Cancellations in Asio

A tale of coroutines and timeouts

Rubén Pérez Hidalgo <rubenperez038@gmail.com>
The C++ Alliance

using std::cpp 2025

Schedule

Coroutines

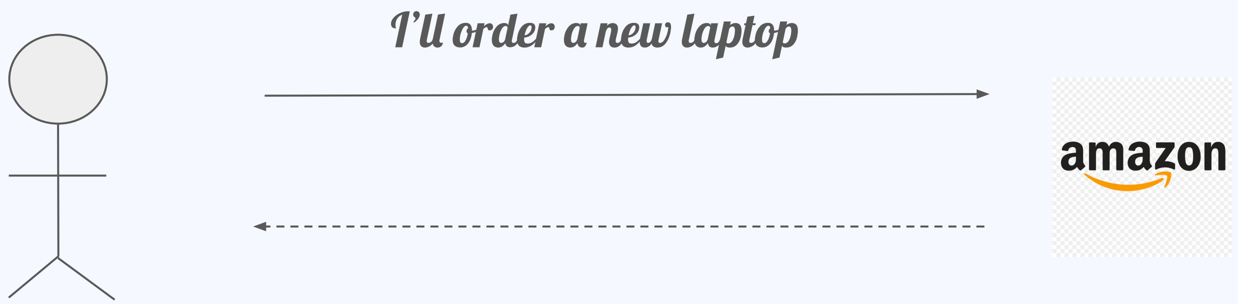
Timeouts

Cancellation

Fear not!

```
co_await asio::async_write(sock, bytes, asio::cancel_after(60s));
```

Sync vs. async programming



Sync vs. async programming

- ✓ Async has **higher throughput** under heavy loads
- ✓ In Asio, async offers **unique functionality**

```
asio::write(sock, bytes);   // how do I set a timeout to this?
```

Asio: Boost and standalone

Platform-independent

async networking

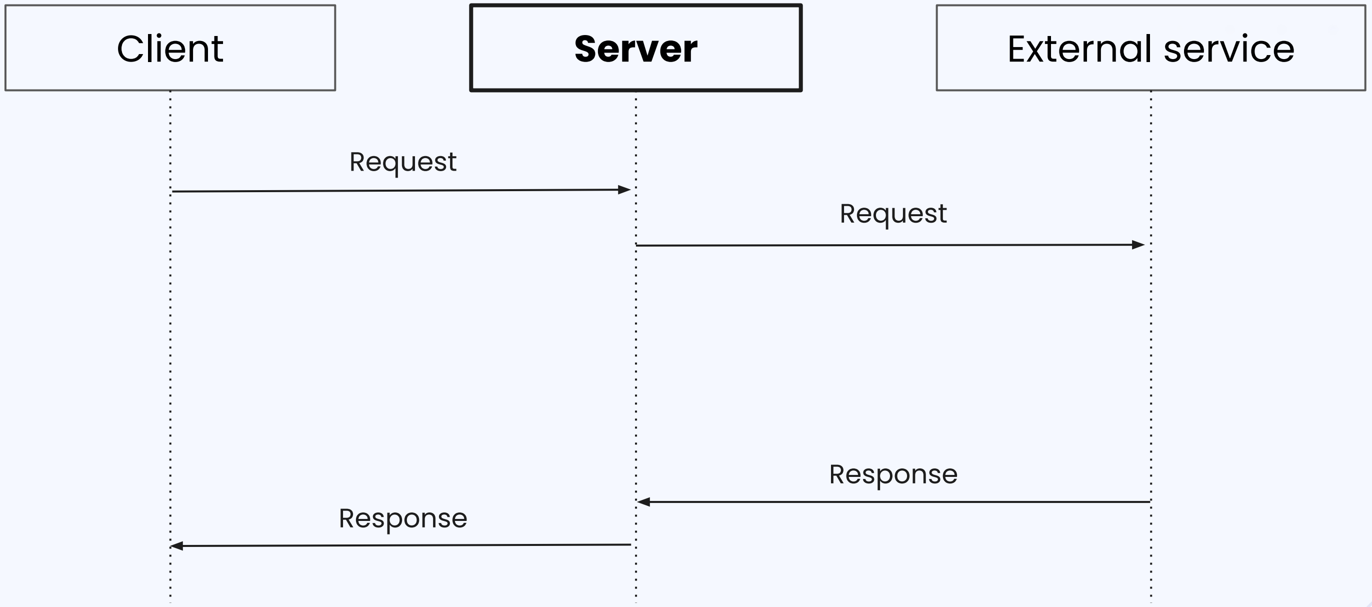
" Powerful but complex "

