and slice statements

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).
- A forgotten pioneer?

Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ 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
}
```


Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

What is a C++ coroutine? / How do you recognize a C++ 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 also return an object of a coroutine type.

- It cannot return just an int, void, double, etc.
- Consequently, main() cannot be a coroutine.

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

Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

Implementation (1/5)

task g(parameters)

task taskf = f(args);
...
T res = co_await taskf;
...
co_return res;

translated into

task

m_coro_handle

std::coroutine_handle<promise_type> m_coro_handle;

data

code

Notice that std::coroutine_handle<TYPE> and promise_type are all classes/structs with data + functions.

__g_state

__resume
__destroy
m_continuation
m_result + m_ready
__suspend_point
// argument copies
// local variables

__coroutine_state

Promise __promise;

__coroutine_state_with_promise <__g_promise_t>

__g_state

std::coroutine_handle<> m_continuation;

task g(parameters)

__coroutine_state* __g_resume(__coroutine_state* s)

task taskf = f(args);
...
...
co_await taskf;
T res =
...
...
co_return res;

suspend_point_0

suspend_point_1

final_suspend

void __g_destroy(__coroutine_state* s)

Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

Implementation (2/5)

- The C++ compiler transforms a coroutine into a “coroutine state” class with
  - data members (to preserve state)
  - member functions (usually 3)
- It is possible to instantiate several objects of a coroutine state class:
  - A coroutine can call itself recursively.
  - One coroutine or a function can call the same coroutine several times sequentially and co_await (the completion of) these coroutines afterwards: those instances should be running in a cooperative way.
- The data members preserve the state of a coroutine instance between suspend and resume operations. The data members are:
  - A promise_type object: it contains a.o. the real return value of the coroutine.
    - Because different coroutines may return values of different types, promise_type is defined as a template class.
  - A copy of the arguments.
  - A variable identifying the current state (usually an int).
  - Local variables/temporaries that preserve the data across suspend/resume points (at the co_await statements).

Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

Implementation (3/5)

- A coroutine_handle is a "smart pointer" to a coroutine state object: it has 3 important member functions:
  - resume(): calls the "resume" function (see below)
  - destroy(): class the "destroy" function (see below)
  - done(): determines if the coroutine has run to completion
- The coroutine state class has 3 member functions:
  - The "ramp" static function
  - A "resume" "virtual" function
  - A "destroy" "virtual" function
- Note: the two "virtual" functions ~~can be~~ are usually implemented as C-style functions
  - i.e., as functions defined outside the coroutine state class, taking a pointer to a coroutine state object as first (and only) parameter
  - The base class defines a "virtual function table" with a pointer to the resume and destroy functions.
  - See Linux kernel drivers for lots of examples of this style.

Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

Implementation (4/5)

- Ramp function
  - Is a factory function
  - Has the same signature as the original coroutine
  - Creates the coroutine state object and returns a (not yet standardized) coroutine return type object
    - The coroutine return type object contains at least a coroutine_handle (see previous slide) to the coroutine state object.
    - It often has ‘task’ in its name.
- Resume function
  - Contains the compiler-transformed body of the original coroutine
    - This code can become very complex if the original code contains (nested) conditionals and loops.
  - The original code is usually transformed into a state machine: switch(state) …
  - Every co_await is transformed into a suspend/resume point
  - Suspend point: return from the coroutine without destroying the coroutine state object (using a normal return statement)
    - Save the current suspend/resume point in the state variable
  - On (re)entry of the resume function, it jumps to the resume point indicated by the state variable.

Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

Implementation (5/5)

- Destroy function
  - Is also implemented as a state machine: the data to be destroyed depends on the state of the coroutine.
  - Is called when the coroutine return object (with its coroutine_handle to the coroutine state object) goes out of scope.
  - Note: a coroutine_handle does not have a destructor that calls the destroy function; the coroutine return object must call the destroy function explicitly (from its destructor).
- The C++ compiler also generates
  - an initial suspend point (code that precedes the user-authored coroutine body)
  - a final suspend point (code that follows the user-authored coroutine body)
  - … both allowing a lot of tailoring of the coroutine behaviour (eager versus lazy coroutine start)
- Conclusion: The C++ compiler does all the hard work that a programmer otherwise would have to do when transforming a sequential blocking style application into an asynchronous non-blocking application.

Public © Capgemini 2025. All rights reserved |

INTRODUCTION TO C++ COROUTINES

Further information

- The previous slides only described coroutine implementation from a static point of view (class).
  - The dynamics (control flow) can be the subject of another presentation.
  - Content based upon https://lewissbaker.github.io/2022/08/27/understanding-the-compiler-transform

- Sorry, folks. This is all you can get from me as an introduction to C++ coroutines.
- 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
  - https://becpp.org/blog/wp-content/uploads/2023/11/Lieven-de-Cock-Coroutines.pdf
- Let's move on to the application domain…

Public © Capgemini 2025. All rights reserved |

AGENDA

1. Brief introduction to C++ coroutines
2. **Brief introduction to (a)synchronous distributed programming (9 slides)**
3. Using a single coroutine library with several asynchronous communication frameworks
4. Summary and conclusions
5. Brief introduction to corolib
6. Other coroutine libraries

Public © Capgemini 2025. All rights reserved |

(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., Boost ASIO, Qt, gRPC, Win overlapped I/O, CORBA (e.g., TAO), ROS (actions), libevent
  - publish-subscribe, e.g., DDS (Data Distribution Service), MQTT, ROS (Robot Operating System), uORB
  - peer-to-peer
- Programming styles
  - RMI (Remote Method Invocation), RPC (Remote Procedure Call)
  - Message communication
    - Messages can be used at the top architectural level, or:
    - RMIs are implemented as a **request message** sent to a target application (or service), followed by a **response/reply message** in the other direction.
    - A service / server typically receives and handles request messages from several clients.
    - Usually: 1 request message (with the input arguments) is followed by 1 response/reply message (with the output arguments and return value).
    - In general: 0 – N request messages, 0 – N response/reply messages; e.g., gRPC streams.
    - In general: a client sends a request message to N servers; a client collects response/reply messages from N servers.

Public © Capgemini 2025. All rights reserved |

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Synchronous RPC/RMI applications

- Ideally, the code for a remote call looks identical (or at least very similar) to a local function call.
- The application source code is easy to write, to read and to extend with new RPC/RMIs.
- The code that is generated for the RPC/RMI **waits internally** for the response/reply of the remote party (often called the server).
  - The calling application synchronizes on the response: we call this a **synchronous** application.
- The response may be handled on the same or on a dedicated ‘completion’ thread:
  - Same thread: the application enters a local event loop, waiting (only) for the event corresponding to the response.
  - Dedicated thread: the calling thread and the completion thread synchronize using a condition variable, semaphore, latch...
- In most cases, this delay while waiting for the response is not noticeable by humans.
- However, in time-critical applications, this delay may be unacceptable.
  - While waiting inside an RPC/RMI for a response, the application cannot accept new inputs.
  - See next slide.

Public © Capgemini 2025. All rights reserved |

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Reactive RPC/RMI-style 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)
  - from other applications (in a distributed system)
- The application **must not wait (block)** internally on responses to requests sent
  - to the environment (e.g., to the file system)
  - to other applications (using a communication framework)
- An application that does not wait (block) on responses is often called **asynchronous**.
- These applications enter an **event loop**, from where they pick up
  - new "unsolicited" requests (from the user, another application, the environment)
  - expected "solicited" responses (to a request previously sent by the application to another application or service)
- 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.

Public © Capgemini 2025. All rights reserved |

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Reactive RPC/RMI-style applications: approaches to achieve reactiveness

- Run every RMI on a separate thread.
  - After launching a thread, the application is ready to accept new input.
  - This requires coordination of threads, protecting shared data.
  - In C++: std::thread, std::async, std::future + std::promise.
  - This approach will not be explored further in this presentation.
- Split the original code into smaller pieces that do not block/wait.
  - These pieces only take a small amount of processing time.
  - See next slides.

Public © Capgemini 2025. All rights reserved |

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Synchronous I/O

```
create
open
read
write
close
remove

connect
read
write
disconnect
```

Asynchronous I/O

```
async_create / on_create_complete
async_open / on_open_complete
async_read / on_read_complete
async_write / on_write_complete
async_close / on_close_complete
async_remove / on_remove_complete

async_connect / on_connect_complete
async_read / on_read_complete
async_write / on_write_complete
async_disconnect / on_disconnect_complete
```

The on_XXX_complete functions (user-defined names) can be lambdas, i.e., anonymous functions.

Public © Capgemini 2025. All rights reserved |

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Source code

Synchronous:

- p1000-sync.cpp
- Running main function on a dedicated thread: p1010-sync-thread.cpp

Asynchronous:

- Uses hard-coded coupling between operations: p1020-async.cpp
- Uses hard-coded coupling between operations and lambdas: p1025-async.cpp
- Running main function on a dedicated thread: p1030-async-thread.cpp

Coroutines:

- Restoring sequential flow: p1040-coroutine.cpp
- Restoring sequential flow: p1060-corolib.cpp

Public © Capgemini 2025. All rights reserved |

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Synchronous RMI

```
eventloop
    ret operationA(in1, out1)
        ret1 = operation1(in11, in12, out11);

        ret2 = operation2(in21, out21, out22);

        ret3 = operation3(in31, inout31, out31);
        return res;
```

- The operations are waiting internally on the of the remote server.
- in1 can be used to calculate the inXY and inoutXY arguments of the 3 operation calls.
- All (in)outXY and retX values can be used to calculate the inXY an inoutXY arguments of the following operation calls.

Asynchronous RMI

```
eventloop
    operationA(in1, out1)
        start_operation1(in11, in12);
    operation1_result(out11, ret1)
        start_operation2(in21);
    operation2_result(out21, out22, ret2);
        start_operation3(in31, inout31);
    operation3_result(inout31, out31, ret3);
```

- None of the operations are waiting internally.
- System can always handle other operations in between.
- How to “stitch” these pieces together?
- How to pass variables from one function to another to calculate the inXY and inoutXY arguments?
- Return type of all operations is void.

Public © Capgemini 2025. All rights reserved |

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Synchronous vs asynchronous vs coroutines – source code (1/2)

Synchronous:

- Original example: p2000-sync-3rmis.cpp

Asynchronous:

- Uses hard-coded coupling between operations: operationN_result calls start_operationN+1: p2010-async-3rmis.cpp
- Removes redundant parameters from the previous example: p2011-async-3rmis.cpp
- Uses lambdas to avoid a hard-coded coupling between a start_operationX function and an operationX_result function: p2012-async-3rmis.cpp
- Evolution of the previous example: it introduces the use of an event queue: p2013-async-3rmis.cpp
- Evolution of the previous example: it introduces a thread to restore the sequential flow of the original example: p2015-async+thread-3rmis.cpp
- Variant of the previous example (different main function): p2016-async+thread-3rmis.cpp

Public © Capgemini 2025. All rights reserved |

(A)SYNCHRONOUS DISTRIBUTED PROGRAMMING

Synchronous vs asynchronous vs coroutines – source code (2/2)

Coroutines:

- Uses synchronous completion: Coroutines uses in and reference (out) parameters: p2020-coroutines-3rmis.cpp
- Variant of the previous example: coroutines group the out parameters and the return value in a struct: p2021-coroutines-3rmis.cpp
- Uses asynchronous completion. Coroutines uses in and reference (out) parameters. Introduces start_operationX async_operation functions that call sendc_operationX functions that place a completion handler in an event queue: p2022-coroutines-3rmis.cpp
- Variant of the previous example: coroutines group out parameters and return value in a struct: p2023-coroutines-3rmis.cpp

Public © Capgemini 2025. All rights reserved |

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. **Using a single coroutine library with asynchronous communication frameworks**
    1. Boost ASIO
    2. Qt5
    3. gRPC
    4. Windows overlapped I/O (based upon cppcoro)
    5. Other (not handled in this presentation)
4. Summary and conclusions
5. Brief introduction to corolib
6. Other coroutine libraries

Public © Capgemini 2025. All rights reserved |

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Using a single coroutine library with several asynchronous communication frameworks
   1. **Boost ASIO**
      1. Why using my own coroutine library with Boost ASIO?
      2. Synchronous
      3. Asynchronous
      4. Asynchronous with completion thread
      5. Coroutines
   2. Qt5
   3. gRPC
   4. Windows overlapped I/O (based upon cppcoro)
   5. Other (not handled in this presentation)
4. Summary and conclusions
5. Brief introduction to corolib
6. Other coroutine libraries

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Why using my own coroutine library with Boost ASIO? (1/3)

- My original goal (June 2019) was to learn about coroutines using the Boost ASIO implementation
- … and *not* to write a coroutine library of my own.
- Consider the following code from boost_1_XX_0/libs/asio/example/cpp20/coroutines/echo_server.cpp
- The Boost definitions do not allow to split…

