From async I/O to senders and receivers

Why another abstraction for asynchronous C++?

Pavel Šimerda

2026-10-20

- The overhead of operating system threads
- Using threads as an abstraction for CPU cores
- Returning I/O to cooperative multitasking
- Looking for a good abstraction for cooperative tasks

© 2026 Pavel Šimerda

Building the storyline

Client sends a request and receives a reply.

```
auto request = prepare_request();
auto channel = tcp_connect(host_name, port_number);

channel.send_all(request.header());
channel.send_all(request.payload());
std::array<std::byte, Header::wire_size> raw_header;
channel.recv_exact(raw_header);
auto header = Header::parse(raw_header);
std::vector<std::byte> reply_payload(header.payload_size());
channel.recv_exact(reply_payload);
channel.shutdown();
Reply reply{header, std::move(reply_payload)};
```

Connecting, sending and receiving are all implemented using blocking system calls. More concurrent clients require more operating system threads.

© 2026 Pavel Šimerda

Server receives a request, performs heavy computation and sends a result.

```
auto listener = tcp_listen(port_number);

std::jthread session{[channel = listener.accept()] mutable {
    std::array<std::byte, Header::wire_size> raw_header;
    channel.recv_exact(raw_header);
    auto header = Header::parse(raw_header);
    std::vector<std::byte> payload(header.payload_size());
    channel.recv_exact(payload);
    Request request{header, payload};
    Reply reply = process_slowly(request);
    channel.send_all(reply.header());
    channel.send_all(reply.payload());
    channel.shutdown();
}};
```

Also uses blocking calls for listening, accepting and receiving and also blocks the thread with slow computation. Requires operating system threads to handle concurrent connections.

© 2026 Pavel Šimerda

Illustrative code for `tcp_connect`, as called in the client example.

```
Channel tcp_connect(const char* host_name, std::uint16_t port_number) {
    int raw = ::socket(AF_INET6, SOCK_STREAM, 0);
    if (raw < 0) throw std::system_error{errno, std::system_category(), "socket"};
    FileDescriptor fd{raw};

    sockaddr_in6 server{};
    server.sin6_family = AF_INET6;
    server.sin6_port = ::htons(port_number);
    int ret = ::inet_pton(AF_INET6, host_name, &server.sin6_addr);
    if (ret != 1) throw std::invalid_argument{"address is not an IPv6 literal"};
    ret = ::connect(fd.get(), reinterpret_cast<sockaddr*>(&server), sizeof server);
    if (ret < 0) throw std::system_error{errno, std::system_category(), "connect"};
    return Channel{std::move(fd)};
}
```

This function would block on name resolution (omitted) and then blocks on the `connect()` system call.

© 2026 Pavel Šimerda

Illustrative code for `tcp_listen`, as called in the server example.

```
Listener tcp_listen(std::uint16_t port_number) {
    int raw = ::socket(AF_INET6, SOCK_STREAM, 0);
    if (raw < 0) throw std::system_error{errno, std::system_category(), "socket"};
    FileDescriptor fd{raw};

    int one = 1;
    int ret = ::setsockopt(fd.get(), SOL_SOCKET, SO_REUSEADDR, &one, sizeof one);
    if (ret < 0) throw std::system_error{errno, std::system_category(), "setsockopt"};

    sockaddr_in6 address{
        .sin6_family = AF_INET6,
        .sin6_port = ::htons(port_number),
        .sin6_addr = in6addr_any,
    };
    ret = ::bind(fd.get(), reinterpret_cast<sockaddr*>(&address), sizeof address);
    if (ret < 0) throw std::system_error{errno, std::system_category(), "bind"};
    ret = ::listen(fd.get(), SOMAXCONN);
    if (ret < 0) throw std::system_error{errno, std::system_category(), "listen"};
    return Listener{std::move(fd)};
}
```

The listener uses a blocking socket (the default) that would later block in the `accept()` call.

© 2026 Pavel Šimerda

Implementation of `Listener::accept()` used in the server example.

```
Channel Listener::accept() const {
    int conn = ::accept(fd_.get(), nullptr, nullptr);
    if (conn < 0) throw std::system_error{errno, std::system_category(), "accept"};
    return Channel{FileDescriptor{conn}};
}
```

The listener blocks in the `accept()` call, as expected.

