Boost.Asio

The universal async model

Rubén Pérez Hidalgo

Sync vs. async programming

Sync vs. async programming

Sync	**Async**
Function calls block	Functions launch an operation and call a continuation handler
Thread-based	Event driven (can use multiple threads)
Smaller throughput (context switches)	Higher throughput
Simple	More complex (depends on your library)
No portable way to set timeouts to operations	More versatile

Asio

- Platform-independent async networking
- Very flexible
- High performance
- Header-only - long build times
- Complex
- C++11

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

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

```
asio::io_service srv;
asio::ip::tcp::socket sock (srv);
sock.async_write_some(..., boost::bind(on_write, ...));
```

```
~$ sudo apt update
```

HTTP client


Client
Server
GET / HTTP/1.1
Host: example.com
User-Agent: Asio
Accept: */*
Headers
Blank line
Body
HTTP/1.1 200 OK
Content-Length: 1256
Content-Type: text/html; charset=UTF-8
<!doctype html>
<html>
...

Sync: initial prototype

```
constexpr std::string_view request =
       "GET / HTTP/1.1\r\n"
       "Host: example.com\r\n"
       "User-Agent: Asio\r\n"
       "Accept: */*\r\n\r\n";

void handle_request_v1(asio::io_context& ctx)
{
   asio::ip::tcp::socket sock(ctx);

   // Connect to the server
   sock.connect(asio::ip::tcp::endpoint(asio::ip::address::from_string("18.154.41.87"), 80));

   // Write the request
   sock.write_some(asio::buffer(request));

   // Read the response
   std::array<char, 1024> buff;
   std::size_t bytes_read = sock.read_some(asio::buffer(buff));
   std::cout << std::string_view(buff.data(), bytes_read) << std::endl;
}
```

I/O object

```
int main()
{
   asio::io_context ctx;
   handle_request_v1(ctx);
}
```

Execution context

- Handler queue
- Timer queue
- Epoll reactor
- …

Sync: short reads & writes

```
void handle_request_v2(asio::io_context& ctx)
{
   asio::ip::tcp::socket sock(ctx);

   // Connect to the server
   sock.connect(asio::ip::tcp::endpoint(asio::ip::address::from_string("18.154.41.87"), 80));

   // Write the request
   asio::write(sock, asio::buffer(request));

   // Read the response
   std::string buff;
   std::size_t bytes_read = asio::read_until(sock, asio::dynamic_buffer(buff), "\r\n\r\n");
   std::cout << std::string_view(buff.data(), bytes_read) << std::endl;
}
```

Composed operation

Sync: resolving hostnames

```
void handle_request_v3(asio::io_context& ctx)
{
   asio::ip::tcp::socket sock(ctx);
   asio::ip::tcp::resolver resolv(ctx);

   // Resolve the hostname and port into a set of endpoints
   asio::ip::tcp::resolver::results_type endpoints = resolv.resolve("example.com", "80");

   // Connect to the server
   asio::connect(sock, endpoints);

   // Write the request
   asio::write(sock, asio::buffer(request));

   // Read the response
   std::string buff;
   std::size_t bytes_read = asio::read_until(sock, asio::dynamic_buffer(buff), "\r\n\r\n");
   std::cout << std::string_view(buff.data(), bytes_read) << std::endl;
}
```

Connect to each endpoint until one succeeds

Sync: baseline

Executor

- Lightweight handle to execution context
- More generic

```
void handle_request_v4(asio::any_io_executor ex)
{
   asio::ip::tcp::socket sock(ex);
   asio::ip::tcp::resolver resolv(ex);

   // Resolve the hostname and port into a set of endpoints
   asio::ip::tcp::resolver::results_type endpoints = resolv.resolve("example.com", "80");

   // Connect to the server
   asio::connect(sock, endpoints);

   // Write the request
   asio::write(sock, asio::buffer(request));

   // Read the response
   std::string buff;
   std::size_t bytes_read = asio::read_until(sock, asio::dynamic_buffer(buff), "\r\n\r\n");
   std::cout << std::string_view(buff.data(), bytes_read) << std::endl;
}
```