```
for (;;)
{
    std::size_t n = co_await socket.async_read_some(boost::asio::buffer(data), use_awaitable);
    co_await async_write(socket, boost::asio::buffer(data, n), use_awaitable);
}
```

- into the following…

```
for (;;)
{
    auto task1 = co_await socket.async_read_some(boost::asio::buffer(data), use_awaitable);
    // Do some other stuff giving Boost ASIO the time to read
    std::size_t n = co_await task1;
    auto task2 = async_write(socket, boost::asio::buffer(data, n), use_awaitable);
    co_await task2;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Why using my own coroutine library with Boost ASIO? (2/3)

- At the statement std::size_t n = co_await task1;

```
note: 'auto
boost::asio::detail::awaitable_frame_base<Executor>::await_transform(boost::asio::awaitable
<T,Executor>) const': could not deduce template argument for
'boost::asio::awaitable<T,Executor>' from 'T'
```

- Boost uses private/protected/deleted constructors and functions: at the statement co_await task2;

```
error C2280: 'boost::asio::awaitable<T,Executor>::awaitable(const
boost::asio::awaitable<T,Executor> &)': attempting to reference a deleted function
```

- Note that C# and Python allow this splitting.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Why using my own coroutine library with Boost ASIO? (3/3)

- Also, IMO Boost does not make a clear separation between the communication library and coroutines:
  - async_read_some, async_write (and the other ASIO functions) are overloaded and provide variants that return a coroutine_return type.
- Return type declaration in Boost is very complex because of the use of complicated macros (Boost has/wants to be compatible with many C++ compiler versions).
- So, I started writing simple coroutine classes which I then used with Boost ASIO.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Examples

- **Synchronous API**
- Asynchronous API
- Asynchronous API with completion thread
- Using corolib

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Synchronous API (1/1): main() + mainflow()

```
void mainflow(boost::asio::io_context& ioContext, int id)
{
    boost::asio::ip::tcp::socket sock{ioContext};
    sock.connect(ep);

    const std::string message = "This is string " + std::to_string(id) +  " to echo\n";
    boost::asio::write(sock, boost::asio::buffer(message));

    std::string answer;
    boost::asio::read(
            sock,
            boost::asio::dynamic_buffer(answer),
            std::bind(completionCondition, std::ref(answer),
            std::placeholders::_1, std::placeholders::_2));
    sock.close();
}
```

```
int main()
{
    boost::asio::io_context ioContext;
    mainflow(ioContext, 0);
    mainflow(ioContext, 1);
    mainflow(ioContext, 2);
    std::thread thr1([&] { mainflow(ioContext, 10); });
    std::thread thr2([&] { mainflow(ioContext, 11); });
    std::thread thr3([&] { mainflow(ioContext, 12); });
    thr1.join();
    thr2.join();
    thr3.join();
    return 0;
}
```

Uses synchronous API: connect, write, read.

Easy to write, read and modify (such as adding a second write after the read).

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Examples

- Synchronous API
- **Asynchronous API**
- Asynchronous API with completion thread
- Using corolib

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API (1/4): main() + start()

```
int main()
{
    boost::asio::io_context ioContext;
    client cl1(ioContext);
    client cl2(ioContext);
    client cl3(ioContext);
    cl1.start("This is string 0 to echo\n");
    cl2.start("This is string 1 to echo\n");
    cl3.start("This is string 2 to echo\n");
    ioContext.run();
    return 0;
}

// in class client
void start(std::string message)
{
    start_connecting(message);
    deadline_.async_wait(
        [this](const boost::system::error_code& error) {
            check_deadline();
        });
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API (2/4): start_connecting()

```
void start_connecting(std::string message)      // pass message to the function that need it
{
    tcp::resolver::results_type::iterator endpoint_iter;
    deadline_.expires_after(std::chrono::seconds(60));
    std::vector<boost::asio::ip::tcp::endpoint> eps;
    eps.push_back(ep);
    boost::asio::async_connect(
        socket_,
        eps,
        [this, message](const boost::system::error_code& error,
            const tcp::endpoint& result_endpoint) {
            if (stopped_)
                return;
            if (!socket_.is_open())
                start_connecting(message);
            else if (error) {
                socket_.close();
                start_connecting(message);
            }
            else
                start_writing(message);       // hard-coded the next step in the
                                              // completion handler of start_connecting
        });
}
```

The completion handler passed to async_connect is not generic, because it hard-codes the next step (start_writing in this case). We also must pass the message-to-write via start_connecting.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API (3/4): start_writing()

```
void start_writing(std::string message)     // this function needs the message
{
    if (stopped_)
        return;
    const char* str = message.c_str();
    boost::asio::async_write(
        socket_,
        boost::asio::buffer(str, strlen(str)),
        [this](const boost::system::error_code& error,
            std::size_t result_n) {
            if (stopped_)
                return;
            if (!error)
                start_reading();            // hard-coded the next step in the
                                            // completion handler of start_writing
            else
                stop();
        });
}
```

The completion handler passed to async_write is not generic, because it hard-codes the next step (start_reading in this case).

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API (4/4): start_reading()

```
void start_reading()
{
    deadline_.expires_after(std::chrono::seconds(30));
    boost::asio::async_read_until(
        socket_,
        boost::asio::dynamic_buffer(input_buffer_), '\n',
        [this](const boost::system::error_code& error,
            std::size_t n) {
            if (stopped_)
                return;
            if (!error) {
                std::string line(input_buffer_.substr(0, n - 1));
                input_buffer_.erase(0, n);
                if (!line.empty())
                    print("client::handle_read(): received: %s\n", line.c_str());
                stop();
            }
            else
                stop();
        });
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Examples

- Synchronous API
- Asynchronous API
- **Asynchronous API with completion thread**
- Using corolib

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API with completion thread (1/5): main() + mainflow()

```
int main()
{
    mainflow(0);
    mainflow(1);
    mainflow(2);
    return 0;
}

void mainflow(int id)
{
    boost::asio::io_context ioContext;
    client cl(ioContext);
    cl.start_deadline_actor();
    std::thread thr1([&ioContext] { ioContext.run(); });
    cl.start_connecting();
    cl.get_result();    // 1
    cl.start_writing("This is the string " + std::to_string(id) + " to echo\n");
    cl.get_result();    // 2
    cl.start_reading();
    cl.get_result();    // 3
    // Waiting for stop
    cl.get_result();    // 4
    thr1.join();
}
```

- The mainflow() function uses again a synchronous style.
- However, each get_result() call blocks the main thread if the operation did not complete yet.
- Note that ioContext.run() runs on a separate thread. (Not sure if Boost ASIO is 100% thread-safe …)

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API with completion thread (2/5): start_connecting()

```
void start_connecting()
{
    tcp::resolver::results_type::iterator endpoint_iter;
    deadline_.expires_after(std::chrono::seconds(60));
    std::vector<boost::asio::ip::tcp::endpoint> eps;
    eps.push_back(ep);
    boost::asio::async_connect(
        socket_,
        eps,
        [this](const boost::system::error_code& error,
            const tcp::endpoint& result_endpoint) {
            if (stopped_)
                return;
            if (!socket_.is_open())
                start_connecting();
            else if (error) {
                socket_.close();
                start_connecting();
            }
            else
                set_result_and_release();     // general function
        });
}
```

This example uses the asynchronous API, but the completion handler calls a general (application-independent) function set_result_and_release().

(In this specific case, there is no result to pass.)

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API with completion thread (3/5): start_writing()

```
void start_writing(std::string message)
{
    if (stopped_)
        return;
    const char* str = message.c_str();
    boost::asio::async_write(
        socket_,
        boost::asio::buffer(str, strlen(str)),
        [this](const boost::system::error_code& error,
            std::size_t result_n) {
            if (stopped_)
                return;
            if (!error)
                set_result_and_release();    // general function
            else
                stop();
        });
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API with completion thread (4/5): start_reading()

```
void start_reading()
{
    deadline_.expires_after(std::chrono::seconds(30));
    boost::asio::async_read_until(
        socket_,
        boost::asio::dynamic_buffer(input_buffer_), '\n',
        [this](const boost::system::error_code& error,
            std::size_t n) {
            if (stopped_)
                return;
            if (!error) {
                std::string line(input_buffer_.substr(0, n - 1));
                input_buffer_.erase(0, n);
                if (!line.empty())
                    print("client::handle_read(): received: %s\n", line.c_str());
                set_result_and_release();     // general function
                stop();
            }
            else
                stop();
        });
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Asynchronous API with completion thread (5/5)

```
void get_result()
{
    m_sema.acquire();
}

void set_result_and_release()
{
    m_sema.release();
}
```

- The implementation of get_result() and set_result_and_release() is remarkably simple.
- In this simple implementation there is no result to pass or get (“void”).
- Both functions must run on a separate thread, of course.
- The thread calling get_result() blocks/waits until the other thread released the semaphore.
  - So, what problem did we solve?
- Note: a C++20 latch could have been used instead of a semaphore.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Examples

- Synchronous API
- Asynchronous API
- Asynchronous API with completion thread
- **Using corolib**

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Using corolib (1/4): main() + mainflow()

```
// in class ClientApp
async_task<int> mainflow()
{
    int counter = 0;
    for (int i = 0; i < 100; i++)
    {
        async_operation<void> sc = start_connecting();
        co_await sc;
        std::string str1 = "This is string ";
        str1 += std::to_string(counter++);
        str1 += " to echo\n";
        async_operation<void> sw = start_writing(str1.c_str(), str1.length() + 1);
        co_await sw;
        async_operation<std::string> sr = start_reading();
        std::string strout = co_await sr;
        steady_timer client_timer(m_ioContext);
        async_operation<void> st = start_timer(client_timer, 100);
        co_await st;
        stop();
    }
    co_return 0;
}
```

```
int main()
{
    boost::asio::io_context ioContext;

    ClientApp c1(ioContext, ep1);
    async_task<int> si = c1.mainflow();
    ioContext.run();
    return 0;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Using corolib (2/4): start_connecting()