© 2026 Pavel Šimerda

Implementation of `Channel::recv_exact()` and `Channel::send_all` from the client and server examples.

```
void Channel::recv_exact(std::span<std::byte> destination) {
    while (!destination.empty()) {
        ssize_t n = ::recv(fd_.get(), destination.data(), destination.size(), 0);
        if (n < 0) throw std::system_error{errno, std::system_category(), "recv"};
        if (n == 0) throw std::runtime_error{"peer closed"};
        destination = destination.subspan(n);
    }
}
```

```
void Channel::send_all(std::span<const std::byte> data) {
    while (!data.empty()) {
        ssize_t n = ::send(fd_.get(), data.data(), data.size(), 0);
        if (n < 0) throw std::system_error{errno, std::system_category(), "send"};
        data = data.subspan(n);
    }
}
```

Both of them block, possibly multiple times, in the `recv()` or `send()` calls.

© 2026 Pavel Šimerda

Nonblocking calls

Adaptation of `tcp_connect` to only block explicitly on file descriptor readiness.

```
Channel tcp_connect_nonblocking(const char* host_name, std::uint16_t port_number) {
    int raw = ::socket(AF_INET6, SOCK_STREAM | SOCK_NONBLOCK, 0);
    /* ... same as tcp_connect ... */
    int ret = ::connect(fd.get(), reinterpret_cast<sockaddr*>(&server), sizeof server);
    if (ret < 0) {
        if (errno != EINPROGRESS) throw std::system_error{errno, std::system_category(), "connect"};

        wait_until_writable(fd);

        int connect_error = 0;
        socklen_t error_size = sizeof connect_error;
        ret = ::getsockopt(fd.get(), SOL_SOCKET, SO_ERROR, &connect_error, &error_size);
        if (ret < 0) throw std::system_error{errno, std::system_category(), "getsockopt"};
        if (connect_error != 0) throw std::system_error{connect_error, std::system_category(), "connect"};
    }
    return Channel{std::move(fd)};
}
```

This function may block once, waiting for the socket to become *ready for writing*. Then the delayed result of `connect()` can be retrieved.

© 2026 Pavel Šimerda

Adaptation of `tcp_listen` and `Listener::accept()` to only block explicitly on file descriptor readiness.

```
Listener tcp_listen_nonblocking(std::uint16_t port_number) {
    int raw = ::socket(AF_INET6, SOCK_STREAM | SOCK_NONBLOCK, 0);
    /* ... same as tcp_listen ... */
}

Channel Listener::accept_nonblocking() const {
    for (;;) {
        wait_until_readable(fd_);

        int conn = ::accept4(fd_.get(), nullptr, nullptr, SOCK_NONBLOCK | SOCK_CLOEXEC);
        if (conn >= 0) return Channel{FileDescriptor{conn}};
        if (errno != EAGAIN && errno != EWOULDBLOCK)
            throw std::system_error{errno, std::system_category(), "accept4"};
    }
}
```

This function can block, usually once, to wait for the socket to become *ready for reading*. Then the `accept()` system call usually succeeds.

© 2026 Pavel Šimerda

Adaptation of `Channel::recv_exact` to only block explicitly on file descriptor readiness.

```
void Channel::recv_exact_nonblocking(std::span<std::byte> destination) {
    while (!destination.empty()) {
        ssize_t n;
        for (;;) {
            wait_until_readable(fd_);

            n = ::recv(fd_.get(), destination.data(), destination.size(), 0);
            if (n >= 0)
                break;
            if (errno != EAGAIN && errno != EWOULDBLOCK)
                throw std::system_error{errno, std::system_category(), "recv"};
        }
        destination = destination.subspan(n);
    }
}
```

This function can block, usually once, to wait for the socket to become *ready for reading*. Then the `recv()` system call usually succeeds.