Async: callbacks

```
class request_handler : public std::enable_shared_from_this<request_handler>
{
   asio::ip::tcp::socket sock;
   asio::ip::tcp::resolver resolv;
   std::string buff;

public:
   request_handler(asio::any_io_executor ex) : sock(ex), resolv(ex) {}

   void start_resolve()
   {
       resolv.async_resolve("example.com", "80", [self = shared_from_this()](error_code ec, asio::ip::tcp::resolver::results_type endpoints) {
           if (ec)
               std::cerr << "Error resolving endpoints: "<< ec.message() << std::endl;
           else
               self->start_connect(std::move(endpoints));
       });
   }

   void start_connect(const asio::ip::tcp::resolver::results_type& endpoints)
   {
       asio::async_connect(sock, endpoints, [self = shared_from_this()](error_code ec, auto) {
           if (ec)
               std::cerr << "Error connecting: "<< ec.message() << std::endl;
           else
               self->start_write();
       });
   }
```

Stable addresses

Completion signature

Initiating function

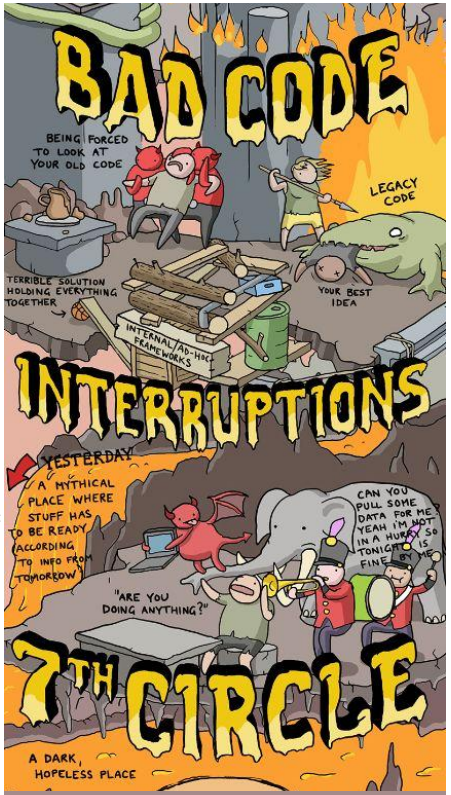
Async: callbacks

```
class request_handler : public std::enable_shared_from_this<request_handler>
{
   asio::ip::tcp::socket sock;
   asio::ip::tcp::resolver resolv;
   std::string buff;

public:
   request_handler(asio::any_io_executor ex) : sock(ex), resolv(ex) {}

   void start_resolve()
   {
       resolv.async_resolve("python.org", "80", [self = shared_from_this()](error_code ec, asio::ip::tcp::resolver::
           if (ec)
               std::cerr << "Error resolving endpoints: "<< ec.message() << std::endl;
           else
               self->start_connect(std::move(endpoints));
       });
   }

   void start_connect(const asio::ip::tcp::resolver::results_type& endpoints)
   {
       asio::async_connect(sock, endpoints, [self = shared_from_this()](error_code ec, auto) {
           if (ec)
               std::cerr << "Error connecting: "<< ec.message() << std::endl;
           else
               self->start_write();
       });
   }
```



Undocumented, 7M LOC callback-based codebase with no tests

Async: coroutines

```
asio::awaitable<void> handle_request_impl ()
{
   // Coroutines know which executor are using
   asio::any_io_executor ex = co_await asio::this_coro::executor;

   // I/O objects
   asio::ip::tcp::socket sock(ex);
   asio::ip::tcp::resolver resolv(ex);

   // Resolve the hostname and port into a set of endpoints
   auto endpoints = co_await resolv.async_resolve("example.com", "80", asio::deferred);

   // Connect to the server
   co_await asio::async_connect(sock, endpoints, asio::deferred);

   // Write the request
   co_await asio::async_write(sock, asio::buffer(request), asio::deferred);

   // Read the response
   std::string buff;
   std::size_t bytes_read = co_await asio::async_read_until(
       sock, asio::dynamic_buffer(buff), "\r\n\r\n", asio::deferred);
}
```

```
void handle_request(asio::any_io_executor ex)
{
   asio::co_spawn(ex, handle_request_impl, [](std::exception_ptr exc) {
       if (exc)
           std::rethrow_exception(exc);
   });
}
```