```
async_operation<void> CommClient::start_connecting()
{
    int index = get_free_index_ts();
    async_operation<void> ret{ this, index, true };
    start_connecting_impl(index);
    return ret;
}
void CommClient::start_connecting_impl(const int idx)
{
    m_stopped = false;
    std::vector<boost::asio::ip::tcp::endpoint> eps;
    eps.push_back(m_ep);
    tcp::resolver::results_type::iterator endpoint_iter;
    m_deadline.expires_after(std::chrono::seconds(5));
    std::chrono::high_resolution_clock::time_point now = get_async_operation_info(idx)->start;
    boost::asio::async_connect(m_socket, eps,
    [this, idx, now](const boost::system::error_code& error, const tcp::endpoint& result_endpoint) {
        (void)result_endpoint;
        if (m_stopped)
            return;
        if (!m_socket.is_open())
            start_connecting_impl(idx);
        else if (error) {
            m_socket.close(); start_connecting_impl(idx);
        }
        else
            completionHandler_ts_v(idx, now);
    });
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Using corolib (3/4): start_writing()

```
async_operation<void> CommCore::start_writing(const char* str, int size)
{
    int index = get_free_index_ts();
    async_operation<void> ret{ this, index, true };
    start_writing_impl(index, str, size);
    return ret;
}
void CommCore::start_writing_impl(const int idx, const char* str, int size)
{
    if (m_stopped)
        return;
    std::chrono::high_resolution_clock::time_point now = get_async_operation_info(idx)->start;
    boost::asio::async_write(m_socket, boost::asio::buffer(str, size),
        [this, idx, now](const boost::system::error_code& error, std::size_t result_n) {
            (void)result_n;
            if (m_stopped)
                return;
            if (!error)
                completionHandler_ts_v(idx, now);
            else
                stop();
        });
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Using corolib (4/4): start_reading()

```
async_operation<std::string> CommCore::start_reading(const char ch)
{
    int index = get_free_index_ts();
    async_operation<std::string> ret{ this, index, true };
    start_reading_impl(index, ch);
    return ret;
}
void CommCore::start_reading_impl(const int idx, const char ch)
{
    m_input_buffer = "";
    m_bytes = 0;
    m_deadline.expires_after(std::chrono::seconds(10));
    std::chrono::high_resolution_clock::time_point now = get_async_operation_info(idx)->start;
    boost::asio::async_read_until(m_socket,
        boost::asio::dynamic_buffer(m_input_buffer), ch,
        [this, idx, now](const boost::system::error_code& error, std::size_t bytes) {
            if (m_stopped)
                return;
            if (!error) {
                m_bytes = bytes;
                m_read_buffer = m_input_buffer;
            }
            else
                m_read_buffer = "EOF";
                completionHandler_ts<std::string>(idx, now, m_read_buffer);
        });
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Source code

Synchronous API

- client0.cpp: client application

Asynchronous API

- client1.cpp: client application

Asynchronous API with completion thread

- client2.cpp: client application

Using coroutines (corolib)

- commclient.cpp: client-specific library code: connecting
- commcore.cpp: common client and server library code: reading and writing
- commserver.cpp : server-specific library code: accepting
- client1.cpp: client application of clientserver.cpp
- clientserver.cpp : client application of server.cpp, server application for client1.cpp
- server.cpp : server application for clientserver.cpp

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Example run (Windows 11)

client1

```
00: mainflow: async_operation<void> sw = start_writing(...);
00: mainflow: co_await sw;
00: mainflow: async_operation<std::string> sr = start_reading();
00: mainflow: std::string strout = co_await sr;
00: mainflow: strout = THIS IS STRING 59 TO ECHO
00: mainflow: async_operation<void> st = start_timer(100);
00: mainflow: co_await st;
00: mainflow: stop();
00: mainflow: 60 -------------------------------------------------------
-------------
00: mainflow: async_operation<void> sc = start_connecting();
00: mainflow: co_await sc;
00: mainflow: async_operation<void> sw = start_writing(...);
00: mainflow: does not co_await sw; !!!!!!
00: mainflow: async_operation<std::string> sr = start_reading();
00: mainflow: std::string strout = co_await sr;
```

clientserver

```
02: mainflow_reading_writing: co_await st;
02: mainflow_client: async_operation<void> sw = commClient.start_writing(...);
02: mainflow_client: co_await sw;
02: mainflow_client: async_operation<std::string> sr = commClient.start_reading()
;
02: mainflow_client: std::string strout = co_await sr;
02: mainflow_client: strout = This is string 59 to echo

02: mainflow_client: async_operation<void> st = commClient.start_timer(client_tim
er, 2000);
02: mainflow_client: co_await st;
02: mainflow_reading_writing: async_operation<void> sw = start_writing(clientSess
ion);
02: mainflow_reading_writing: co_await sw;
02: mainflow_reading_writing: clientSession->close();
02: mainflow_reading_writing: co_return
```

client2

```
00: mainflow: async_operation<void> sw = start_writing(...);
00: mainflow: co_await sw;
00: mainflow: async_operation<std::string> sr = start_reading();
00: mainflow: std::string strout = co_await sr;
00: mainflow: strout = THIS IS STRING 53 TO ECHO
00: mainflow: async_operation<void> st = start_timer(100);
00: mainflow: co_await st;
00: mainflow: stop();
00: mainflow: 54 -------------------------------------------------------
-------------
00: mainflow: async_operation<void> sc = start_connecting();
00: mainflow: co_await sc;
00: mainflow: async_operation<void> sw = start_writing(...);
00: mainflow: co_await sw;
00: mainflow: async_operation<std::string> sr = start_reading();
00: mainflow: std::string strout = co_await sr;
00: mainflow: strout = THIS IS STRING 54 TO ECHO
00: mainflow: async_operation<void> st = start_timer(100);
00: mainflow: co_await st;
```

server

```
02: mainflow_reading_writing: co_await sw;
02: mainflow_reading_writing: clientSession->close();
02: mainflow_reading_writing: co_return
02: mainflow: mainflow_reading_writing(commCore);
02: mainflow_reading_writing: async_operation<std::string> sr = start_reading(Co
mmClient);
02: mainflow_reading_writing: std::string strout = co_await sr;
02: mainflow: async_operation<void> sa = start_accepting(commCore);
02: mainflow: co_await sa;
02: mainflow_reading_writing: strout = This is string 60 to echo

02: mainflow_reading_writing: async_operation<void> st = start_timer(client_time
r, 500);
02: mainflow_reading_writing: co_await st;
02: mainflow_reading_writing: async_operation<void> sw = start_writing(clientSes
sion);
02: mainflow_reading_writing: co_await sw;
02: mainflow_reading_writing: clientSession->close();
02: mainflow_reading_writing: co_return
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Example run (Ubuntu 24.04 on WSL)

client1

Windows PowerShell | johan@DESKTOP-8IVUKRL: /

```
00: mainflow: std::string strout = co_await sr;
00: mainflow: strout = THIS IS STRING 21 TO ECHO
00: mainflow: async_operation<void> st = start_timer(100);
00: mainflow: co_await st;
00: mainflow: stop();
00: mainflow: 22 ----------------------------------------------
--------------------
00: mainflow: async_operation<void> sc = start_connecting();
00: mainflow: co_await sc;
00: mainflow: async_operation<void> sw = start_writing(...);
00: mainflow: co_await sw;
00: mainflow: async_operation<std::string> sr = start_reading()
;
00: mainflow: std::string strout = co_await sr;
```

clientserver

Windows PowerShell | johan@DESKTOP-8IVUKRL: /mnt

```
02: mainflow_client: commClient.stop();
02: mainflow_client: co_return;
02: mainflow_client: commClient.stop();
02: mainflow_client: co_return;
02: mainflow_client: strout = This is string 21 to echo

02: mainflow_client: async_operation<void> st = commClient.start_timer(cli
ent_timer, 2000);
02: mainflow_client: co_await st;
02: mainflow_client: strout = This is string 19 to echo

02: mainflow_client: async_operation<void> st = commClient.start_timer(cli
ent_timer, 2000);
02: mainflow_client: co_await st;
```

client2

Windows PowerShell | johan@DESKTOP-8IVUKRL: /n

```
00: mainflow: co_await sc;
00: mainflow: async_operation<void> sw = start_writing(...);
00: mainflow: co_await sw;
00: mainflow: async_operation<std::string> sr = start_reading();
00: mainflow: std::string strout = co_await sr;
00: mainflow: strout = THIS IS STRING 19 TO ECHO
00: mainflow: async_operation<void> st = start_timer(100);
00: mainflow: co_await st;
00: mainflow: stop();
00: mainflow: 20 ----------------------------------------------
--------------------
00: mainflow: async_operation<void> sc = start_connecting();
00: mainflow: co_await sc;
00: mainflow: async_operation<void> sw = start_writing(...);
00: mainflow: does not co_await sw; !!!!!!
00: mainflow: async_operation<std::string> sr = start_reading();
00: mainflow: std::string strout = co_await sr;
```

server

Windows PowerShell | johan@DESKTOP-8IVUKRL: /mnt

```
nt_timer, 500);
02: mainflow_reading_writing: co_await st;
02: mainflow_reading_writing: strout = This is string 22 to echo

02: mainflow_reading_writing: async_operation<void> st = start_timer(clie
nt_timer, 500);
02: mainflow_reading_writing: co_await st;
02: mainflow_reading_writing: async_operation<void> sw = start_writing(cl
ientSession);
02: mainflow_reading_writing: co_await sw;
02: mainflow_reading_writing: async_operation<void> sw = start_writing(cl
ientSession);
02: mainflow_reading_writing: co_await sw;
02: mainflow_reading_writing: clientSession->close();
02: mainflow_reading_writing: co_return
02: mainflow_reading_writing: clientSession->close();
02: mainflow_reading_writing: co_return
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH BOOST ASIO

Source code – all coroutine-related code is in one .cpp file (developed Q4 2019)

Not using coroutines

- p0700s.cpp: server application for p0700c.cpp, p0710c.cpp
- p0700c.cpp: client application of p0700s.cpp
- p0710c.cpp: client application of p0700s.cpp

Using coroutines

- p0800s.cpp: server application for p0800cs.cpp
- p0800cs.cpp: client application of p0800s.cpp, server application for p0800c.cpp, p0810c.cpp, p0820c.cpp
- p0800c.cpp: client application of p0800cs.cpp
- p0810c.cpp: client application of p0800cs.cpp
- p0820c.cpp: client application of p0800cs.cpp
- p0830s.cpp: server application for p0830c.cpp
- p0830c.cpp: client application of p0830s.cpp

Public © Capgemini 2025. All rights reserved |

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Using a single coroutine library with several asynchronous communication frameworks
    1. Boost ASIO
    2. **Qt5**
    3. gRPC
    4. Windows overlapped I/O (based upon cppcoro)
    5. Other (not handled in this presentation)
4. Summary and conclusions
5. References and links

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH QT5

Introduction

- In 2021 I used Qt5 for TCP/IP, TLS, DTLS and serial line communication.
- One subtask was to measure the round-trip delay on a serial line between a Linux (Ubuntu 18.04) laptop and an embedded device running embedded Linux (built with PetaLinux).
  - The laptop sends a request message and waits for the response message from the device.
- The measurement application has a double loop:
  - Outer loop: constructs request messages with several lengths between a minimum and a maximum length.
  - Inner loop: sends a message of a fixed length N times and waits for the response, then calculating the average over N interactions.
- The application could not use a double for loop because of the asynchronous nature of Qt5 (using signals and slots).
- The code used in this presentation uses TCP/IP instead of serial communication (which would require a serial line or null modem and two USB-to-serial converters).
  - The calculation of timing with TCP/IP is less reliable, but this is not important: it is all about using a double loop in combination with an asynchronous API.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH QT5

Example (1/4): application code: measurementLoop40

```
async_task<int> TcpClient02::measurementLoop40(TcpClientCo& tcpClient)
{
    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);
    }
    co_return 0;
}
```