© 2026 Pavel Šimerda

Adaptation of `Channel::send_all` to only block explicitly on file descriptor readiness.

```
void Channel::send_all_nonblocking(std::span<const std::byte> data) {
    while (!data.empty()) {
        ssize_t n;
        for (;;) {
            wait_until_writable(fd_);

            n = ::send(fd_.get(), data.data(), data.size(), 0);
            if (n >= 0)
                break;
            if (errno != EAGAIN && errno != EWOULDBLOCK)
                throw std::system_error{errno, std::system_category(), "send"};
        }
        data = data.subspan(n);
    }
}
```

This function can block, usually once, to wait for the socket to become *ready for writing*. Then the `send()` system call usually succeeds.

© 2026 Pavel Šimerda

For experiments with nonblocking sockets, explicit waits for readiness may be useful.

```
void wait_until(FileDescriptor& fd, short events) {
    pollfd poll_fd{fd.get(), events, 0};
    int ret = ::poll(&poll_fd, 1, -1);
    if (ret < 0) throw std::system_error{errno, std::system_category(), "poll"};
}

void wait_until_readable(FileDescriptor& fd) { wait_until(fd, POLLIN); }
void wait_until_writable(FileDescriptor& fd) { wait_until(fd, POLLOUT); }
```

In real programs we usually need to wait for *readiness events* from multiple file descriptors to allow starting and resuming concurrent tasks within a single operating system thread.

© 2026 Pavel Šimerda

The reactor pattern

Illustrative code for a *readiness* event loop.

```
void Reactor::run() {
    while (!waiters_.empty()) {
        std::vector<pollfd> fds;
        for (const Waiter& waiter : waiters_)
            fds.push_back({.fd = waiter.fd, .events = waiter.events, .revents = 0});
        int ret = ::poll(fds.data(), fds.size(), -1);
        if (ret < 0) throw std::system_error{errno, std::system_category(), "poll"};
        auto waiting = std::move(waiters_);
        waiters_.clear();
        for (std::size_t i = 0; i < waiting.size(); ++i) {
            if (fds[i].revents != 0)
                waiting[i].continuation.resume();
            else
                waiters_.push_back(std::move(waiting[i]));
        }
    }
}
```

Each ready continuation is resumed exactly once and then forgotten. One that still has to wait must re-register. You can also explore *completion* event loops.

© 2026 Pavel Šimerda

Concurrent tasks need to run in steps using the reactor pattern.

```
void Session::start(Reactor& reactor, Channel channel) {
    auto session = std::make_unique<Session>(std::move(channel));
    Session& s = *session;
    recv_exact(reactor, s.channel, s.header_bytes,
        [&reactor, &s, session = std::move(session)] mutable {
            s.header = Header::parse(s.header_bytes);
            s.payload.resize(s.header.payload_size());
            recv_exact(reactor, s.channel, s.payload,
                [&reactor, &s, session = std::move(session)] mutable {
                    s.reply = process_slowly(Request{s.header, s.payload});
                    send_all(reactor, s.channel, s.reply.header(),
                        [&reactor, &s, session = std::move(session)] mutable {
                            send_all(reactor, s.channel, s.reply.payload(),
                                [&s, session = std::move(session)] {
                                    s.channel.shutdown();
                                });
                        });
                });
        });
}
```

Reactor needs to use `std::move_only_function` with possible heap allocation. Session in this example also uses heap allocation. All of these can be eliminated with clever design.

© 2026 Pavel Šimerda

Illustrative code for the asynchronous version of `recv_exact`.

```
void recv_exact(Reactor& reactor, Channel& channel, std::span<std::byte> remaining,
                        std::move_only_function<void()> next) {
    if (recv_exact_step(channel, remaining) == Progress::pending) {
        reactor.on_readable(channel.fd().get(),
            [&reactor, &channel, remaining, next = std::move(next)] mutable {
                recv_exact(reactor, channel, remaining, std::move(next));
            });
        return;
    }
    next();
}
```

What would be a loop in blocking code is a rescheduling of the current continuation to run again when the file descriptor is ready. Implementation of an asynchronous version of `send_all` would be analogous.

© 2026 Pavel Šimerda

A decent alternative to nested callbacks is a state machine.

```
class SessionStateMachine : public StateMachine {
public:
    void resume(Reactor& reactor) override;

private:
    enum class State { recv_header, recv_payload, send_header, send_payload, done };

    Channel channel;
    State state = State::recv_header;

    std::array<std::byte, Header::wire_size> header_bytes;
    Header header;
    std::vector<std::byte> payload;
    Reply reply;

    std::span<std::byte> receiving{header_bytes};
    std::span<const std::byte> sending;
};
```

What are local variables in blocking functions, and captures to the lambdas, is member variables to hand-written state machines.