https://github.com/chriskohlhoff/asio/

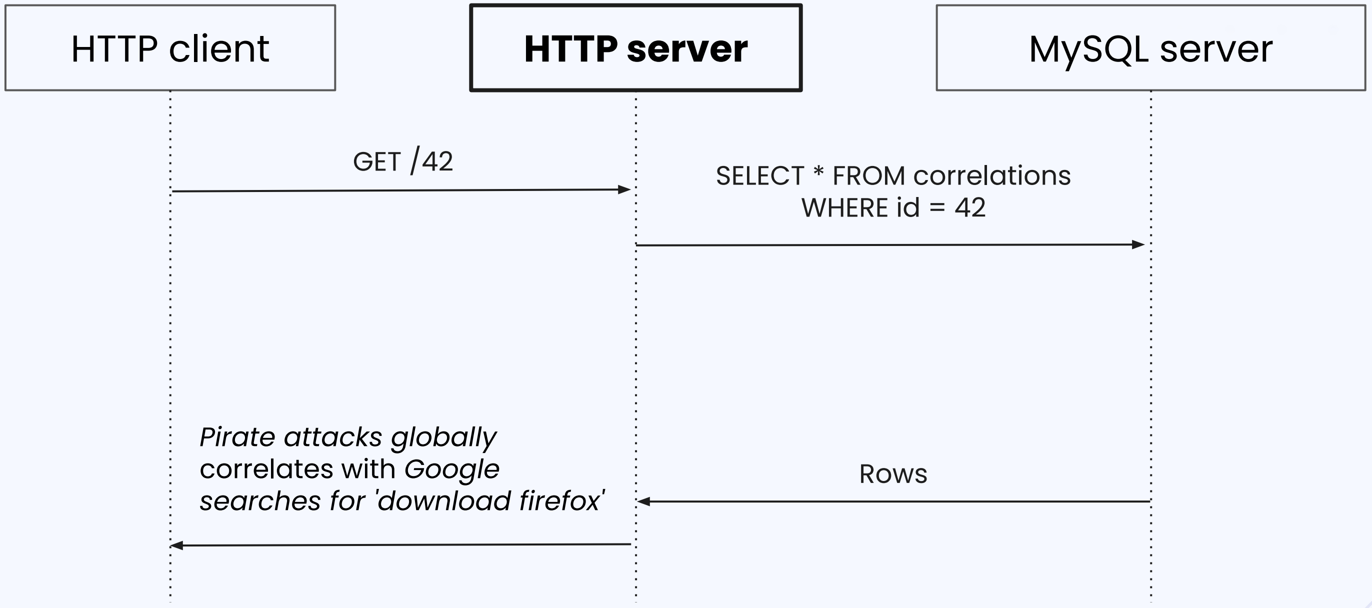
https://github.com/boostorg/asio/

- ✓ Boost.Beast (HTTP and websocket)
- ✓ Boost.MySQL
- ✓ Boost.Redis
- ✓ Boost.MQTT5

The example


Client
Server
External service
Request
Request
Response
Response

The example


HTTP client
HTTP server
MySQL server
GET /42
SELECT * FROM correlations WHERE id = 42
Rows
Pirate attacks globally correlates with *Google searches for 'download firefox'*

Pseudocode

```
tcp_acceptor acceptor;

while (true) {
   connection conn = acceptor.wait_for_client_connection();

   as_background_task {
       request req = conn.read_request();
       response res = query_database(req);
       conn.write_response(res);
   }
}
```

(*) Not handling HTTP keep-alive

Sync (1)

```cpp
int main()
{
    asio::io_context ctx;

    // Set up an object listening for TCP connections in port 8080
    asio::ip::tcp::acceptor acceptor(ctx);
    acceptor.open(asio::ip::tcp::v4());
    acceptor.set_option(asio::socket_base::reuse_address(true));
    acceptor.bind({asio::ip::make_address("0.0.0.0"), 8080});
    acceptor.listen();

    while (true)
    {
        // Accept a connection
        asio::ip::tcp::socket sock = acceptor.accept();

        // Launch a session.
        run_session(sock);
    }
}
```

Execution context

- ✓ Handler queue
- ✓ Timer queue
- ✓ Epoll reactor
- ✓ …

I/O object

I/O operation

```cpp
namespace asio = boost::asio;
namespace beast = boost::beast;
namespace http = boost::beast::http;
namespace mysql = boost::mysql;
```