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


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH QT5

Example (2/4): start_reading (first implementation)

```
async_operation<QByteArray> TcpClientCo::start_reading(bool doDisconnect)
{
    int index = get_free_index();
    async_operation<QByteArray> ret{ this, index };
    start_reading_impl(index, doDisconnect);
    return ret;
}

void TcpClientCo::start_reading_impl(const int idx, bool doDisconnect)
{
    m_connections[idx] = connect(this, &TcpClientCo::responseReceivedSig,
        [this, idx, doDisconnect](QByteArray msg) {
            print(PRI2, "%p: TcpClientCo::handle_read(): idx = %d\n", this, idx);
            completionHandler<QByteArray>(idx, msg);
            if (doDisconnect) {
                if (!disconnect(m_connections[idx]))
                    print(PRI1, "%p: TcpClientCo::handle_read(): idx = %d, Warning: disconnect
failed\n", this, idx);
            }
        }
    );
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH QT5

Example (3/4): start_reading (second implementation)

```
async_operation<QByteArray> TcpClientCo1::start_reading()     // no doDisconnect parameter
    int index = get_free_index();
    async_operation<QByteArray> ret{ this, index };          // no doDisconnect parameter
    start_reading_impl(index);
    return ret;
}

void TcpClientCo1::start_reading_impl(const int idx)         // no doDisconnect parameter
{
    m_index_read = idx;      // New statement

    // Original code from tcpclientco.cpp has been moved to connectToServer()
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH QT5

Example (4/4): start_reading (second implementation)

```
// Code from bool TcpClientCo1::connectToServer(QString& serverIPaddress, quint16 serverPort):

connectionInfo->m_socket = socket;
connectionInfo->m_connection_ReadyRead    = connect(socket, &QTcpSocket::readyRead,    this,
&TcpClientCo1::readyReadTcp);
connectionInfo->m_connection_disconnected = connect(socket, &QTcpSocket::disconnected, this,
&TcpClientCo1::disconnectedServer);
connectionInfo->m_connection_stateChanged = connect(socket, &QTcpSocket::stateChanged, this,
&TcpClientCo1::stateChanged);
m_connectionInfoList.append(connectionInfo);

m_connection_read = connect(this, &TcpClientCo1::responseReceivedSig,
    [this](QByteArray msg)
    {
        int idx = m_index_read;
        print(PRI2, "%p: TcpClientCo1::handle_read() lambda: idx = %d\n", this, idx);
        completionHandler<QByteArray>(idx, msg);
    }
);

m_connection_connect = connect(this, &TcpClientCo1::connectedSig,
    [this]()
    {
        int idx = m_index_connect;
        print(PRI2, "%p: TcpClientCo1::handle_connect() lambda: idx = %d\n", this, idx);
        completionHandler_v(idx);
    }
);
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH QT5

Example run (Windows 11)

Windows PowerShell | johan@DESKTOP-8IVUKRL: /mnt/

```
Time taken by 1 transaction (length = 19) (averaged over 20 transactions) is : 13.604095 ms
Time taken by 1 transaction (length = 20) (averaged over 20 transactions) is : 13.413275 ms
Time taken by 1 transaction (length = 21) (averaged over 20 transactions) is : 12.929550 ms
Time taken by 1 transaction (length = 22) (averaged over 20 transactions) is : 13.077855 ms
Time taken by 1 transaction (length = 23) (averaged over 20 transactions) is : 12.803435 ms
Time taken by 1 transaction (length = 24) (averaged over 20 transactions) is : 12.850710 ms
Time taken by 1 transaction (length = 25) (averaged over 20 transactions) is : 13.351850 ms
Time taken by 1 transaction (length = 26) (averaged over 20 transactions) is : 13.729525 ms
Time taken by 1 transaction (length = 27) (averaged over 20 transactions) is : 13.242920 ms
Time taken by 1 transaction (length = 28) (averaged over 20 transactions) is : 12.845230 ms
Time taken by 1 transaction (length = 29) (averaged over 20 transactions) is : 13.170805 ms
Time taken by 1 transaction (length = 30) (averaged over 20 transactions) is : 14.447885 ms
Time taken by 1 transaction (length = 31) (averaged over 20 transactions) is : 15.070605 ms
Time taken by 1 transaction (length = 32) (averaged over 20 transactions) is : 14.120340 ms
Time taken by 1 transaction (length = 33) (averaged over 20 transactions) is : 13.168630 ms
Time taken by 1 transaction (length = 34) (averaged over 20 transactions) is : 13.175865 ms
Time taken by 1 transaction (length = 35) (averaged over 20 transactions) is : 12.968590 ms
Time taken by 1 transaction (length = 36) (averaged over 20 transactions) is : 13.096830 ms
Time taken by 1 transaction (length = 37) (averaged over 20 transactions) is : 13.235675 ms
Time taken by 1 transaction (length = 38) (averaged over 20 transactions) is : 13.303815 ms
Time taken by 1 transaction (length = 39) (averaged over 20 transactions) is : 13.032070 ms
Time taken by 1 transaction (length = 40) (averaged over 20 transactions) is : 13.267825 ms
class corolib::async_task<int> __cdecl TcpClient02::measurementLoop50(class TcpClientCo &,in
t) end
class corolib::async_task<int> __cdecl TcpClient02::measurementLoop51(void) end
00: --- mainTask: measurementLoop52();
class corolib::async_task<int> __cdecl TcpClient02::measurementLoop52(void) begin
class corolib::async_task<int> __cdecl TcpClient02::measurementLoop50(class TcpClientCo &,in
t) begin
class corolib::async_task<int> __cdecl TcpClient02::measurementLoop50(class TcpClientCo &,in
t) begin
Time taken by 1 transaction (length = 11) (averaged over 20 transactions) is : 14.250180 ms
Time taken by 1 transaction (length = 11) (averaged over 20 transactions) is : 24.722900 ms
Time taken by 1 transaction (length = 12) (averaged over 20 transactions) is : 13.551395 ms
Time taken by 1 transaction (length = 13) (averaged over 20 transactions) is : 13.678020 ms
Time taken by 1 transaction (length = 12) (averaged over 20 transactions) is : 24.883960 ms
Time taken by 1 transaction (length = 14) (averaged over 20 transactions) is : 13.563535 ms
Time taken by 1 transaction (length = 15) (averaged over 20 transactions) is : 13.268610 ms
Time taken by 1 transaction (length = 13) (averaged over 20 transactions) is : 24.706070 ms
```

Windows Power | johan@DESKTOP-8I

```
2.18.16.1"  on port 22334
class corolib::async_task<int> __cdecl TcpServer02::accept
Task(void)
class corolib::async_task<int> __cdecl TcpServer02::readTa
sk(void)
class corolib::async_task<int> __cdecl TcpServer02::discon
nectTask(void)
void __cdecl TcpServer::newTCPConnection(void)
    Connection from  QTcpSocket(0x2967d5a1c80)
    peerHostAddress =  "::1"
    peerName =  ""
    peerPort =  61590
void __cdecl TcpServer::newTCPConnection(void) QTcpSocket(
0x2967d5a1c80)
class corolib::async_task<int> __cdecl TcpServer02::accept
Task(void) after co_await op_accept
```

Windows Powe | johan@DESKTOP-8

```
class corolib::async_task<int> __cdecl TcpServer02::mainT
ask(void)
void __cdecl TcpServer::startListening(unsigned short) "1
72.18.16.1"  on port 22434
class corolib::async_task<int> __cdecl TcpServer02::accep
tTask(void)
class corolib::async_task<int> __cdecl TcpServer02::readT
ask(void)
class corolib::async_task<int> __cdecl TcpServer02::disco
nnectTask(void)
void __cdecl TcpServer::newTCPConnection(void)
    Connection from  QTcpSocket(0x21f757308b0)
    peerHostAddress =  "::1"
    peerName =  ""
    peerPort =  61589
void __cdecl TcpServer::newTCPConnection(void) QTcpSocket
(0x21f757308b0)
class corolib::async_task<int> __cdecl TcpServer02::accep
tTask(void) after co_await op_accept
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH QT5

Example run (Ubuntu 24.04 on WSL)

Windows PowerShell | johan@DESKTOP-8IVUKRL: /mnt

```
Time taken by 1 transaction (length = 37) (averaged over 20 transactions) is : 48.005094 ms
Time taken by 1 transaction (length = 40) (averaged over 20 transactions) is : 44.245096 ms
corolib::async_task<int> TcpClient02::measurementLoop60(TcpClientCo&, int) end
Time taken by 1 transaction (length = 38) (averaged over 20 transactions) is : 47.983619 ms
Time taken by 1 transaction (length = 39) (averaged over 20 transactions) is : 48.017982 ms
Time taken by 1 transaction (length = 40) (averaged over 20 transactions) is : 47.993588 ms
corolib::async_task<int> TcpClient02::measurementLoop60(TcpClientCo&, int) end
corolib::async_task<int> TcpClient02::measurementLoop66() end
00: --- mainTask: measurementLoop67();
corolib::async_task<int> TcpClient02::measurementLoop67() begin
corolib::async_task<int> TcpClient02::measurementLoop60(TcpClientCo&, int) begin
corolib::async_task<int> TcpClient02::measurementLoop60(TcpClientCo&, int) begin
Time taken by 1 transaction (length = 11) (averaged over 20 transactions) is : 42.762388 ms
Time taken by 1 transaction (length = 11) (averaged over 20 transactions) is : 47.973647 ms
Time taken by 1 transaction (length = 12) (averaged over 20 transactions) is : 44.611702 ms
Time taken by 1 transaction (length = 12) (averaged over 20 transactions) is : 48.203393 ms
Time taken by 1 transaction (length = 13) (averaged over 20 transactions) is : 44.981557 ms
Time taken by 1 transaction (length = 13) (averaged over 20 transactions) is : 47.969222 ms
Time taken by 1 transaction (length = 14) (averaged over 20 transactions) is : 44.413272 ms
Time taken by 1 transaction (length = 14) (averaged over 20 transactions) is : 48.009384 ms
Time taken by 1 transaction (length = 15) (averaged over 20 transactions) is : 44.990902 ms
Time taken by 1 transaction (length = 15) (averaged over 20 transactions) is : 47.981163 ms
Time taken by 1 transaction (length = 16) (averaged over 20 transactions) is : 44.598268 ms
Time taken by 1 transaction (length = 16) (averaged over 20 transactions) is : 48.007429 ms
Time taken by 1 transaction (length = 17) (averaged over 20 transactions) is : 44.788665 ms
Time taken by 1 transaction (length = 17) (averaged over 20 transactions) is : 47.980803 ms
Time taken by 1 transaction (length = 18) (averaged over 20 transactions) is : 44.174817 ms
Time taken by 1 transaction (length = 18) (averaged over 20 transactions) is : 47.995287 ms
Time taken by 1 transaction (length = 19) (averaged over 20 transactions) is : 44.819877 ms
Time taken by 1 transaction (length = 19) (averaged over 20 transactions) is : 48.000715 ms
Time taken by 1 transaction (length = 20) (averaged over 20 transactions) is : 44.562169 ms
Time taken by 1 transaction (length = 20) (averaged over 20 transactions) is : 47.991483 ms
Time taken by 1 transaction (length = 21) (averaged over 20 transactions) is : 45.197136 ms
Time taken by 1 transaction (length = 21) (averaged over 20 transactions) is : 47.994014 ms
Time taken by 1 transaction (length = 22) (averaged over 20 transactions) is : 43.996500 ms
```

Windows | johan@DESI

```
corolib::async_task<int> TcpServer02::readTask
()
corolib::async_task<int> TcpServer02::disconne
ctTask()
void TcpServer::newTCPConnection()
    Connection from  QTcpSocket(0x615f216bc850
)
    peerHostAddress =  "::ffff:127.0.0.1"
    peerName =  ""
    peerPort =  48716
void TcpServer::newTCPConnection() QTcpSocket(
0x615f216bc850)
corolib::async_task<int> TcpServer02::acceptTa
sk() after co_await op_accept
```

