P2586R0: Standard Secure Networking

Document #:	P2586R0
Date:	2022-09-13
Project:	Programming Language C++ SG4 Study Group Networking Library Evolution Working Group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

A proposal for standard secure networking in C++ using the same deadline-based optionally non-blocking scatter-gather low-level i/o API design as [P1183] *`file_handle`* *and* *`mapped_file_handle`*. Neither asynchronous nor coroutine i/o is proposed at this time, and therefore this proposal has no opinion which concerns [P2300] *`std::execution`*. To reduce scope to the feasible, neither proposed is any form of secure i/o other than Transport Layer Security (TLS) v1.2 or higher, which is a stream based, connection-orientated, protocol.

Like with all my low-level i/o proposals, this proposal is designed to work well on Freestanding targets without `malloc`, and without C++ exceptions globally enabled nor Thread Local Storage being available, and on implementations where TLS networking is offloaded onto a dedicated coprocessor connected via SPI as would be normal in low end microcontrollers. It therefore does not require the IS to specify in normative wording anything about cryptography nor TLS implementation specifics (other than at a very high abstract level). This proposal's design encourages standard library implementations to hide implementation detail behind a strong ABI boundary, such that TLS backends can be upgraded or changed without requiring the C++ program to be recompiled or relinked.

A reference implementation of the proposed standard secure networking can be found at `https://github.com/ned14/llfio/blob/develop/include/llfio/v2.0/tls_socket_handle.hpp`. It compiles on any C++ 14 compiler, and is known to work on recent editions of GCC, clang and Microsoft Visual Studio, on x86, x64, ARM and AArch64, with the test suite passing on Apple Mac OS/BSD, Linux, and Microsoft Windows. Whilst its enclosing library has been in production use for many years now, this specific reference implementation was written at the request of some members of LEWG for this proposal paper only. It passes its test suite, but is not a mature reference implementation.

Contents

1 Introduction

In the Autumn of 2021 in the vote on [P2464], LEWG decided to stop pursuing the Networking TS as the C++ Standard Library's answer for networking. I was privately asked by some leading members of LEWG to come up with a proposal for standard networking which better fitted all of LEWG's historical concerns ([P1861] is representative, but incomplete) about an ideal design to standardise for networking. I was not well at the time, I had been enduring the lingering side effects of a viral illness which greatly impacted my productivity (and indeed quality of life) for much of 2021. However, I indicated at the time my hope to have a go at a design over the Christmas 2021 holidays, and if I was successful, I would prioritise the writing of this proposal paper over new revisions of all my currently in-flight WG21 proposal papers which have been languishing since year 2020 due to the global pandemic.

I managed to knock something together over Christmas 2021, but it took a further four months of debugging the reference implementation outside of work hours to get it to pass its test suite. I also undertook two rounds of early design review via Reddit `/r/cpp`, and this affected a few design choices:

1. February 2022: `https://www.reddit.com/r/cpp/comments/szg5h8/asking_for_api_design_feedback_on_possible_future/`
2. April 2022: `https://www.reddit.com/r/cpp/comments/u6z8gr/proposed_standard_secure_sockets_reference/`

I would like to thank those from `/r/cpp` who took the time to provide such valuable and useful early feedback.

Alas, completion and submission of this paper then took a backseat to me becoming very very busy outside of work from May 2022 onwards (I am designing and building a house!), so it is only thanks to my current client MayStreet London Stock Exchange Group that you are reading this paper now, as they kindly gave me some work time to get my in-flight standards papers over the line.

As with [P1183] *`file_handle` and `mapped_file_handle`*, this proposal is designed around [P1028] *SG14 `status_code` and standard `error` object*. Writing code using these without [P0709] *Zero-overhead deterministic exceptions: Throwing values* is somewhat clunky, but deterministic and non-dependent on Thread Local Storage which I think are important qualities for the proposed API given its target use cases. One can either write Result-orientated code whereby every potential point of control flow inversion is explicitly annotated using a `TRY` operation, or one can exit Result-

orientated programming into normal C++ by asking failed `result<T>` return transports to tell their `error` object to throw itself as a type-based exception throw.