Sync (2)

```
     void run_session(asio::ip::tcp::socket& sock)
     {
         // Read a request
         beast::flat_buffer buff;
         http::request<http::empty_body> req;
---►     http::read(sock, buff, req);
         std::uint64_t id = parse_id(req.target()); // Given "/42", get the number 42

         // Query the database
         mysql::any_connection conn(sock.get_executor());
---►     conn.connect({.username = "me", .password = "secret", .database = "correlations"});
         mysql::results r;
---►     conn.execute(mysql::with_params("SELECT subject FROM correlations WHERE id = {}", id), r);

         // Compose the response
         http::response<http::string_body> res;
         if (r.rows().empty())
             res.result(http::status::not_found);
         else
             res.body() = r.rows().at(0).at(0).as_string();

         // Write the response back
         res.version(req.version());
         res.keep_alive(false);
         res.prepare_payload();
---►     http::write(sock, res);
     }
```

Composed I/O operation

Executor

- ✓ Pointer to execution context
- ✓ Customization point

Sync (1 revisited)

```cpp
int main()
{
    asio::io_context ctx;

    // Set up an object listening for TCP connections in port 8080
    asio::ip::tcp::acceptor acceptor(ctx);
    acceptor.open(asio::ip::tcp::v4());
    acceptor.set_option(asio::socket_base::reuse_address(true));
    acceptor.bind({asio::ip::make_address("0.0.0.0"), 8080});
    acceptor.listen();

    while (true)
    {
        // Accept a connection
        asio::ip::tcp::socket sock = acceptor.accept();

        // Launch a session.
        run_session(sock);
    }
}
```

Execution context

- ✓ Handler queue
- ✓ Timer queue
- ✓ Epoll reactor
- ✓ …

I/O object

I/O operation

Async (1)

```cpp
asio::awaitable<void> run_session(asio::ip::tcp::socket& sock)
{
    // Read a request
    beast::flat_buffer buff;
    http::request<http::empty_body> req;
    co_await http::async_read(sock, buff, req);
    std::uint64_t id = parse_id(req.target());

    // Query the database
    mysql::any_connection conn(sock.get_executor());
    co_await conn.async_connect({.username = "me",
          .password = "secret", .database = "correlations"});

    // ...
}
```

```cpp
void run_session(asio::ip::tcp::socket& sock)
{
    // Read a request
    beast::flat_buffer buff;
    http::request<http::empty_body> req;
    http::read(sock, buff, req);
    std::uint64_t id = parse_id(req.target());

    // Query the database
    mysql::any_connection conn(sock.get_executor());
    conn.connect({.username = "me",
        .password = "secret", .database = "correlations"});

    // ...
}
```

A sync function can be trivially converted into a coroutine by applying a set of transformations

Async (2)

```cpp
asio::awaitable<void> run_server()
{
   // Set up an object listening for TCP connections in port 8080
   asio::ip::tcp::acceptor acceptor(co_await asio::this_coro::executor);
   acceptor.open(asio::ip::tcp::v4());
   acceptor.set_option(asio::socket_base::reuse_address(true));
   acceptor.bind({asio::ip::make_address("0.0.0.0"), 8080});
   acceptor.listen();

   // Accept connections in a loop
   while (true)
   {
       // Accept a connection
       asio::ip::tcp::socket sock = co_await acceptor.async_accept();

       // Launch a session.
       co_await run_session(sock);
   }
}
```

```cpp
int main()
{
   asio::io_context ctx;

   asio::co_spawn(
       // Spawn a coroutine using this execution context
       ctx,

       // The actual code to run, as an awaitable
       run_server(),

       // When the coroutine finishes, run this callback.
       // Propagate exceptions thrown in the coroutine.
       [](std::exception_ptr exc) {
           if (exc)
               std::rethrow_exception(exc);
       }
   );

   ctx.run();
}
```

Async (3)

```cpp
asio::awaitable<void> run_server()
{
   // Set up an object listening for TCP connections in port 8080
   asio::ip::tcp::acceptor acceptor(co_await asio::this_coro::executor);
   acceptor.open(asio::ip::tcp::v4());
   acceptor.set_option(asio::socket_base::reuse_address(true));
   acceptor.bind({asio::ip::make_address("0.0.0.0"), 8080});
   acceptor.listen();

   // Accept connections in a loop
   while (true)
   {
       // Accept a connection
       asio::ip::tcp::socket sock = co_await acceptor.async_accept();

       // Launch a session, but don't wait for it
       asio::co_spawn(
           co_await asio::this_coro::executor,  // Use the same executor as this coroutine
           run_session(std::move(sock)),        // The coroutine to run, as an awaitable
           [](std::exception_ptr exc) {         // If an exception is thrown, log it
               if (exc) log_error(exc);
           }
       );
   }
}
```