Windows | johan@DESI

```
8.26.177"  on port 22434
corolib::async_task<int> TcpServer02::acceptTa
sk()
corolib::async_task<int> TcpServer02::readTask
()
corolib::async_task<int> TcpServer02::disconne
ctTask()
void TcpServer::newTCPConnection()
    Connection from  QTcpSocket(0x588a5de9c850
)
    peerHostAddress =  "::ffff:127.0.0.1"
    peerName =  ""
    peerPort =  42996
void TcpServer::newTCPConnection() QTcpSocket(
0x588a5de9c850)
corolib::async_task<int> TcpServer02::acceptTa
sk() after co_await op_accept
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH QT5

Source code

- Application: tcpclient02.cpp
- start_reading (first implementation): tcpclientco.cpp
- start_reading (second implementation): tcpclientco1.cpp

Public © Capgemini 2025. All rights reserved |

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Using a single coroutine library with several asynchronous communication frameworks
    1. Boost ASIO
    2. Qt5
    3. **gRPC**
        1. Original examples (not using coroutines)
        2. Using coroutines
    4. Windows overlapped I/O (based upon cppcoro)
    5. Other (not handled in this presentation)
4. Summary and conclusions
5. Brief introduction to corolib
6. Other coroutine libraries

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Introduction

- I started using gRPC with corolib after I introduced the Python implementation at a client in 2023.
  - The Python gRPC implementation is very easy to install and use.
  - The C++ implementation, on the other hand, …
- grpc contains examples that come literally from the gRPC distribution…
- … to which I added examples using corolib.
- The slides below compare 3 examples
  - first showing some original code (not using coroutines)
  - then showing some code using coroutines.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Examples

- **Example 1: greeter: uses 1 RPC**
- Example 2: multiplex: uses 2 RPCs that are started one after the other and are then waited upon
- Example 3: route_guide: uses 1 RPC, streaming reply

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 1a: greeter – 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.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 1b: greeter – with coroutines (corolib)

```
async_task<std::string> SayHelloCo(const std::string& user) {
    HelloRequest request;
    request.set_name(user);
    HelloReply reply;
    ClientContext context;
    Status status;
    co_await start_SayHello(&context, request, reply, status);
    if (status.ok())
        co_return reply.message();
    else
        co_return "RPC failed";
}
async_operation<void> start_SayHello(ClientContext* pcontext, HelloRequest& request,
                                      HelloReply& reply, Status& status) {
    int index = get_free_index();
    async_operation<void> ret{ this, index };
    stub_->async()->SayHello(pcontext, &request, &reply,
        [&status, index, this](Status s) {
            print(PRI1, "start_SayHello: handler\n");
            status = std::move(s);
            completionHandler_v(index);
        });
    return ret;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example run (Windows 11)

Windows PowerShell

```
00: Waiting 1000 milliseconds before exiting
PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\grpc\cpp\helloworld> .\greeter_cb_coroutine_client.exe

00: main: runSayHello(greeter);
00: runSayHello
00: SayHello: pre
01: SayHello: handler
00: SayHello: post
00: runSayHello: Greeter received: Hello world 0
00: SayHello: pre
02: SayHello: handler
00: SayHello: post
00: runSayHello: Greeter received: Hello world 1
00: SayHello: pre
01: SayHello: handler
00: SayHello: post
00: runSayHello: Greeter received: Hello world 2
00: SayHello: pre
02: SayHello: handler
00: SayHello: post
00: runSayHello: Greeter received: Hello world 3
00: SayHello: pre
01: SayHello: handler
00: SayHello: post
00: runSayHello: Greeter received: Hello world 4
00: SayHello: pre
02: SayHello: handler
00: SayHello: post
00: runSayHello: Greeter received: Hello world 5
00: SayHello: pre
01: SayHello: handler
00: SayHello: post
00: runSayHello: Greeter received: Hello world 6
00: SayHello: pre
02: SayHello: handler
00: SayHello: post
00: runSayHello: Greeter received: Hello world 7
00: SayHello: pre
01: SayHello: handler
```

Windows PowerShell

```
-a----         11/1/2025   9:55 AM      152612864 greeter_client.pdb
-a----         11/1/2025   9:55 AM       14396416 greeter_coroutine_client.exe
-a----         11/1/2025   9:55 AM      116459649 greeter_coroutine_client.ilk
-a----         11/1/2025   9:55 AM      151900160 greeter_coroutine_client.pdb
-a----         11/1/2025   9:55 AM       14398464 greeter_coroutine_client2.exe
-a----         11/1/2025   9:55 AM      116472153 greeter_coroutine_client2.ilk
-a----         11/1/2025   9:55 AM      152350720 greeter_coroutine_client2.pdb
-a----         11/1/2025   9:55 AM       14444544 greeter_coroutine_client2a.exe
-a----         11/1/2025   9:55 AM      116660426 greeter_coroutine_client2a.ilk
-a----         11/1/2025   9:55 AM      152498176 greeter_coroutine_client2a.pdb
-a----         11/1/2025   9:55 AM       15353856 greeter_server.exe
-a----         11/1/2025   9:55 AM      124593817 greeter_server.ilk
-a----         11/1/2025   9:55 AM      161361920 greeter_server.pdb
-a----         11/1/2025   9:54 AM           4332 helloworld.grpc.pb.cc
-a----         11/1/2025   9:54 AM          13865 helloworld.grpc.pb.h
-a----         11/1/2025   9:54 AM          19105 helloworld.pb.cc
-a----         11/1/2025   9:54 AM          17094 helloworld.pb.h
-a----         11/1/2025   9:54 AM        2792032 hw_grpc_proto.lib
-a----          6/2/2023   1:05 PM        7627776 libcrypto-3-x64.dll
-a----          6/2/2023   1:07 PM        7487488 libprotobufd.dll
-a----          6/2/2023   1:05 PM        1096704 libssl-3-x64.dll
-a----          6/2/2023   1:09 PM        2069504 re2.dll
-a----         8/31/2025   4:25 PM            455 run.bat
-a----          6/2/2023   1:10 PM         209920 zlibd1.dll


PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\grpc\cpp\hellowor
ld> .\greeter_server.exe
Server listening on 0.0.0.0:50051
```

Learn about this picture

Public © Capgemini 2025. All rights reserved

USING COROUTINES WITH GRPC

Examples

- Example 1: greeter: uses 1 RPC
- **Example 2: multiplex: uses 2 RPCs that are started one after the other and are then waited upon**
- Example 3: route_guide: uses 1 RPC, streaming reply

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 2a: multiplex – 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.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 2b: multiplex – with coroutines (corolib) (1/5)

```
async_task<void> SayHello_GetFeatureCo_when_all() {
    async_task<std::string> t1 = SayHelloCo();
    async_task<std::string> t2 = GetFeatureCo();
    when_all wa(t1, t2);
    co_await wa;
    std::cout << t1.get_result();
    std::cout << t2.get_result();
    co_return;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 2b: multiplex – with coroutines (corolib) (2/5)

```
async_task <std::string> SayHelloCo() {
    ClientContext hello_context;
    helloworld::HelloRequest hello_request;
    helloworld::HelloReply hello_response;
    hello_request.set_name("coroutine user");

    Status hello_status = co_await start_SayHello(&hello_context, hello_request, hello_response);
    std::stringstream strstr;
    // Act upon the status of the actual RPC.
    if (hello_status.ok()) {
        strstr << "Greeter received: " << hello_response.message() << std::endl;
    }
    else {
        strstr << "Greeter failed: " << hello_status.error_message() << std::endl;
    }
    co_return strstr.str();
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 2b: multiplex – with coroutines (corolib) (3/5)

```
async_operation<Status> start_SayHello(ClientContext* pcontext,
                                       helloworld::HelloRequest& request,
                                       helloworld::HelloReply& reply) {
    int index = get_free_index();
    async_operation<Status> ret{ this, index };
    helloworld::Greeter::NewStub(channel_)->async()->SayHello(pcontext, &request, &reply,
        [index, this](Status s) {
            print(PRI5, "start_SayHello - completion handler\n");
            Status status = std::move(s);
            completionHandler<Status>(index, status);
        });
    return ret;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 2b: multiplex – with coroutines (corolib) (4/5)

```
async_task<std::string> GetFeatureCo() {
    ClientContext feature_context;
    routeguide::Point feature_request;
    routeguide::Feature feature_response;

    feature_request.set_latitude(50);
    feature_request.set_longitude(100);
    Status feature_status = co_await start_GetFeature(&feature_context,
                                                       feature_request, feature_response);
    std::stringstream strstr;
    if (feature_status.ok()) {
        strstr << "Found feature: " << feature_response.name() << std::endl;
    }
    else {
        strstr << "Getting feature failed: " << feature_status.error_message() << std::endl;
    }
    co_return strstr.str();
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 2b: multiplex – with coroutines (corolib) (5/5)

```
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) {
            print(PRI5, "start_GetFeature - completion handler\n");
            Status status = std::move(s);
            completionHandler<Status>(index, status);
        });
    return ret;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example run (Windows 11)

Windows PowerShell

```
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
Greeter received: Hello coroutine user
Found feature: Feature: latitude: 50, longitude: 100
00: ----------------------------------------------------------------
00:     cons    dest    diff    max     c>p     c<p     c>h     c<h
00: ope 201     201     0       3
00: cor 300     300     0       3       0       300     300     0
00: pro 300     300     0       3       0       300
00: fin 300     300     0       2                       0       0
00: ----------------------------------------------------------------
00: Waiting 1000 milliseconds before exiting
PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\grpc\cpp\multiplex>
```

Windows PowerShell

```
-a----        11/1/2025   9:56 AM     122019598 multiplex_coroutine_client3-when_all.ilk
-a----        11/1/2025   9:56 AM     157822976 multiplex_coroutine_client3-when_all.pdb
-a----        11/1/2025   9:56 AM      15225856 multiplex_coroutine_client3-when_any.exe
-a----        11/1/2025   9:56 AM     122049049 multiplex_coroutine_client3-when_any.ilk
-a----        11/1/2025   9:56 AM     157945856 multiplex_coroutine_client3-when_any.pdb
-a----        11/1/2025   9:56 AM      15208448 multiplex_coroutine_client3.exe
-a----        11/1/2025   9:56 AM     121956364 multiplex_coroutine_client3.ilk
-a----        11/1/2025   9:56 AM     157855744 multiplex_coroutine_client3.pdb
-a----        11/1/2025   9:56 AM      15259136 multiplex_coroutine_client4.exe
-a----        11/1/2025   9:56 AM     122170568 multiplex_coroutine_client4.ilk
-a----        11/1/2025   9:56 AM     158052352 multiplex_coroutine_client4.pdb
-a----        11/1/2025   9:56 AM      15268864 multiplex_coroutine_client5.exe
-a----        11/1/2025   9:56 AM     122263552 multiplex_coroutine_client5.ilk
-a----        11/1/2025   9:56 AM     158052352 multiplex_coroutine_client5.pdb
-a----        11/1/2025   9:56 AM      16241664 multiplex_server.exe
-a----        11/1/2025   9:56 AM     133293505 multiplex_server.ilk
-a----        11/1/2025   9:56 AM     167628800 multiplex_server.pdb
-a----         6/2/2023   1:09 PM       2069504 re2.dll
-a----        11/1/2025   9:54 AM         11458 route_guide.grpc.pb.cc
-a----        11/1/2025   9:54 AM         43834 route_guide.grpc.pb.h
-a----        11/1/2025   9:54 AM         51136 route_guide.pb.cc
-a----        11/1/2025   9:54 AM         52279 route_guide.pb.h
-a----        8/31/2025   4:25 PM           350 run.bat
-a----         6/2/2023   1:10 PM        209920 zlibd1.dll


PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\grpc\cpp\multiplex> .\multip
lex_server.exe
Server listening on 0.0.0.0:50051
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Examples

- Example 1: greeter: uses 1 RPC
- Example 2: multiplex: uses 2 RPCs that are started one after the other and are then waited upon
- **Example 3: route_guide: uses 1 RPC, streaming response**

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 3a: route_guide – original example without coroutines

```
class Reader : public grpc::ClientReadReactor<Feature> {
public:
    Reader(RouteGuide::Stub* stub, float coord_factor,
        const routeguide::Rectangle& rect)
        : coord_factor_(coord_factor) {
        stub->async()->ListFeatures(&context_, &rect, this);
        StartRead(&feature_);
        StartCall();
    }
    void OnReadDone(bool ok) override {
        if (ok) {
            // output statement deleted to save some space
            StartRead(&feature_);
        }
    }
    void OnDone(const Status& s) override {
        std::unique_lock<std::mutex> l(mu_);
        status_ = s;
        done_ = true;
        cv_.notify_one();            // looks like handle.resume()
    }
    Status Await() {                 // looks like co_await
        std::unique_lock<std::mutex> l(mu_);
        cv_.wait(l, [this] { return done_; });
        return std::move(status_);
    }
private:
    // member variables omitted to save some space
};
```

Class Reader.