> [*Note:* The committee may wish to consider that a frequent piece of feedback from the Reddit design reviews was that [P0709] *Zero-overhead deterministic exceptions: Throwing values* would be highly desirable so we can have the compiler stamp out this boilerplate for us, instead of having to write it out by hand manually. – end note]

Given that my last paper on Networking [P2052] *Making modern C++ i/o a consistent API experience from bottom to top* was rejected by SG4 Networking, for this R0 paper revision I am mainly going to describe this proposal via commented source code example of use, and prose describing the overall vision, rather than API specifics, in order to save myself time if this proposal is not very warmly received. This should be sufficient for WG21 to decide if it wants to see more or not. If you wish to review API implementation specifics, you can do so at `https://ned14.github.io/llfio/tls__socket__handle_8hpp.html`.

1.1 Example: Retrieving a web page over HTTPS

This example of use shows how one might retrieve a HTML page from a website via TLS using this proposal:

```
static constexpr string_view test_host = "github.com";
static constexpr string_view get_request = "GET / HTTP/1.0\r\nHost: github.com\r\n\r\n";

// Get the default TLS socket source for this platform. There
// can be many TLS socket sources, each with different
// properties, and you can filter and enumerate and examine
// them. This just gets the standard library's suggested default.
//
// Note that while this is indeed a smart pointer, it does NOT
// mean that the pointer returned is dynamically allocated!
// It may be a reference counted singleton, or a static singleton.
// This goes for all the other smart pointers returned in this
// proposal, they can come from an internal fixed size static
// array. No malloc required!
tls_socket_source_ptr tls_socket_source =
  tls_socket_source_registry::default_source().instantiate().value();

// Create a multiplexable connecting socket from the TLS socket
// source. Multiplexable means you can do i/o on more than one
// of them at a time per kernel thread (which on POSIX would mean
// they are non-blocking, on other platforms e.g. Windows it means
// they are OVERLAPPED).
tls_byte_socket_ptr sock =
  tls_socket_source->multiplexable_connecting_socket(ip::family::any).value();
{
  // Connect the socket to the test host on port 443, timing out
  // after a five second deadline. This sets the TLS session host
  // from the hostname, and the default is to authenticate the TLS
  // certificate of things we connect to using the local system's
  // certificate store. The host name is therefore needed.
  //
```

```
  // This is just the convenience API. If you want to, you can go
  // do each step here separately. In fact, this convenience API
  // is actually implemented 100% using other public APIs.
  result<void> r = sock->connect(test_host, 443, chrono::seconds(5));
  if(r.has_error())
  {
    if(r.error() == errc::timed_out
      || r.error() == errc::host_unreachable
      || r.error() == errc::network_unreachable)
    {
      cout << "\nNOTE: Failed to connect to " << test_host
           << " within five seconds. Error was: " << r.error().message()
           << std::endl;
      return;
    }
    r.value();  // throw the failure as an exception
  }
}

// The string printed here will be implementation defined. A non-normative
// note will suggest item separation with commas.
cout << "\nThe socket which connected to " << test_host
     << " negotiated the cipher " << sock->algorithms_description() << std::endl;

// Create const buffer from which to write the HTTP/1.0 request to the host.
// TLS socket handle defines its const buffer type as being a span<const byte>,
// so because the string view is const char, we need to reinterpret cast.
tls_socket_handle::const_buffer_type get_request_buffer(
  reinterpret_cast<const llfio::byte *>(get_request.data()), get_request.size());

// Write the HTTP/1.0 request to the host. The write() API
// is 100% identical to proposed file_handle::write() and mapped_file_handle::write()
// and in the reference library it is in fact an inherited virtual function
// from the common base byte_io_handle class.
size_t written = sock->write({get_request_buffer}).value();
TEST_REQUIRE(written == get_request.size());

// Fetch the front page. The connection will close once all data is sent
// due to that being HTTP 1.0 default semantics, and that causes reads to
// return no bytes read. Note the deadline in case remote doesn't close the
// socket in a timely fashion.
vector<byte> buffer(4096);
size_t offset = 0;
for(size_t nread = 0;
  (nread = sock->read({{buffer.data() + offset,
    buffer.size() - offset}}, chrono::seconds(3)).value()) > 0;)
{
  offset += nread;
  if(buffer.size() - offset < 1024)
  {
    buffer.resize(buffer.size() + 4096);
  }
}
buffer.resize(offset);
cout << "\nRead from " << test_host << " " << offset
     << " bytes. The first 1024 bytes are:\n\n"
```

```
     << string_view(reinterpret_cast<const char *>(buffer.data()),
    buffer.size()).substr(0, 1024)  << "\n" << std::endl;

// Gracefully shutdown the TLS connection and close the socket
sock->shutdown_and_close().value();
```