Timeouts (1)

```
asio::awaitable<void> run_session(asio::ip::tcp::socket sock)
{
   // Read a request
   beast::flat_buffer buff;
   http::request<http::empty_body> req;
   co_await http::async_read(sock, buff, req);
   std::uint64_t id = parse_id(req.target());

   // Query the database
   mysql::any_connection conn(sock.get_executor());
   co_await conn.async_connect(
       {.username = "me", .password = "secret", .database = "correlations"},
   );

   // ...
}
```

Timeouts (2)

```
asio::awaitable<void> run_session(asio::ip::tcp::socket sock)
{
   using namespace std::chrono_literals;

   // Read a request
   beast::flat_buffer buff;
   http::request<http::empty_body> req;
   co_await http::async_read(sock, buff, req, asio::cancel_after(30s));
   std::uint64_t id = parse_id(req.target());

   // Query the database
   mysql::any_connection conn(sock.get_executor());
   co_await conn.async_connect(
       {.username = "me", .password = "secret", .database = "correlations"},
       asio::cancel_after(30s)
   );

   // ...
}
```

Completion token

Timeout => Operation fails with `asio::error::operation_aborted`

For the coroutine, a network error

(*) `beast::tcp::stream` also supports built-in timeout functionality.

Completion tokens

```
template <
   class Body,
   class CompletionToken = asio::deferred_t>
auto async_write(
   asio::ip::tcp::socket& sock,
   const http::response<Body>& msg,
   CompletionToken&& token = asio::deferred
);*
```

Initiating function

* Simplified for exposition purposes.
The actual definition is more generic.

Default completion token

Return something we can `co_await`

The return type changes with `CompletionToken`

```
co_await http::async_write(sock, res);
co_await http::async_write(sock, res, asio::deferred);
co_await http::async_write(sock, res, asio::cancel_after(30s));

http::async_write(sock, res, [](error_code ec, std::size_t bytes_written) {
    // ...
});
std::future<std::size_t> future = http::async_write(sock, res, asio::use_future);
```