Function Await() looks like a co_await...

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 3b: route_guide – with coroutines (corolib) (1/2)

```
class ReaderCo : public grpc::ClientReadReactor<Feature> {
public:
    ReaderCo(RouteGuide::Stub* stub, float coord_factor,
        const routeguide::Rectangle& rect)
        : coord_factor_(coord_factor) {
        stub->async()->ListFeatures(&context_, &rect, this);
        StartRead(&feature_);
        StartCall();
    }
    void OnReadDone(bool ok) override {
        if (ok) {
           // output statement deleted to save some space
            StartRead(&feature_);
        }
    }
    void OnDone(const Status& s) override {
        print(PRI1, "ReaderCo::OnDone\n");
        status_ = s;
        eventQueueThr.push(
            [this]() {
                this->completionHandler_(status_);
            });
    }
    void setCompletionHandler(std::function<void(Status)>&& completionHandler) {
        completionHandler_ = std::move(completionHandler);
    }
private:
    // member variables omitted to save some space
};
```

Class ReaderCo.

The implementation of function OnDone has changed, and there is no function Await anymore.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example 3b: route_guide – with coroutines (corolib) (2/2)

```
async_task<void> ListFeaturesCo() {
  routeguide::Rectangle rect;
  Feature feature;
  // Initialize rect (omitted)
  ReaderCo reader(stub_.get(), kCoordFactor_, rect);
  async_operation<ReaderResult> op = start_ListFeatures(&reader);
  op.auto_reset(true);
  bool done = false;
  do {
      ReaderResult result = co_await op;
      if (result.state == ReaderState::Value) {
          // Print result (omitted)
      }
      else if (result.state == ReaderState::Done) {
          done = true;
          // ...
      }
  } while (!done);
  co_return;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Example run (Windows 11)

Windows PowerShell

```
Got message Second message at 0, 1
Got message Second message at 0, 1
Got message Second message at 0, 1
Got message Second message at 0, 1
Got message Third message at 1, 0
Got message Third message at 1, 0
Got message Third message at 1, 0
Got message Third message at 1, 0
Got message Third message at 1, 0
Got message Third message at 1, 0
Got message Third message at 1, 0
Got message First message at 0, 0
Got message Fourth message at 0, 0
Got message First message at 0, 0
Got message Fourth message at 0, 0
Got message First message at 0, 0
Got message Fourth message at 0, 0
Got message First message at 0, 0
Got message Fourth message at 0, 0
Got message First message at 0, 0
Got message Fourth message at 0, 0
Got message First message at 0, 0
Got message Fourth message at 0, 0
Got message First message at 0, 0
Got message Fourth message at 0, 0
Got message First message at 0, 0
00: completionHandler called
00: RouteChatCo: after co_await
00: main: tlf6.wait();
00: Leaving main
00: ---------------------------------------------------------------------
00:      cons    dest    diff    max     c>p     c<p     c>h     c<h
00: ope  19      19      0       2
00: cor  24      24      0       7       0       24      24      0
00: pro  24      24      0       7       0       24
00: fin  24      24      0       6                       0       0
00: ---------------------------------------------------------------------
00: Waiting 1000 milliseconds before exiting
PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\grpc\cpp\route_guide>
```

Windows PowerShell

```
-a----        11/1/2025   9:56 AM     155086848 route_guide_callback_client.pdb
-a----        11/1/2025   9:56 AM      16020480 route_guide_callback_server.exe
-a----        11/1/2025   9:56 AM     126041305 route_guide_callback_server.ilk
-a----        11/1/2025   9:56 AM     162328576 route_guide_callback_server.pdb
-a----        11/1/2025   9:56 AM      15166464 route_guide_client.exe
-a----        11/1/2025   9:56 AM     118984765 route_guide_client.ilk
-a----        11/1/2025   9:56 AM     154619904 route_guide_client.pdb
-a----        11/1/2025   9:56 AM      15349760 route_guide_coroutine_client.exe
-a----        11/1/2025   9:56 AM     119663823 route_guide_coroutine_client.ilk
-a----        11/1/2025   9:56 AM     155750400 route_guide_coroutine_client.pdb
-a----        11/1/2025   9:56 AM      15348224 route_guide_coroutine_client2.exe
-a----        11/1/2025   9:56 AM     119721561 route_guide_coroutine_client2.ilk
-a----        11/1/2025   9:56 AM     155783168 route_guide_coroutine_client2.pdb
-a----        11/1/2025   9:56 AM      15349760 route_guide_coroutine_client_a.exe
-a----        11/1/2025   9:56 AM     119662789 route_guide_coroutine_client_a.ilk
-a----        11/1/2025   9:56 AM     156209152 route_guide_coroutine_client_a.pdb
-a----        8/31/2025   4:25 PM         13768 route_guide_db.json
-a----        11/1/2025   9:54 AM        920726 route_guide_helper.lib
-a----        11/1/2025   9:56 AM      15884288 route_guide_server.exe
-a----        11/1/2025   9:56 AM     125037992 route_guide_server.ilk
-a----        11/1/2025   9:56 AM     161787904 route_guide_server.pdb
-a----        8/31/2025   4:25 PM           251 run.bat
-a----         6/2/2023   1:10 PM        209920 zlibd1.dll


PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\grpc\cpp\route_guide> .\rout
e_guide_server.exe
DB parsed, loaded 100 features.
Server listening on 0.0.0.0:50051
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH GRPC

Source code

Example 1: greeter

- a - original: greeter_callback_client.cc
- b - with coroutines: greeter_cb_coroutine_client.cc

Example 2: multiplex

- a - original: multiplex_client2.cc
- b – with coroutines: multiplex_coroutine_client2.cc, multiplex_coroutine_client3-when_all.cc

Example 3: route_guide

- a - original: route_guide_callback_client.cc
- b – with coroutines: route_guide_coroutine_client.cc

Public © Capgemini 2025. All rights reserved |

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Using a single coroutine library with several asynchronous communication frameworks
    1. Boost ASIO
    2. Qt5
    3. gRPC
    4. **Windows Overlapped I/O (based upon cppcoro)**
    5. Other (not handled in this presentation)
4. Summary and conclusions
5. Brief introduction to corolib

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

Introduction to cppcoro

- In 2019 I started learning coroutines using cppcoro: https://github.com/lewissbaker/cppcoro
- This library uses Windows Overlapped I/O as its communication library.
- The library is not maintained since about 2019 by its original author (Lewis Baker): it still uses the “experimental” include files.
- I included and updated the core part of cppcoro in corolib.
  - This allows using the corolib coroutine classes with the Windows Overlapped I/O backend of cppcoro.
- Maintained version of corolib: https://github.com/andreasbuhr/cppcoro
- Unfortunately, the integration of the cppcoro coroutine classes with the communication backend was a bit too tight: some modifications were necessary to one file to allow using the corolib coroutine classes.

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

Examples

- **cppcoro**
- corolib

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

cppcoro (1/4): main, mainflow, task2

```
int main()
{
    mainflow();
    return 0;
}

void mainflow()
{
    io_service ioSvc;
     std::string serverAddressStr = readServerAddress();
     auto serverAddress = cppcoro::net::ipv4_endpoint::from_string(serverAddressStr);
     (void) sync_wait(
        when_all(
            task1(ioSvc, serverAddress),
            task2(ioSvc)
        ));
}

task<int> task2(io_service& ioSvc)
{
    ioSvc.process_events();
    co_return 0;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

cppcoro (2/4): task1, echoclient

```
task<int> task1(io_service& ioSvc, std::optional<ipv4_endpoint>& serverAddress)
{
    auto stopOnExit = on_scope_exit([&] { ioSvc.stop(); });
    co_await echoClient(ioSvc, serverAddress);
    co_return 0;
}

task<int> echoClient(io_service& ioSvc, std::optional<ipv4_endpoint>& serverAddress)
{
    socket connectingSocket = socket::create_tcpv4(ioSvc);
    connectingSocket.bind(ipv4_endpoint{});
    co_await connectingSocket.connect(*serverAddress);
    co_await when_all(send(connectingSocket), receive(connectingSocket));
    co_await connectingSocket.disconnect();
    co_return 0;
};
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

cppcoro (3/4): send

```
task<int> send(socket& connectingSocket)
{
    std::uint8_t buffer[100];
    std::size_t totalBytesSent = 0;
    for (std::uint64_t i = 0; i < 1000; i += sizeof(buffer)) {
        for (std::size_t j = 0; j < sizeof(buffer); ++j)
            buffer[j] = 'a' + ((i + j) % 26);
        std::size_t bytesSent = 0;
        do {
            bytesSent +=
                co_await connectingSocket.send(buffer + bytesSent, sizeof(buffer) - bytesSent);
             totalBytesSent += bytesSent;
        } while (bytesSent < sizeof(buffer));
    }
    connectingSocket.close_send();
    co_return 0;
};
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

cppcoro (4/4): receive

```
task<int> receive(socket& connectingSocket)
{
    std::uint8_t buffer[100];
    std::uint64_t totalBytesReceived = 0;
    std::size_t bytesReceived;
    do {
        bytesReceived = co_await connectingSocket.recv(buffer, sizeof(buffer));
        for (std::size_t i = 0; i < bytesReceived; ++i)
        {
            std::uint64_t byteIndex = totalBytesReceived + i;
            std::uint8_t expectedByte = 'a' + (byteIndex % 26);
            if (buffer[i] != expectedByte)
                std::cout << "buffer[i] != expectedByte\n";
        }
        totalBytesReceived += bytesReceived;
    } while (bytesReceived > 0);
    co_return 0;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

Example run cppcoro (Windows 11)

Windows PowerShell

```
send: bytesSent = 100
send: bytesSent = 100
send: bytesSent = 100
send: bytesSent = 100
send: bytesSent = 100
send: bytesSent = 100
send: bytesSent = 100
send: totalBytesSent = 1000
receive: bytesReceived = 100
receive: totalBytesReceived = 100
receive: bytesReceived = 100
receive: totalBytesReceived = 200
receive: bytesReceived = 100
receive: totalBytesReceived = 300
receive: bytesReceived = 100
receive: totalBytesReceived = 400
receive: bytesReceived = 48
receive: totalBytesReceived = 448
receive: bytesReceived = 100
receive: totalBytesReceived = 548
receive: bytesReceived = 100
receive: totalBytesReceived = 648
receive: bytesReceived = 100
receive: totalBytesReceived = 748
receive: bytesReceived = 100
receive: totalBytesReceived = 848
receive: bytesReceived = 100
receive: totalBytesReceived = 948
receive: bytesReceived = 52
receive: totalBytesReceived = 1000
receive: bytesReceived = 0
receive: totalBytesReceived = 1000
receive: totalBytesReceived = 1000
main: leaving
PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\cppcoro\exam
```