© 2026 Pavel Šimerda

Continuations are `case` blocks selected by `State` values.

```
void SessionStateMachine::resume(Reactor& reactor) {
    switch (state) {
        /* ... */
            receiving = payload;
            state = State::recv_payload;
            [[fallthrough]];
        case State::recv_payload:
            if (recv_exact_step(channel, receiving) == Progress::pending) {
                reactor.on_readable(channel.fd().get(), *this);
                return;
            }
            reply = process_slowly(Request{header, payload});
            sending = reply.header();
            state = State::send_header;
            [[fallthrough]];
        /* ... */
    }
}
```

This example uses `[[fallthrough]]` to signal that we intentionally run the next step immediately in case the operations could be carried out immediately. Reactor understands the `.resume()` protocol.

© 2026 Pavel Šimerda

The state machine version of `recv_exact` has its own simple state.

```
enum class Progress { pending, done };

Progress recv_exact_step(Channel& channel, std::span<std::byte>& remaining) {
    while (!remaining.empty()) {
        ssize_t n = ::recv(channel.fd().get(), remaining.data(), remaining.size(), 0);
        if (n < 0) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) return Progress::pending;
            throw std::system_error{errno, std::system_category(), "recv"};
        }
        if (n == 0) throw std::runtime_error{"peer closed"};
        remaining = remaining.subspan(n);
    }
    return Progress::done;
}
```

This way the calling state machine knows when to re-run the `recv_exact_step()` and when to continue with the next operations.

© 2026 Pavel Šimerda

Coroutine syntax is the closest to the original blocking code.

```
Task run_session(Reactor& reactor, Channel channel) {
    std::array<std::byte, Header::wire_size> header_bytes;
    co_await recv_exact(reactor, channel, header_bytes);
    auto header = Header::parse(header_bytes);
    std::vector<std::byte> payload(header.payload_size());
    co_await recv_exact(reactor, channel, payload);
    auto reply = process_slowly(Request{header, payload});
    co_await send_all(reactor, channel, reply.header());
    co_await send_all(reactor, channel, reply.payload());
    channel.shutdown();
}

Task recv_exact(Reactor& reactor, Channel& channel, std::span<std::byte> remaining) {
    while (recv_exact_step(channel, remaining) == Progress::pending)
        co_await reactor.readable(channel.fd().get());
}
```

However, C++20 coroutines are allocated on the heap. This may be a blocker in many setups. The reactor stores the `std::coroutine_handle<>`.

© 2026 Pavel Šimerda

Senders

All of the existing alternatives have their flaws.

- Lambda chains are ugly and have lifetime issues
- State machines are too easy to mess with
- Coroutines rely on dynamic allocation

Senders allow you to build the chain from the beginning to the end, adding new steps to the work itself, before it becomes the state object.

© 2026 Pavel Šimerda

Sender chains are normally all you should need to care about.

- Senders and sender adaptors, the building blocks
- Receivers, the multi-way callback objects
- Operation state, the actual state objects

Receivers and operation states are there to provide the runtime environment for the senders. The implementations look like magic but they are really just a set of clever template tricks.

© 2026 Pavel Šimerda

You can run a sender and wait for its result in your thread.

```
auto [reply] = std::this_thread::sync_wait(std::move(work)).value();
```

Behind the scenes this connects the sender to a receiver that is used to deliver the result back to the waiting thread. You can make a special version of `sync_wait` to run a reactor instead of just waiting.

© 2026 Pavel Šimerda

A sender may be as simple as a provider of a predetermined value.

```
namespace exec = std::execution;

auto work = exec::just(reply);
```

This is an easy way to express a value that goes into a sender chain.

© 2026 Pavel Šimerda

Or it can be a transformation of an existing sender.

```
auto work = exec::just(request)
    | exec::then([](Request request) { return process_slowly(request); });
```

It isn't all that useful to transform the result of `just` or of another `then`. The usual position would be between I/O operations.