Complies with Asio's **universal async model**

Adapter tokens

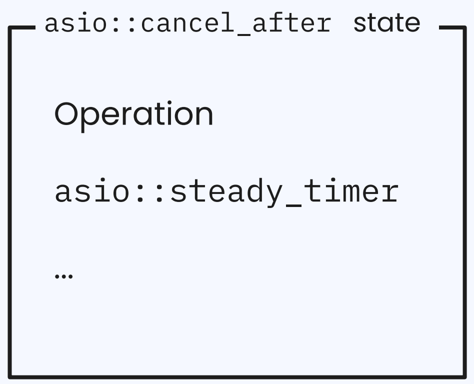
Wrap other completion tokens

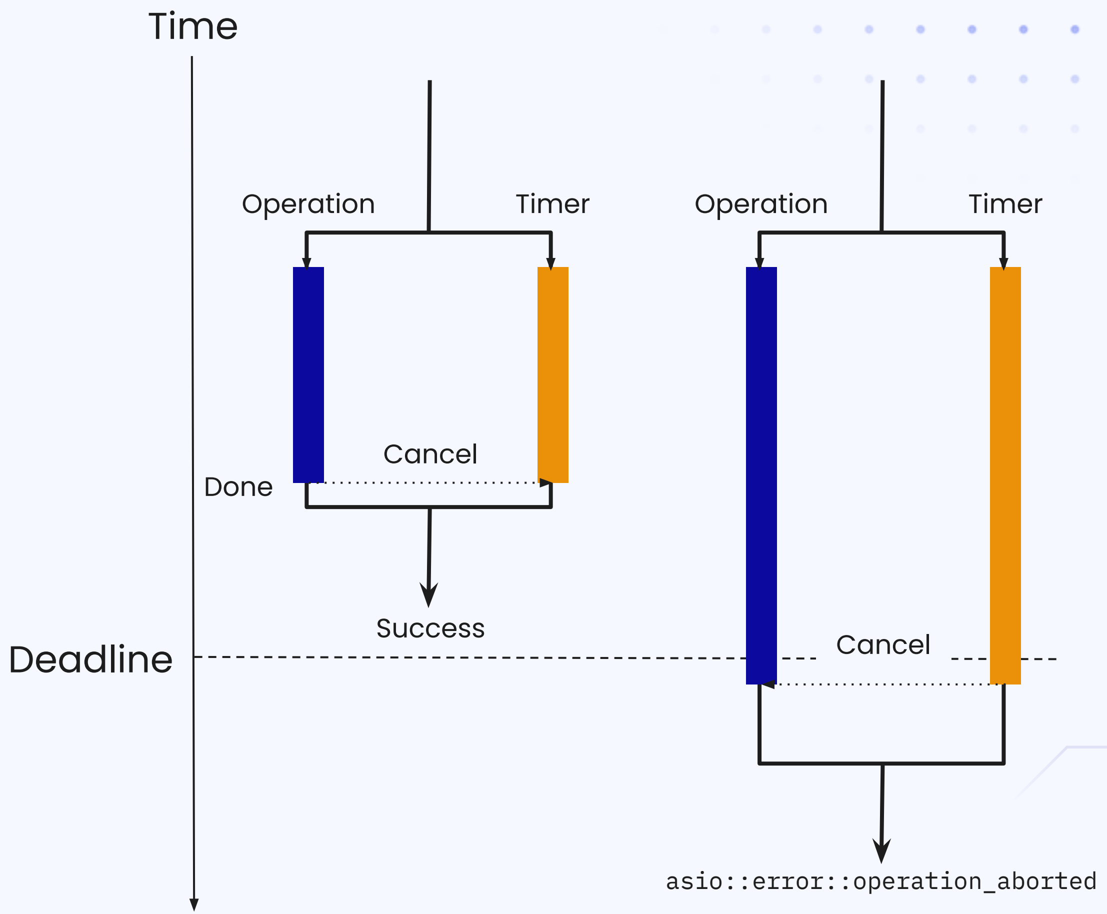
```
co_await http::async_write(sock, res, asio::cancel_after(30s));

co_await http::async_write(sock, res, asio::cancel_after(30s, asio::deferred));


http::async_write(sock, res, asio::cancel_after(30s, [](error_code ec, std::size_t bytes_written) {
        // ...
}));
```