Completion token

- Return type (according to completion signature)
- Initiation time (operation starts when co_await'ed)

```
void(error_code, std::size_t) => std::size_t
void(error_code) => void
```

Async: as_tuple

```
asio::awaitable<error_code> handle_request_impl ()
{
   asio::any_io_executor ex = co_await asio::this_coro::executor;
   asio::ip::tcp::socket sock(ex);
   asio::ip::tcp::resolver resolv(ex);

   // Completion token that combines handler arguments into a std::tuple
   // Can be used to use error codes instead of exceptions
   constexpr auto tok = asio::as_tuple(asio::deferred);

   // Resolve the hostname and port into a set of endpoints
   auto [ec1, endpoints] = co_await resolv.async_resolve("example.com", "80", tok);
   if (ec1)
       co_return ec1;

   // Connect to the server
   auto [ec2, unused] = co_await asio::async_connect(sock, endpoints, tok);
   if (ec2)
       co_return ec2;
```

Async: associated characteristics

```
asio::awaitable<void> handle_request_impl ()
{
   // Coroutines know which executor are using
   asio::any_io_executor ex = co_await asio::this_coro::executor;

   // I/O objects
   asio::ip::tcp::socket sock(ex);
   asio::ip::tcp::resolver resolv(ex);

   // A completion token with an associated allocator
   auto tok = asio::bind_allocator(custom_allocator<void>(), asio::deferred);

   // Resolve the hostname and port into a set of endpoints
   auto endpoints = co_await resolv.async_resolve("example.com", "80", tok);

   // Connect to the server
   co_await asio::async_connect(sock, endpoints, tok);

   // Write the request
   co_await asio::async_write(sock, asio::buffer(request), tok);

   // Read the response
   std::string buff;
   std::size_t
       bytes_read = co_await asio::async_read_until(sock, asio::dynamic_buffer(buff), "\r\n\r\n",
tok);
   std::cout << std::string_view(buff.data(), bytes_read) << std::endl;
```

Customize how the operation runs:

- Allocator
- Executor
- Immediate executor
- Cancellation slot

Compliant composed op:

- Usable with any completion token
- Propagates associated characteristics

Not easy - libraries help!

Async: parallel groups

```
asio::awaitable<void> handle_request_with_timeout ()
{
   auto ex = co_await asio::this_coro::executor;

   // Setup a timer
   asio::steady_timer timer(ex);
   timer.expires_after(std::chrono::seconds(5));

   // Launch the coroutine and the timer in parallel
   auto [completion_order, coro_exc, timer_ec] = co_await asio::experimental::make_parallel_group(
       asio::co_spawn(ex, handle_request_impl, asio::deferred),
       timer.async_wait(asio::deferred)
   ).async_wait(
       asio::experimental::wait_for_one(),                    // After the 1st op completes, cancel the other
       asio::deferred
   );

   // Check for errors
   if (coro_exc)
       std::rethrow_exception(coro_exc);
}
```

```
std::array<std::size_t, 2>
    {0, 1} // coro finished 1st
    {1, 0} // timer finished 1st

std::exception_ptr // coro's result

error_code // timer's result
```