Windows PowerShell

```
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 64
echoServer: bytesSent = 64
echoServer: bytesReceived = 40
echoServer: bytesSent = 40
echoServer: bytesReceived = 0
echoServer: totalBytesReceived = 0, totalBytesSent = 1000
echoServer: leaving
main: leaving
PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\cppcoro\exa
mples-cc>
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

Examples

- cppcoro
- **corolib**

Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

corolib (1/4): main, mainflow

```
int main()
{
    io_service ioSvc;
    async_task<int> t = mainflow(ioSvc);
    ioSvc.process_events();
    int v = t.get_result();
    return 0;
}

async_task<int> mainflow(io_service& ioSvc)
{
    std::string serverAddressStr = readServerAddress();
    auto serverAddress = cppcoro::net::ipv4_endpoint::from_string(serverAddressStr);
    co_await echoClient(ioSvc, serverAddress);
    co_return 0;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

corolib (2/4): echoClient

```
async_task<int> echoClient(io_service& ioSvc, std::optional<ipv4_endpoint>& serverAddress)
{
    socket connectingSocket = socket::create_tcpv4(ioSvc);
    connectingSocket.bind(ipv4_endpoint{});
    socket_wrapper sw(connectingSocket);
    co_await sw.connect(*serverAddress);
    async_task<int> cl = send(sw);
    async_task<int> rc = receive(sw);
    when_all wa({ &cl, &rc });
    co_await wa;
    co_await sw.disconnect();
    ioSvc.stop();
    co_return 0;
};
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

corolib (3/4): send

```
async_task<int> send(socket_wrapper& sw)
{
    std::uint8_t buffer[100];
    std::size_t totalBytesSent = 0;
    for (std::uint64_t i = 0; i < 1000; i += sizeof(buffer))
    {
        for (std::size_t j = 0; j < sizeof(buffer); ++j)
            buffer[j] = 'a' + ((i + j) % 26);
        std::size_t bytesSent = 0;
        do {
            bytesSent +=
                co_await sw.send(buffer + bytesSent, sizeof(buffer) - bytesSent);
            totalBytesSent += bytesSent;
        } while (bytesSent < sizeof(buffer));
    }
    co_return 0;
};
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

corolib (4/4): receive

```
async_task<int> receive(socket_wrapper& sw)
{
    std::uint8_t buffer[100];
    std::uint64_t totalBytesReceived = 0;
    std::size_t bytesReceived;
    do {
        bytesReceived = co_await sw.recv(buffer, sizeof(buffer));
        for (std::size_t i = 0; i < bytesReceived; ++i)
        {
            std::uint64_t byteIndex = totalBytesReceived + i;
            std::uint8_t expectedByte = 'a' + (byteIndex % 26);
            if (buffer[i] != expectedByte)
                print(PRI1, "buffer[i] != expectedByte\n");
        }
        totalBytesReceived += bytesReceived;
    }
    while (bytesReceived > 0 && totalBytesReceived < 1000);
    co_return 0;
}
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

Example run corolib (Windows 11)

Windows PowerShell

```
00: receive - after co_await sw.recv
00: receive - bytesReceived = 64
00: receive - totalBytesReceived = 704
00: receive - after co_await sw.recv
00: receive - bytesReceived = 64
00: receive - totalBytesReceived = 768
00: receive - after co_await sw.recv
00: receive - bytesReceived = 64
00: receive - totalBytesReceived = 832
00: receive - after co_await sw.recv
00: receive - bytesReceived = 64
00: receive - totalBytesReceived = 896
00: receive - after co_await sw.recv
00: receive - bytesReceived = 64
00: receive - totalBytesReceived = 960
00: receive - after co_await sw.recv
00: receive - bytesReceived = 40
00: receive - totalBytesReceived = 1000
00: receive - totalBytesReceived = 1000
00: receive - leaving
00: echoClient - after co_await wa
00: echoClient - after co_await sw.disconnect
00: echoClient - leaving
00: mainflow - after co_await echoClient
00: mainflow - entering
00: main - leaving
00: ---------------------------------------------------------------
00:     cons    dest    diff    max     c>p     c<p     c>h     c<h
00: ope 28      28      0       2
00: cor 31      31      0       5       0       31      31      0
00: pro 31      31      0       5       0       31
00: fin 31      31      0       3                       0       0
00: ---------------------------------------------------------------
00: Waiting 1000 milliseconds before exiting
PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\cppcoro\exam
```

Windows PowerShell

```
00: echoServer - bytesSent = 64
00: echoServer - after co_await sw.recv
00: echoServer - bytesReceived = 64
00: echoServer - after co_await sw.send
00: echoServer - bytesSent = 64
00: echoServer - after co_await sw.recv
00: echoServer - bytesReceived = 64
00: echoServer - after co_await sw.send
00: echoServer - bytesSent = 64
00: echoServer - after co_await sw.recv
00: echoServer - bytesReceived = 64
00: echoServer - after co_await sw.send
00: echoServer - bytesSent = 64
00: echoServer - after co_await sw.recv
00: echoServer - bytesReceived = 40
00: echoServer - after co_await sw.send
00: echoServer - bytesSent = 40
00: echoServer - after co_await sw.recv
00: echoServer - bytesReceived = 0
00: echoServer: totalBytesReceived = 1000, totalBytesSent = 1000
00: echoServer - after co_await sz.disconnect
00: echoServer - leaving
00: mainflow - after co_await echoServer
00: mainflow - entering
00: main - leaving
00: ---------------------------------------------------------------
00:     cons    dest    diff    max     c>p     c<p     c>h     c<h
00: ope 36      36      0       2
00: cor 37      37      0       3       0       37      37      0
00: pro 37      37      0       3       0       37
00: fin 37      37      0       1                       0       0
00: ---------------------------------------------------------------
00: Waiting 1000 milliseconds before exiting
PS C:\X\workspace4l\corolib-master\out\build\x64-Debug\examples\cppcoro\exa
mples-cl>
```


Public © Capgemini 2025. All rights reserved |

USING COROUTINES WITH WINDOWS OVERLAPPED I/O

Source code

cppcoro: echo_client2.cpp

corolib: echo_client2.cpp

Public © Capgemini 2025. All rights reserved |

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Using a single coroutine library with several asynchronous communication frameworks
    1. Boost ASIO
    2. Qt5
    3. gRPC
    4. Windows Overlapped I/O (based upon cppcoro)
    5. **Other (not handled in this presentation)**
4. Summary and conclusions
5. Brief introduction to corolib
6. Other coroutine libraries

Public © Capgemini 2025. All rights reserved

USING COROUTINES WITH …

Other (a)synchronous communication frameworks that are used with corolib

- TAO (The ACE (ADAPTIVE (A Dynamically Assembled Protocol Transformation, Integration and eValuation Environment)) ORB (Object Request Broker))
  - I used TAO for an ORB comparison study in 2008 in cooperation with PrismTech.
  - Note: PrismTech is now part of ADLINK Technology.
- ROS 2 (Robot Operating System 2)
  - I used ROS (1) Melodic for an internal Altran project (Robotic Arm, Q4 2020).
  - ROS 2 uses DDS as communication infrastructure.
- libevent
  - A previous client uses libevent for communication between local applications and gRPC for communication with external applications.
- curl
  - Inspired upon https://www.youtube.com/watch?v=78nwm9EP23A
- Please see subdirectories under examples for all source code.

Public © Capgemini 2025. All rights reserved

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Using a single coroutine library with several asynchronous communication frameworks
4. **Summary and conclusions (5 slides)**
5. Brief introduction to corolib

Public © Capgemini 2025. All rights reserved

SUMMARY AND CONCLUSIONS

Summary and conclusions (1/5)

- Synchronous communication frameworks (SCFs)
  - Synchronous API: e.g., connect, open, write, read, …
  - SCFs allow developing distributed applications that are easy to read, write and maintain
  - … but that are not reactive (i.e., they may not be able to act immediately upon new inputs)
  - … unless blocking operations are run on dedicated user threads (e.g., using std::async).
- Asynchronous communication frameworks (ACFs)
  - Asynchronous API: e.g., async_connect, async_open, async_write, async_read … + completion handlers
  - ACFs allow restoring the reactiveness
  - … at the expense of chopping up the application into small non-blocking but interdependent pieces in case of a single-threaded framework: the implementation of completion handler N starts action N+1

Public © Capgemini 2025. All rights reserved

SUMMARY AND CONCLUSIONS

Summary and conclusions (2/5)

- In an ACF, the application starts a remote operation but does not wait for the result of the operation.
- The application registers a completion handler with the ACF on starting the operation.
- When the result arrives, the ACF calls this completion handler …
  - on a dedicated thread, asynchronously with the launching thread
  - or on the same thread, from an event loop where the application must always return to.
- The ACFs using a dedicated completion handler thread do not need/have an event loop.
  - However, the programmer must foresee a mechanism for the main thread not to fall off the program.
- Notice that there are other coding styles for ACFs, sometimes without any “visible” completion handlers (e.g., state machines).

Public © Capgemini 2025. All rights reserved

SUMMARY AND CONCLUSIONS

Summary and conclusions (3/5)

How to use/restore a *synchronous programming style* using a…

- Single-threaded ACF
  - The framework calls the completion handler from an event loop.
  - The application returns to this event loop after calling an asynchronous function.
  - Examples: Boost ASIO, Qt, Windows Overlapped I/O, TAO, …
  - Solution 1: Use coroutines!
  - Solution 2: Run the event loop on a separate thread to make it multi-threaded (if the framework is thread-safe…).
- Multi-threaded ACF
  - The framework calls the completion handler from a dedicated thread.
  - The framework does not (necessarily) provide an event loop by itself.
  - Examples: gRPC, …
  - Solution 1: Use coroutines with an event queue and loop, although…
  - Solution 2: There is no need to use coroutines: use a condition variable / semaphore / latch instead. Note: this introduces local waiting points and makes the application less reactive unless the waiting function runs on its own thread.

Public © Capgemini 2025. All rights reserved

SUMMARY AND CONCLUSIONS

Summary and conclusions (4/5)

In a table:

Can we use a synchronous (sequential) style using an ACF?	Completion handler runs on application thread	Completion handler runs on a dedicated thread
Not using coroutines	No	Yes (use CV or semaphore)
Using coroutines	Yes	Yes (but not really needed: use CV or semaphore instead)

- To use a synchronous programming style in case the completion handler runs on the same main thread (and the application returns to an event loop), we must use coroutines.
- To ensure reactiveness in case the completion handler runs on a dedicated thread and a condition variable or semaphore is used instead of coroutines, the awaiting function must run on a separate thread as well.

Public © Capgemini 2025. All rights reserved

SUMMARY AND CONCLUSIONS

Summary and conclusions (5/5)

- A single coroutine library can be used with several asynchronous communication frameworks (ACFs).
- Coroutines can be used to write distributed applications using a synchronous programming style yet executing/behaving in an efficient reactive asynchronous way.
  - Coroutines offer the advantages of both styles while not introducing any new major disadvantages.
  - Synchronous style: easy to read, write and maintain, but not reactive.
  - Asynchronous style: reactive, at the expensive of having to split the code into small, non-blocking fragments.
- If an ACF runs the completion handler on a dedicated thread, it is possible to maintain a synchronous style using synchronization mechanisms such as condition variables, (binary) semaphores or latches.
  - However, this introduces internal waiting points.
- If an ACF does *not* run the completion handler on a dedicated thread (single-threaded framework), then the use of coroutines is inevitable to restore a synchronous style and reactiveness.

Public © Capgemini 2025. All rights reserved

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Using a single coroutine library with several asynchronous communication frameworks
4. Summary and conclusions
5. **Brief introduction to corolib (14 slides)**
6. Other coroutine libraries

Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

Introduction

- Hobby project
  - Developed in my free time or between client projects.
  - Started on June 1, 2019.
  - First presentation: Belgian C++ Users Group, 29 January 2020: Slides + code samples: http://becpp.org/blog/wp-content/uploads/2020/02/Johan-Vanslembrouck-Coroutines-in-C20.zip
  - First commit to GitHub in June 2020 using corolib as name: https://github.com/JohanVanslembrouck/corolib
- 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:
  - To be able to use both classes in a variety of contexts, they have become heavy (in terms of data members).
  - Some data members 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.

Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

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

client application | server application | ... | async I/O application — application layer (OS independent)

C++ coroutine library — coroutine library layer

Boost ASIO wrapper | Qt wrapper | gRPC wrapper | libevent wrapper | ... | prop. I/O wrapper — wrapper / adaptation layer

Boost ASIO | Qt | gRPC | libevent | ... | proprietary async I/O library — asynchronous communication middleware layer (available on one or multiple OSs)

Linux | macOS | Windows | bare metal — operating systems layer (including communication stacks)

Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

The start_<operation>, start_<operation>_impl pattern