© 2026 Pavel Šimerda

I/O operations would be done by chaining another sender.

```
auto work = recv_exact(reactor, channel, Header::wire_size)
    | exec::then([](std::vector<std::byte> bytes) { return Header::parse(bytes); })
    | exec::let_value([&](Header& header) {
        return recv_exact(reactor, channel, header.payload_size());
    });
```

`let_value` continues with the sender its lambda returns, so the size read from the header decides the second receive. Each of the I/O senders can pass the operation state to a reactor and have the continuation invoked from the reactor.

© 2026 Pavel Šimerda

You may need to run slow work outside your I/O thread.

```
auto work = exec::just(request)
    | exec::continues_on(workers)
    | exec::then([](Request request) { return process_slowly(request); })
    | exec::continues_on(loop);
```

The operation is effectively rescheduled twice and moved between different schedulers. `workers` is a scheduler of a thread pool, for example one from stdexec's `static_thread_pool` or `std::execution::get_parallel_scheduler()`. `loop` is a scheduler of the reactor thread, where I/O operations have to be started and their continuations are run.

© 2026 Pavel Šimerda

You may decide to catch the error in the chain and act upon it.

```
auto work = exec::just(std::vector<std::byte>{})
    | exec::then([](std::vector<std::byte> bytes) { return Header::parse(bytes); })
    | exec::upon_error([](auto&&) noexcept {
        std::println(std::cerr, "bad header");
        return Header{};
    });
```

This way if the header is malformed, you can decide what to do about it, for example to provide a sane default value after reporting the error.

© 2026 Pavel Šimerda

It is possible to start two different tasks at once.

```
auto work = exec::when_all(
    exec::schedule(workers) | exec::then([&] { return process_slowly(first); }),
    exec::schedule(workers) | exec::then([&] { return process_slowly(second); }));
```

Both are scheduled to run and the next continuation is run when both have succeeded. If one of them fails or is stopped, the other is asked to stop.

© 2026 Pavel Šimerda

Client sends a request and receives a reply.

```
auto request = prepare_request();

auto client = connect_to(reactor, host_name, port_number)
    | exec::let_value([&](Channel& channel) {
        return send_all(reactor, channel, request.header())
            | exec::let_value([&] { return send_all(reactor, channel, request.payload()); })
            | exec::let_value([&] { return recv_exact(reactor, channel, Header::wire_size); })
            | exec::then([](std::vector<std::byte> bytes) { return Header::parse(bytes); })
            | exec::let_value([&](Header& header) {
                return recv_exact(reactor, channel, header.payload_size())
                    | exec::then([&](std::vector<std::byte> payload) {
                        return Reply{header, std::move(payload)};
                    });
            })
            | exec::then([&](Reply reply) { channel.shutdown(); return reply; });
    });
```

Every step starts when the previous one completes, and no operating system thread blocks in between.

© 2026 Pavel Šimerda

Server receives a request, performs heavy computation and sends a result.

```
auto listener = tcp_listen_nonblocking(port_number);

auto server = accept_connection(reactor, listener)
    | exec::let_value([&](Channel& channel) {
        return recv_exact(reactor, channel, Header::wire_size)
            | exec::then([](std::vector<std::byte> bytes) { return Header::parse(bytes); })
            | exec::let_value([&](Header& header) {
                return recv_exact(reactor, channel, header.payload_size())
                    | exec::then([&](std::vector<std::byte> payload) {
                        return Request{header, std::move(payload)};
                    });
            })
            | exec::continues_on(workers)
            | exec::then([](Request request) { return process_slowly(request); })
            | exec::continues_on(loop)
            | exec::let_value([&](Reply& reply) {
                return send_all(reactor, channel, reply.header())
                    | exec::let_value([&] { return send_all(reactor, channel, reply.payload()); });
            })
            | exec::then([&] { channel.shutdown(); });
    })
    | exec::upon_error([](auto&&) noexcept { std::println(std::cerr, "session failed"); });
```

The size read from the header decides the second receive, the heavy computation runs on the pool, the reply is sent from the reactor thread, and an error anywhere ends up in `upon_error`.

© 2026 Pavel Šimerda

Questions?

Pavel Šimerda

pavel@simerda.eu