Why this example needs a multiplexable socket is worth explaining. Firstly, the difference between a multiplexable and a non-multiplexable handle in LLFIO is that the former can require up to two syscalls per i/o, whereas the latter never requires more than one. Non-multiplexable i/o may be more amenable to DMA in certain circumstances on some platforms than multiplexable, and the kernel scheduler may be able to make more intelligent scheduling decisions if i/o blocks. LLFIO therefore defaults to non-multiplexable as it is the more efficient choice, and it also has portable semantics across all platforms for all types of i/o handle.

Some implementations may be able to implement the above perfectly well with non-multiplexable sockets, however some may refuse non-zero non-infinite deadline i/o on a non-multiplexable socket (just to be clear here, an implementation may support `chrono::seconds(0)` length waits or infinite waits but nothing in between). In the above, we are just lazy and accept the efficiency hit for convenience.

There are surely implementations out there where multiplexable sockets are not available, however whether those are worth explicitly supporting in a future standard I do not know.

1.2 Example: Multiplexing TLS server connections

One can create listening TLS sockets, and poll for new things happening on them:

```
string host;  // set to one of my network cards

// Get me a FIPS 140 compliant source of TLS sockets which uses kernel
// sockets, this call takes bits set and bits masked, so the implementation
// features bitfield is masked for the FIPS 140 bit, and only if that is set
// is the TLS socket source considered.
//
// The need for a kernel sockets based TLS implementation is so poll() will
// work, as it requires kernel sockets compatible input.
static constexpr auto required_features =
  tls_socket_source_implementation_features::FIPS_140_2
  | tls_socket_source_implementation_features::kernel_sockets;
tls_socket_source_ptr tls_socket_source =
  tls_socket_source_registry::default_source(required_features,
    required_features).instantiate().value();

// Create a listening TLS socket on any IP family.
listening_tls_socket_handle_ptr serversocket = tls_socket_source
  ->multiplexable_listening_socket(ip::family::any).value();

// The default is to NOT authenticate the TLS certificates of those connecting
// in with the system's TLS certificate store. If you wanted something
// different, you'd set that now using:
// serversocket->set_authentication_certificates_path()