Cancellation





Cancellation

Object-wide cancellation

I/O object specific

Targets all operations

```
sock.cancel();
```

Per-operation cancellation

Works for any operation, including composed ones

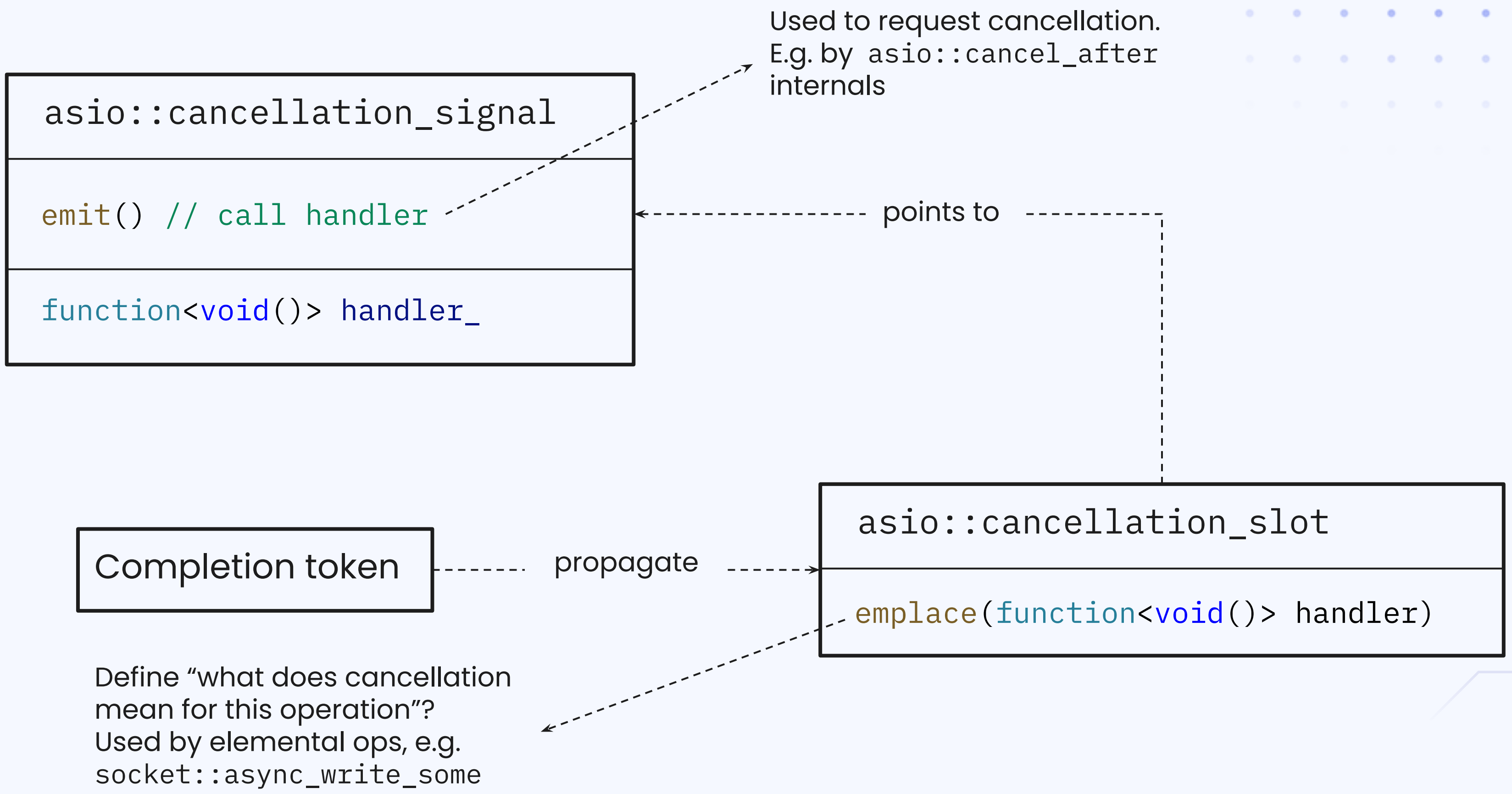
Targets a specific operation

Signal/slot mechanism

```
asio::cancel_after(30s)
```

Only async operations can be cancelled!