Boost.Beast

```
asio::awaitable<void> handle_request_impl ()
{
   auto ex = co_await asio::this_coro::executor;
   asio::ip::tcp::socket sock(ex);
   asio::ip::tcp::resolver resolv(ex);

   // Resolve the hostname and port into a set of endpoints
   auto endpoints = co_await resolv.async_resolve("example.com", "80", asio::deferred);

   // Connect to the server
   co_await asio::async_connect(sock, endpoints, asio::deferred);

   // Compose and write the request
   http::request<http::string_body> req{http::verb::get, "/", 11};
   req.set(http::field::host, "example.com");
   req.set(http::field::user_agent, "Beast");
   co_await http::async_write(sock, req, asio::deferred);

   // Read the response
   beast::flat_buffer buff;
   http::response<http::string_body> res;
   co_await http::async_read(sock, buff, res, asio::deferred);
   std::cout << res << std::endl;
}
```

The ecosystem

Boost.Beast

HTTP and websockets

Boost.MySQL

MySQL and MariaDB

Boost.Redis

Redis connectivity

Boost.Cobalt

Awaitables, generators and algorithms

Alternatives

- libuv (C): https://github.com/libuv/libuv
  uvw (C++ wrapper)
- uSockets (C): https://github.com/uNetworking/uSockets
  uWebSockets (based on uSockets)
- libunifex (experimental):
  https://github.com/facebookexperimental/libunifex

```
self.complete(error_code());
```

Thank you!

https://github.com/anarthal/usingstdcpp-2024

Rubén Pérez Hidalgo

Async: per-operation cancellation

Object-wide cancellation

Cancels all outstanding operations on an object

No way to cancel composed operations

```
sock.cancel()
```

Per-operation cancellation

Based on cancellation slots

Doesn’t affect other operations

Usable within composed operations

Async: per-operation cancellation

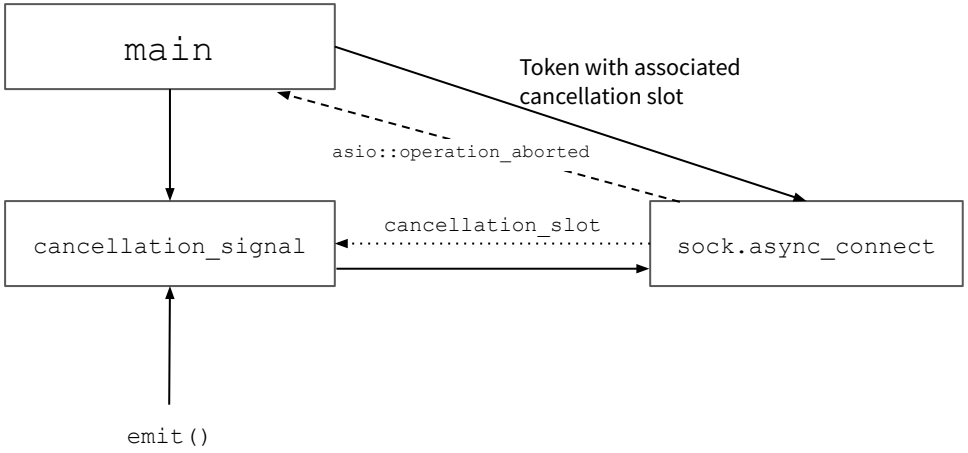
```
int main()
{
   asio::io_context ctx;
   asio::ip::tcp::socket sock{ctx};
   asio::steady_timer timer{ctx};
   const auto endpoint = asio::ip::tcp::endpoint(/*...*/);

   asio::cancellation_signal sig;

   // Wait 5 seconds, then trigger cancellation
   timer.expires_after(std::chrono::seconds(5));
   timer.async_wait([&sig](error_code) {
       // When the timer fires, trigger cancellation
       sig.emit(asio::cancellation_type::terminal);
   });

   sock.async_connect(
       endpoint,
       asio::bind_cancellation_slot(sig.slot(), [](error_code ec) {
            std::cout << "Connect finished: " << ec.message() <<
std::endl;
       })
   );

   ctx.run();
}
```