```
async_operation<void> Timer01::start_timer(steady_timer& timer, int ms)
{
    int index = get_free_index();
    async_operation<void> ret{ this, index };
    start_timer_impl(index, timer, ms);
    return ret;
}
void Timer01::start_timer_impl(const int idx, steady_timer& tmr, int ms)
{
    tmr.expires_after(std::chrono::milliseconds(ms));
    tmr.async_wait(
        [this, idx, ms](const boost::system::error_code& error) {
            if (!error)
                completionHandler_v(idx);
        });
}

// Usage:
async_operation<void> op_timer1a = start_timer(timer1, 1000);
co_await op_timer1a;
// or
co_await start_timer(timer1, 1000);
```


Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

The start_<operation>, start_<operation>_impl pattern

- The original idea was to pass the address of the async_operation ret object declared in the start() function to the start_impl() function…
- … relying on return value optimization (RVO) (copy elision) for passing the final address of the object to the start_impl() function.
- Unfortunately, that did not work (in 2019 with Visual Studio 2019).
- The use of RVO depends on the compiler (version) and even on some of the options used (such as debug mode, optimization level).
- An alternative mechanism was implemented in August 2019, using an index into an array of pointers to async_operation objects (in fact using a base class of async_operation).

Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

The start_<operation>, start_<operation>_impl pattern

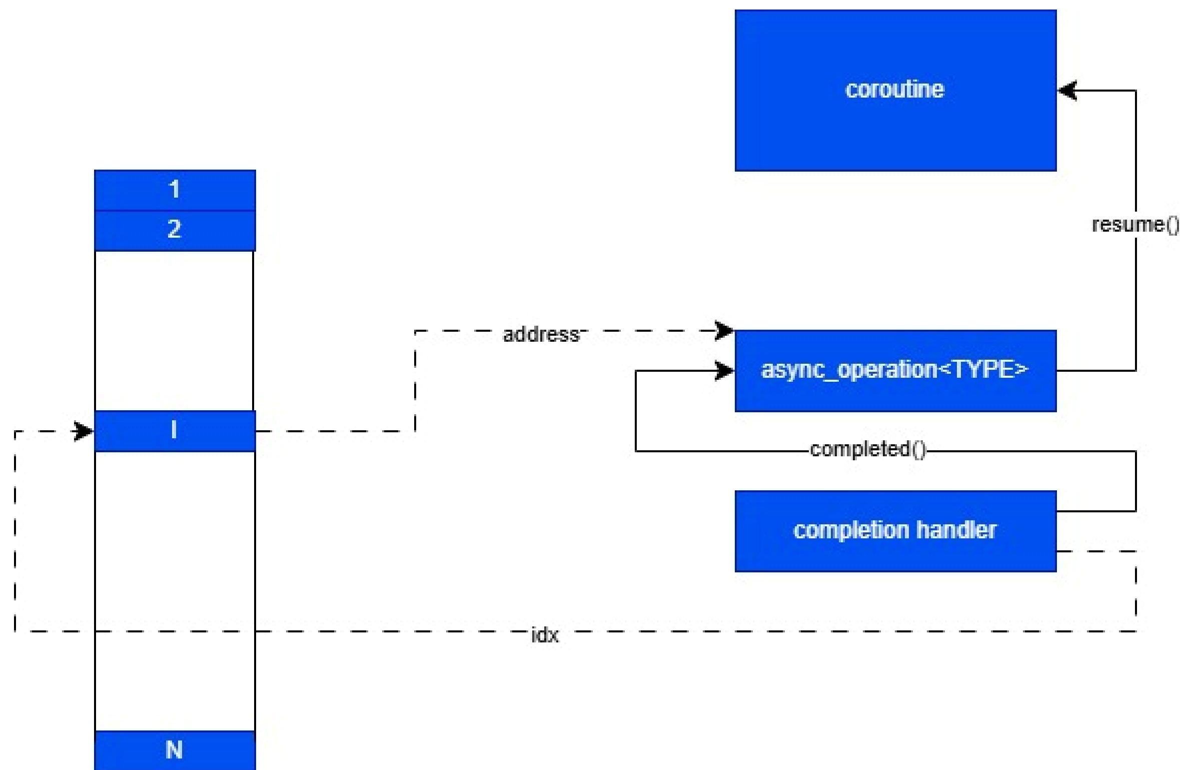
- First, we search a free index in the array using get_free_index().
- This index is passed to the async_operation constructor.
- The constructor places its this-address at the index in the array, the destructor removes the address from the array.
- The move constructor and assignment operator will replace the old address with the new address when an object is moved.
- Copy constructor and assignment operator are deleted.

```
async_operation<void> Timer01::start_timer(steady_timer& timer, int ms)
{
    int index = get_free_index();
    async_operation<void> ret{ this, index };
    start_timer_impl(index, timer, ms);
    return ret;
}
```


Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

Using an array to store async_operation addresses



- The completion handler (a lambda) is passed an index (idx) into an array of pointers to async_operation (base class) objects in its capture list.
- The asynchronous communication framework calls the completion handler when the asynchronous I/O function or RMI has completed.
- After having retrieved the address of the async_operation object in the array, the completion handler calls the completed() member function on the found object.
- The completed() function calls resume() on the coroutine_handle that refers to the coroutine in which async_operation is located and that is used to co_await the asynchronous operation.

Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

The start_<operation>, start_<operation>_impl pattern

- The async_operation object must be created *before* the call of the start_impl function and not at the return statement.
- In case of immediate completion, the completion handler needs the address of the async_operation object to write the reply, even if the object will be moved to its final location at the return in the start function.
- In other words, don’t use the following code:

```
async_operation<void> Timer01::start_timer(steady_timer& timer, int ms)
{
    int index = get_free_index();
    start_timer_impl(index, timer, ms);
    return { this, index };
}
```


Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

The start_<operation>, start_<operation>_impl pattern

- In case the async_operation object goes out-of-scope before the operation completes, the completion handler will find a null pointer at the index, indicating it cannot write the response to the async_operation object (that does not exist anymore) and it cannot resume the coroutine in which the async_operation object was declared.
- To deal with the possibility that another async_operation object has already taken the entry at that index, the array has been extended to save a timestamp (of creation) together with the address at that entry.
- The completion handler will now test if the timestamp it was passed in its capture list is the same as the timestamp that it finds at the entry at the index.
- If this is not the case, then its entry was taken by another object: same action as if there is a nullptr at the entry.
- See code on next slide.

Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

The start_<operation>, start_<operation>_impl pattern

- Using a timestamp

```
async_operation<void> CommCore::start_timer(steady_timer& timer, int ms)
{
    int index = get_free_index_ts();
    async_operation<void> ret{ this, index, true };
    start_timer_impl(index, timer, ms);
    return ret;
}

void CommCore::start_timer_impl(const int idx, steady_timer& tmr, int ms)
{
    tmr.expires_after(std::chrono::milliseconds(ms));
    std::chrono::high_resolution_clock::time_point now = get_async_operation_info(idx)->start;
    tmr.async_wait(
        [this, idx, now](const boost::system::error_code& error) {
            if (!error)
                completionHandler_ts_v(idx, now);
        });
}
```


Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

The start_<operation>, start_<operation>_impl pattern

- It is possible to insert the content of the start_impl() function in the start() function (see code below).
- However, for modularity reasons and because the start_impl() function does call a communication framework function, while the start() function does not, this separation allows placing both functions in different header or source files, where only the file containing the start_impl() function depends on the communication framework.

```
async_operation<void> Timer01::start_timer(steady_timer& timer, int
ms)
{
    int index = get_free_index();
    async_operation<void> ret{ this, index };
    timer.expires_after(std::chrono::milliseconds(ms));
    timer.async_wait(
        [this, index, ms](const boost::system::error_code& error) {
            if (!error)
                completionHandler_v(index);
        });
    return ret;
}
```


Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

The start_<operation>, start_<operation>_impl pattern

- Why not using the following code (see code fragment below)?
- Although it simplifies the implementation, it looks rather clumsy at the application level.
- Practical use: if we want to re-use the same async_operation object.

```
void Timer02::start_timer(async_operation_base& async_op, steady_timer& tmr, int ms)
{
    async_operation_base* p_async_op = &async_op;
    tmr.expires_after(std::chrono::milliseconds(ms));
    tmr.async_wait(
        [this, p_async_op, ms](const boost::system::error_code& error) {
            if (!error)
                completionHandler_v(p_async_op);
        });
}
// Usage:
async_operation<void> op_timer1{ this };
op_timer1.auto_reset(true);
start_timer(op_timer1, timer1, 1000);
co_await op_timer1;
start_timer(op_timer1, timer1, 2000);
co_await op_timer1;
```


Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

Source code

- completionHandler_v() + completionHandler_ts_v() : commservice.h
- start_timer() + start_timer_impl(): commcore.cpp
- Timer01: timer01.cpp
- Timer02: timer02.cpp

Public © Capgemini 2025. All rights reserved

BRIEF INTRODUCTION TO COROLIB

Directory structure (1/2)

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


Public © Capgemini 2025. All rights reserved

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
│   ├── corba
│   ├── corolab (contains the (somewhat modified) code of the 2020 presentation)
│   ├── final_suspend (contains the full source code used in this presentation)
│   ├── initial_suspend (contains the full source code used in this presentation)
│   ├── rvo
│   ├── transform
│   ├── why-coroutines
│   └── why-coroutines2
└── tests (uses GoogleTest)
```


Public © Capgemini 2025. All rights reserved

AGENDA

1. Brief introduction to C++ coroutines
2. Brief introduction to (a)synchronous distributed programming
3. Using a single coroutine library with several asynchronous communication frameworks
4. Summary and conclusions
5. Brief introduction to corolib (14 slides)
6. **Other coroutine libraries**

Public © Capgemini 2025. All rights reserved

OTHER COROUTINE LIBRARIES

(and their use with ACFs)

- cppcoro: https://github.com/lewissbaker/cppcoro, https://github.com/andreasbuhr/cppcoro
  - Used with Windows Overlapped I/O
- Boost ASIO
- cobalt: https://github.com/boostorg/cobalt
  - Used with Boost ASIO
- qcoro: https://github.com/qcoro/qcoro
  - Used with Qt
- folly: https://github.com/facebook/folly
- concurrencpp: https://github.com/David-Haim/concurrencpp
- libcoro: https://github.com/jbaldwin/libcoro
- co_curl: https://github.com/hanickadot/co_curl

Public © Capgemini 2025. All rights reserved

Thank you!

Public © Capgemini 2025. All rights reserved