Cancellation

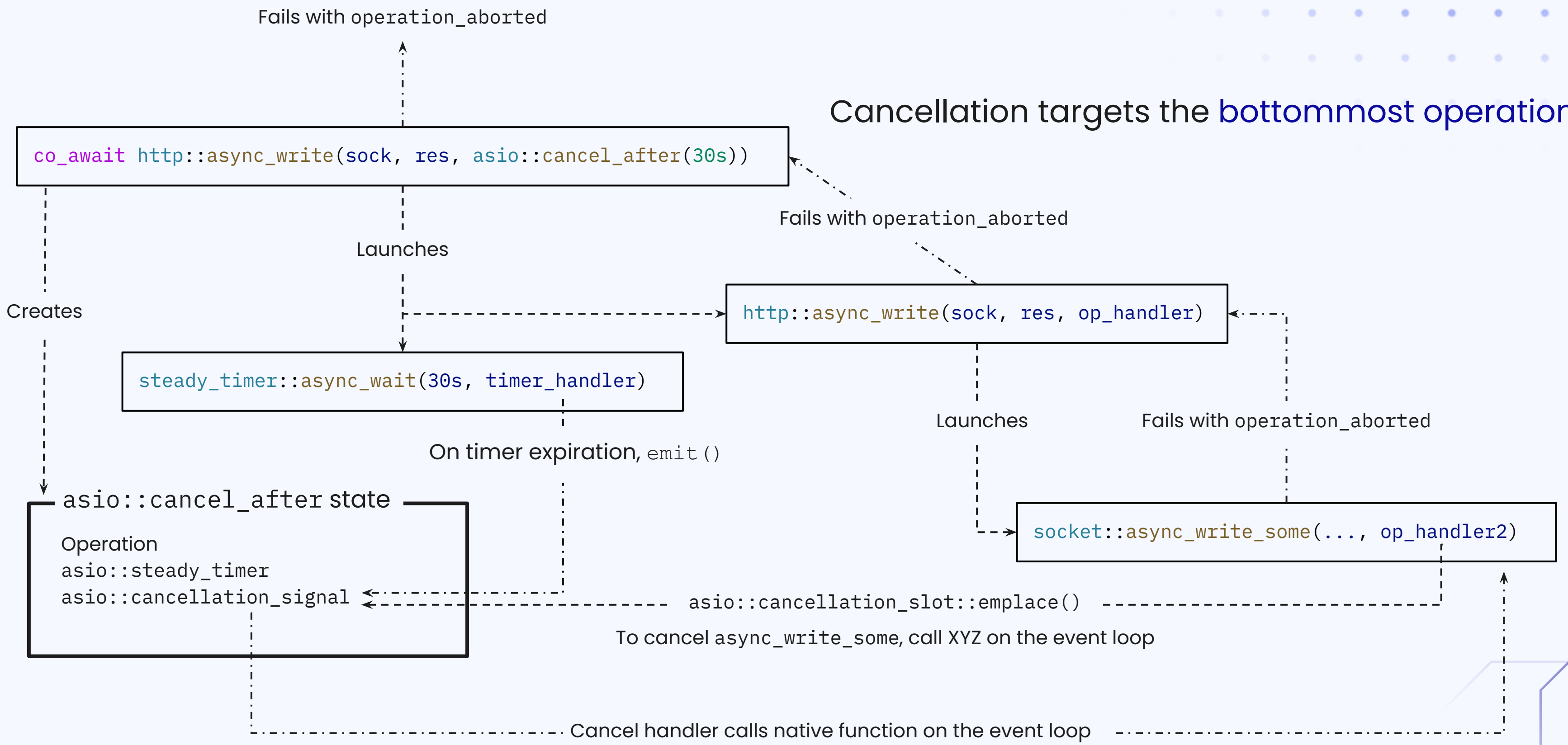


asio::cancellation_signal
emit() // call handler
function<void()> handler_
Used to request cancellation.
E.g. by asio::cancel_after
internals
points to
asio::cancellation_slot
emplace(function<void()> handler)
Completion token
propagate
Define "what does cancellation
mean for this operation"?
Used by elemental ops, e.g.
socket::async_write_some

(*) Signatures simplified for exposition purposes.

Cancellation

Cancellation targets the bottommost operation



Fails with operation_aborted
co_await http::async_write(sock, res, asio::cancel_after(30s))
Creates
Launches
Fails with operation_aborted
http::async_write(sock, res, op_handler)
steady_timer::async_wait(30s, timer_handler)
On timer expiration, emit()
Launches
Fails with operation_aborted
asio::cancel_after state
Operation
asio::steady_timer
asio::cancellation_signal
socket::async_write_some(..., op_handler2)
asio::cancellation_slot::emplace()
To cancel async_write_some, call XYZ on the event loop
Cancel handler calls native function on the event loop

(*) Proxy signals/slots omitted to simplify.

co_spawn timeout

```
asio::awaitable<void> run_session(asio::ip::tcp::socket& sock)
{
   // Read a request
   beast::flat_buffer buff;
   http::request<http::empty_body> req;
   co_await http::async_read(sock, buff, req, asio::cancel_after(30s));
   std::uint64_t id = parse_id(req.target());

   // Query the database
   mysql::any_connection conn(sock.get_executor());
   co_await conn.async_connect({.username = "me", .password = "secret", .database = "correlations"}, asio::cancel_after(30s));
   mysql::results r;
   co_await conn.async_execute(mysql::with_params("SELECT subject FROM correlations WHERE id = {}", id), r, asio::cancel_after(30s));

   // Compose the response
   http::response<http::string_body> res;
   if (r.rows().empty())
       res.result(http::status::not_found);
   else
       res.body() = r.rows().at(0).at(0).as_string();

   // Write the response back
   res.version(req.version());
   res.keep_alive(false);
   res.prepare_payload();
   co_await http::async_write(sock, res, asio::cancel_after(30s));
}
```

co_spawn timeout

```cpp
asio::awaitable<void> run_session(asio::ip::tcp::socket& sock)
{
   // Read a request
   beast::flat_buffer buff;
   http::request<http::empty_body> req;
   co_await http::async_read(sock, buff, req, asio::cancel_after(30s));
   std::uint64_t id = parse_id(req.target());

   // Query the database
   mysql::any_connection conn(sock.get_executor());
   co_await conn.async_connect({.username = "me", .password = "secret", .database = "correlations"});
   mysql::results r;
   co_await conn.async_execute(mysql::with_params("SELECT subject FROM correlations WHERE id = {}", id), r);

   // Compose the response
   http::response<http::string_body> res;
   if (r.rows().empty())
       res.result(http::status::not_found);
   else
       res.body() = r.rows().at(0).at(0).as_string();

   // Write the response back
   res.version(req.version());
   res.keep_alive(false);
   res.prepare_payload();
   co_await http::async_write(sock, res, asio::cancel_after(30s));
}
```

```cpp
asio::awaitable<http::response<http::string_body>>
handle_request(std::uint64_t id);
```