```

```
// Bind the listening TLS socket to port 8989 on the NIC described by host
// ip::address incidentally is inspired by ASIO's class, it is very
// similar. I only made it more constexpr.
serversocket->bind(ip::make_address(host + ":8989").value()).value();

// poll() works on three spans. We deliberately have separate poll_what
// storage so we can avoid having to reset topoll's contents per poll()
vector<pollable_handle *> pollable_handles;
vector<poll_what> topoll, whatchanged;

// We will want to know about new inbound connections to our listening
// socket
pollable_handles.push_back(serversocket.get());
topoll.push_back(poll_what::is_readable);
whatchanged.push_back(poll_what::none);

// Connected socket state
struct connected_socket {
  // The connected socket
  tls_socket_handle_ptr socket;

  // The endpoint from which it connected
  ip::address endpoint;

  // The size of registered buffer actually allocated (we request system
  // page size, the DMA controller may return that or more)
  size_t rbuf_storage_length{utils::page_size()};
  size_t wbuf_storage_length{utils::page_size()};

  // These are shared ptrs to memory potentially registered with the NIC's
  // DMA controller and therefore i/o using them would be true whole system
  // zero copy
  tls_socket_handle::registered_buffer_type rbuf_storage, wbuf_storage;

  // These spans of byte and const byte index into buffer storage and get
  // updated by each i/o operation to describe what was done
  tls_socket_handle::buffer_type rbuf;
  tls_socket_handle::const_buffer_type wbuf;

  explicit connected_socket(pair<tls_socket_handle_ptr, ip::address> s)
    : socket(std::move(s.first))
    , endpoint(std::move(s.second))
    , rbuf_storage(socket->allocate_registered_buffer(rbuf_storage_length).value())
    , wbuf_storage(socket->allocate_registered_buffer(wbuf_storage_length).value())
    , rbuf(*rbuf_storage)            // fill this buffer
    , wbuf(wbuf_storage->data(), 0)  // nothing to write
    {}
};
vector<connected_socket> connected_sockets;

// Begin the processing loop
for(;;) {
  // We need to clear whatchanged before use, as it accumulates bits set.
  fill(whatchanged.begin(), whatchanged.end(), poll_what::none);

  // As with all LLFIO calls, you can set a deadline here so this call
```

```
  // times out. The default as usual is wait until infinity.
  size_t handles_changed
    = poll(whatchanged, {pollable_handles}, topoll).value();

  // Loop the polled handles looking for events changed.
  for(size_t handleidx = 0;
    handles_changed > 0 && handleidx < pollable_handles.size();
    handleidx++) {
    if(whatchanged[handleidx] == poll_what::none) {
      continue;
    }
    if(0 == handleidx) {
      // This is the listening socket, and there is a new connection to be
      // read, as listening socket defines its read operation to be
      // connected sockets and the endpoint they connected from.
      pair<tls_socket_handle_ptr, ip::address> s;
      serversocket->read({s}).value();
      connected_sockets.emplace_back(std::move(s));

      // Watch this connected socket going forth for data to read or failure
      pollable_handles.push_back(connected_sockets.back().socket.get());
      topoll.push_back(poll_what::is_readable | poll_what::is_errored);
      whatchanged.push_back(poll_what::none);
      handles_changed--;
      continue;
    }
    // There has been an event on this socket
    auto &sock = connected_sockets[handleidx - 1];
    handles_changed--;
    if(whatchanged[handleidx] & poll_what::is_errored) {
      // Force it closed and ignore it from polling going forth
      sock.socket->close().value();
      sock.rbuf_storage.reset();
      sock.wbuf_storage.reset();
      pollable_handles[handleidx] = nullptr;
    }
    if(whatchanged[handleidx] & poll_what::is_readable) {
      // Set up the buffer to fill. Note the scatter buffer list
      // based API.
      sock.rbuf = *sock.rbuf_storage;
      sock.socket->read({{&sock.rbuf, 1}}).value();
      // sock.rbuf has its size adjusted to bytes read
    }
    if(whatchanged[handleidx] & poll_what::is_writable) {
      // If this was set in topoll, it means write buffers
      // were full at some point, and we are now being told
      // there is space to write some more
      if(!sock.wbuf.empty()) {
        // Take a copy of the buffer to write, as it will be
        // modified in place with what was written
        auto b(sock.wbuf);
        sock.socket->write({{&b, 1}}).value();
        // Adjust the buffer to yet to write
        sock.wbuf = {sock.wbuf.data() + b.size(), sock.wbuf.size() - b.size() };
      }
      if(sock.wbuf.empty()) {
```

```
        // Nothing more to write, so no longer poll for this
        topoll[handleidx]&=~poll_what::is_writable;
      }
    }

    // Process buffers read and written for this socket ...
  }
}
```

Whilst `poll` is infinitely better than `select`, it has by definition linear scaling to sockets in flight, and as a result most OS kernels refuse to accept more than a few hundred inputs per syscall (we do nothing special on that, we pass to syscall `poll` exactly what is above, and return whatever error code it returns). Despite this, this simple method of multiplexing i/o on a single kernel thread will get you surprisingly far in most real world use cases for i/o multiplexing – only when you operate thousands of concurrent connections per kernel thread do scalability issues emerge, and then becomes important alternative syscall techniques such as `epoll` or `io_uring` for Linux, Grand Central Dispatch for Apple Mac OS, or IOCP and RIO for Microsoft Windows.

The reference implementation (LLFIO) supports runtime pluggable i/o multiplexers settable per handle or per thread, and its test suite proves the aforementioned maximum performance platform APIs for efficiently multiplexing huge numbers of connections per kernel thread work just fine, and at an efficiency matching or surpassing ASIO. However I do not intend to propose any of that for standardisation because:

- [P2300] *`std::execution`* lacks the extra lifecycle stage needed for (in my opinion) efficient generic stacking of async layers, of which TLS over a million connections per kernel thread is a classic example (note: In SG1/P2300 terminology, this is called 'async cleanup').
- [P2300] *`std::execution`* lacks (in my opinion) the hard guarantee needed for efficient stacking of async layers, which is never-possible dynamic memory allocation.