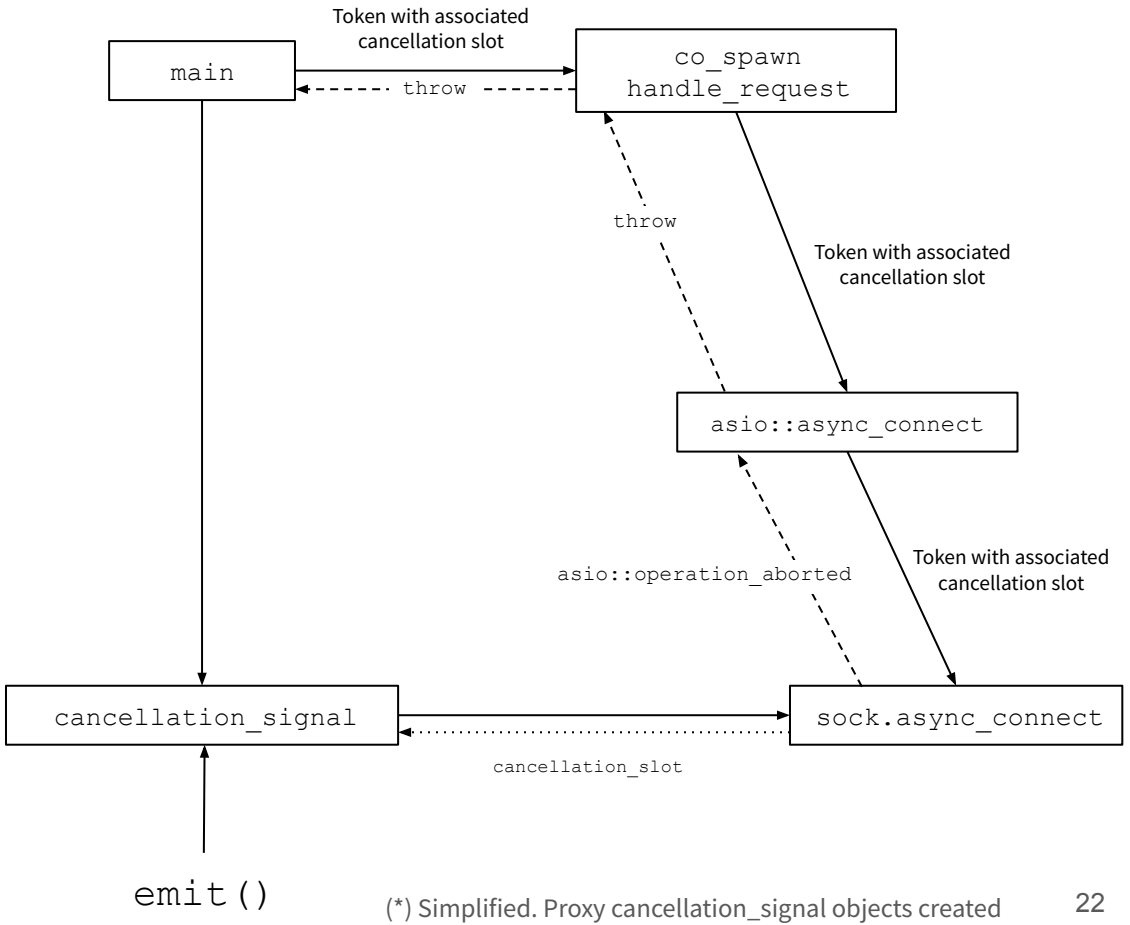


Async: per-operation cancellation

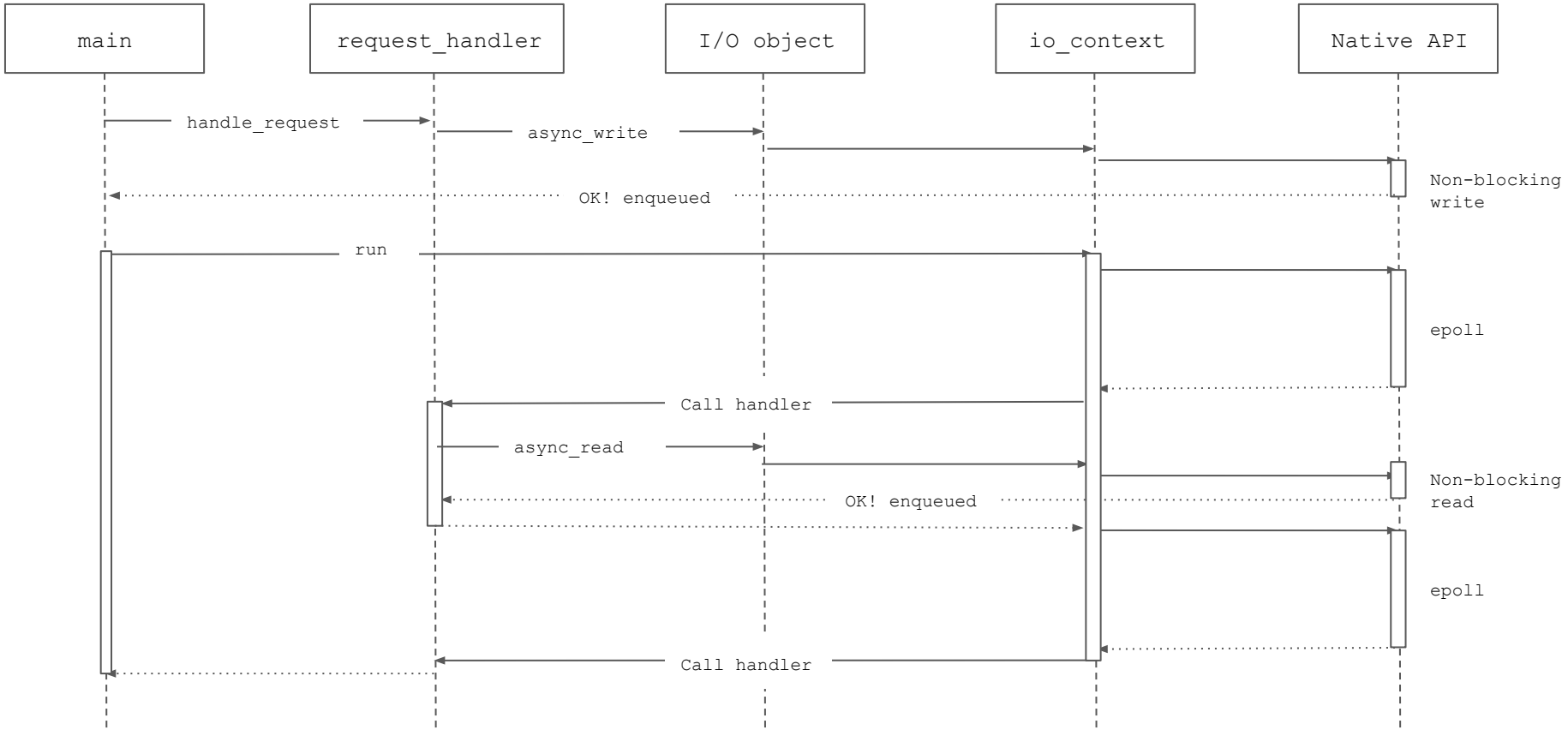
```
int main()
{
    asio::io_context ctx;
    asio::ip::tcp::socket sock{ctx};
    asio::steady_timer timer{ctx};
    asio::cancellation_signal sig;

    // Wait 5 seconds, then trigger cancellation
    timer.expires_after(std::chrono::seconds(5));
    timer.async_wait([&sig](error_code) {
        // When the timer fires, trigger cancellation
        sig.emit(asio::cancellation_type::terminal);
    });

    asio::co_spawn(
        ctx,
        handle_request_impl,
        asio::bind_cancellation_slot(
            sig.slot(),
            [](std::exception_ptr ptr) {
                if (ptr) std::rethrow_exception(ptr);
            }
        )
    );
    ctx.run();
}
```



(*) Simplified. Proxy cancellation_signal objects created


main
request_handler
I/O object
io_context
Native API
handle_request
async_write
Non-blocking write
OK! enqueued
run
epoll
Call handler
async_read
Non-blocking read
OK! enqueued
epoll
Call handler