co_spawn timeout

```cpp
asio::awaitable<void> run_session(asio::ip::tcp::socket& sock)
{
    // Read a request
    beast::flat_buffer buff;
    http::request<http::empty_body> req;
    co_await http::async_read(sock, buff, req);
    std::uint64_t id = parse_id(req.target());

    // Query the database
    // TODO: can we set a timeout to handle_request, as a whole?
    http::response<http::string_body> res = co_await handle_request(id);

    // Write the response back
    res.version(req.version());
    res.keep_alive(false);
    res.prepare_payload();
    co_await http::async_write(sock, res, asio::cancel_after(30s));
}
```

co_spawn timeout

```
asio::awaitable<void> run_session(asio::ip::tcp::socket& sock)
{
   // Read a request
   beast::flat_buffer buff;
   http::request<http::empty_body> req;
   co_await http::async_read(sock, buff, req);
   std::uint64_t id = parse_id(req.target());

   // Query the database
   http::response<http::string_body> res = co_await asio::co_spawn(
       co_await asio::this_coro::executor,
       handle_request(id),
       asio::cancel_after(30s)
   );

   // Write the response back
   res.version(req.version());
   res.keep_alive(false);
   res.prepare_payload();
   co_await http::async_write(sock, res, asio::cancel_after(30s));
}
```

```
asio::co_spawn(
    ctx,
    run_server(),
    [](std::exception_ptr exc) {
        if (exc) std::rethrow_exception(exc);
    }
);
```

Thanks!

Do you have any questions?

https://github.com/anarthal/usingstdcpp-2025

https://github.com/anarthal/servertech-chat

Rubén Pérez Hidalgo
The C++ Alliance

Avoiding exceptions

```
auto [ec, bytes_written] = co_await http::async_write(
         sock, res, asio::as_tuple);

auto [ec, bytes_written] = co_await http::async_write(
         sock, res, asio::cancel_after(30s, asio::as_tuple));
```

Cancellation types

Guarantees after cancellation

Terminal

`cancellation_type_t::terminal`

I/O object left in undefined state
Close or destroy

`http::async_write()`

Partial

`cancellation_type_t::partial`

Operation may have side-effects observable through its API

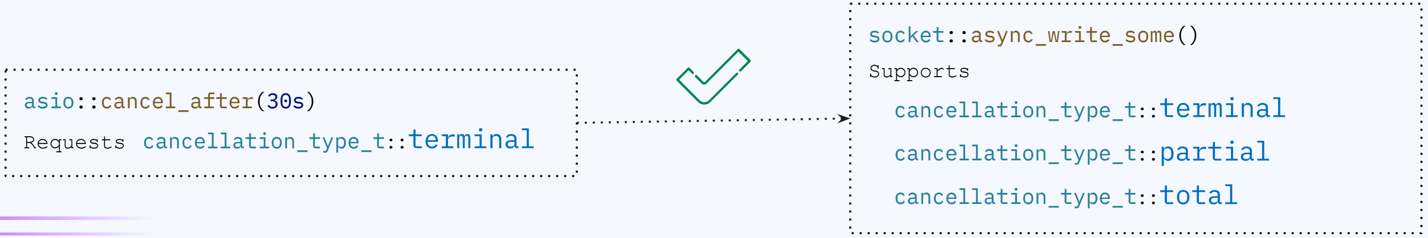
`asio::async_write()`

Total

`cancellation_type_t::total`

Operation either completes or has no side-effects

`socket::async_write_some()`



```
asio::cancel_after(30s)
Requests cancellation_type_t::terminal
```

```
socket::async_write_some()
Supports
  cancellation_type_t::terminal
  cancellation_type_t::partial
  cancellation_type_t::total
```

Cancellation types

Guarantees after cancellation

Terminal

`cancellation_type_t::terminal`

I/O object left in undefined state
Close or destroy

`http::async_write()`

Partial

`cancellation_type_t::partial`

Operation may have side-effects observable through its API

`asio::async_write()`

Total

`cancellation_type_t::total`

Operation either completes or has no side-effects

`socket::async_write_some()`

```
asio::cancel_after(30s, asio::cancellation_type_t::total)
Requests cancellation_type_t::total
```

```
http::async_write()
Supports
  cancellation_type_t::terminal
```