  > [*Note:* To add relevant detail, myself and the authors of P2300 disagree about whether dynamic memory allocation can be given hard guarantees in the present P2300 design (I say no, they say yes, and I don't think we can reach reconciliation). I would say there is consensus agreement that on some implementations, internal lists of pointers to state records would be needed, and any fixed bounds on list size would be fixed bounds on async operations in flight, and where we disagree is on how standard library implementers would handle that (I think they'll dynamically allocate on demand, like ASIO does). I should also mention that in SG1/P2300 terminology, one solution to this could be 'async allocation'. – end note]

- The i/o multiplexer abstraction is a refinement of [P2052] *Making modern C++ i/o a consistent API experience from bottom to top* which was rejected by SG4 Networking in the early 2020 WG21 meeting, so one would assume that remains WG21's opinion on that choice of design for implementing Sender-Receiver (plus, [P2300] remains an insufficient subset of the [P2052] proposal for our needs in my opinion as just aforementioned).
- I think it is very important that this proposal works perfectly on Freestanding and low end microcontrollers, where the current lack of standard networking abstraction is undoubtedly a pain point. Said systems don't need much i/o multiplexing as they tend to have a hardware

limit of two, four or eight open connections, and linear complexity multiplexing is more than plenty.

- I think launching multiple kernel threads each multiplexing a few hundred open connections gets you plenty far into performance networking if you need that and are unwilling to leave standard C++. And if you need more, chances are you'll be happy to use a platform-specific solution or a well known C++ library solution such as ASIO. For the vast majority of *standard* C++ users of networking I think linear polling is 'good enough'.

What remains an open design question is whether to standardise the coroutine API, which is identical to the blocking API except its function names are prefixed with `co_` and they return an eager (and not lazy) awaitable. This is discussed more in the open design questions section, but for now I would not propose the coroutine API either.

1.3 Example: Wrapping a third party socket implementation with TLS

```
// I want a TLS socket source which supports wrapping externally
// supplied byte_socket_handle instances
static constexpr auto required_features =
  tls_socket_source_implementation_features::supports_wrap;
tls_socket_source_ptr tls_socket_source =
  tls_socket_source_registry::default_source(required_features,
    required_features).instantiate().value();

// Create a raw kernel socket. This could also be your custom
// subclass of byte_socket_handle or listening_byte_socket_handle.
//
// byte_socket_handle and listening_byte_socket_handle default
// to your OS kernel's BSD socket implementation, but their
// implementation is completely customisable. In fact, tls_socket_ptr
// points at an unknown (i.e. ABI erased) byte_socket_handle implementation.
byte_socket_handle rawsocket
  = byte_socket_handle::byte_socket(ip::family::any).value();

listening_byte_socket_handle rawlsocket
  = listening_byte_socket_handle::listening_byte_socket(ip::family::any).value();

// Attempt to wrap the raw socket with TLS. Note the "attempt" part,
// just because a TLS implementation supports wrapping doesn't mean
// that it will wrap this specific raw socket, it may error out.
//
// Note also that no ownership of the raw socket is taken - you must
// NOT move it in memory until the TLS wrapped socket is done with it.
tls_socket_handle_ptr securesocket
  = tls_socket_source->wrap(&rawsocket).value();
listening_tls_socket_handle_ptr serversocket
  = tls_socket_source->wrap(&rawlsocket).value();
```

In order to support non-standard socket implementations, TLS socket sources may be able to wrap a third party supplied socket implementation – any implementation of `byte_socket_handle` or `listening_byte_socket_handle` which meets the API guarantees is suitable. There isn't much more to say here, other than that this facility would exist in the proposed standardisation.

2 Original design goals from before reference implementation was written

1. There must be a hard ABI boundary across which implementation detail cannot leak. To be specific, if an implementation uses Microsoft SChannel or OpenSSL then absolutely no details of that choice must permeate the ABI boundary.

   Rationale: WG21 is not in the business of specifying cryptography libraries, and it's a hole nobody rational wants to dig.

2. No imposition must be taken on the end user choice of asynchronous i/o model i.e. if the user wants to use ASIO, Qt, unifex, or any other third party async framework, this API is to enforce no requirements on that choice.

   Rationale: A majority of 'simple' use cases for networking just need to operate one or a few sockets, and don't need a full fat async framework or one which can pump millions of concurrent connections per kernel thread. Fully blocking i/o, or non-blocking multiplexed i/o with `poll` is all they need and having to master a complex async i/o framework just to pump a single socket is not a positive end user experience.

3. Whole system zero copy i/o, ultra low latency userspace TCP/IP stacks, NIC or kernel accelerated TLS and other 'fancy networking tech' ought to be easily exposable by the standard API without leaking any implementation details.

   Rationale: It is frustrating when networking implementations assume that the only networking possible is implemented by your host OS kernel, but you have a fancy Mellanox card capable of so much more. This leads to hacks such as runtime binary patching of networking syscalls into redirects. Avoiding the need for this would be valuable.

4. The design should be Freestanding capable, and suit well the kind of networking available on Arduino microcontrollers et al.

   Rationale: On very small computers your networking is typically implemented by a fixed size coprocessor capable of one, four or maybe eight TCP connections. Being able to write and debug your code on desktop, and then it would work without further effort on a microcontroller, is valuable. Also, the ability to work well with C++ exceptions globally disabled, and with no malloc available (i.e. the proposed API design never unbounded allocates memory), is valuable.

5. We should accommodate, or at least not get in the way of, implementer's proprietary networking implementation enhancements e.g. on one major implementation the only networking allowed is a proprietary secure socket running on a proprietary dynamically scaling async framework; on another major implementation there is a proprietary tight integration between their proprietary secure socket implementation, their whole system zero copy i/o framework, and their dynamic concurrency and i/o multiplexing framework.

   Rationale: Leaving freedom for platforms to innovate leaves open future standardisation opportunities.

3 Open design questions

3.1 Should we default listening TLS sockets to verifying TLS certifications of clients connecting in?

There were a fair few people in the Reddit design review rounds who very strongly felt that the default ought to be that TLS certificates are validated in both directions i.e.

1. When a client connects to a server, the client validates the TLS certificate which the server returns. If the server cannot be validated, the connection is aborted.
2. When a server sees an incoming connection, the server validates the TLS certificate which the client supplies. If the client cannot be validated, the connection is refused.

The current default is to implement number 1 but not number 2 i.e. one has to explicitly opt-in if number 2 is desired, and explicitly opt-out if number 1 is NOT desired.

Thinking this design question through, on the one hand defaulting to more security always seems wise, and on that basis was why people had strong opinions. However, there are other considerations:

- In the overwhelming number of TLS connections formed in the world today (HTTPS) the client does NOT supply a TLS certificate to the server being connected to.
- Indeed, in almost all those cases, the client *couldn't* supply a TLS certificate with a validation chain acceptable to a server even if it wanted to (e.g. client is behind a NAT, so the client's IP address would not match that on the certificate).
- Due to lack of utilisation by HTTPS, some hardware TLS offload implementations can not supply TLS certificates at all when connecting to a server.
- Kernel TLS offload implementations can offload authentication for transmission only, not receipt (i.e. they specifically optimise our current choice of defaults). Kernel TLS offload will likely get defaulted to enabled in near future releases of Linux and FreeBSD as it offers compelling performance gains.
- Exchanging certificates is not free of cost, each certificate chain which must be exchanged with a counterparty is Kilobytes in size, not only does this consume bandwidth but also adds latency as the cryptography to validate a certificate is a considerable amount of work.

On those bases I currently have the defaults the way that they are, but I would like WG21 to poll on this design question please.

3.2 Should a coroutine API be proposed based on `poll`?

Earlier in this paper I discussed my reasons for not proposing standard support for runtime pluggability of the networking proposed by this paper into P2300 the asynchronous framework currently being proposed for standardisation.

Something dropped as a consequence is the coroutine API, which has identical semantics to the blocking/non-blocking API, except that instead of sleeping the calling kernel thread for the deadline

requested, it instead sleeps the calling C++ coroutine, and during which other C++ coroutines awaiting completion of i/o can be resumed.

Whilst it is easy to say 'yes we definitely want one of those' it is more complicated to decide what to standardise:

- Do we leave it entirely up to implementations to decide how best to implement a coroutine API? This risks ad hoc standardisation of incompatible proprietary APIs.
- Do we define the coroutine API in terms of `poll` and an implementation defined choice of some escape hatch? This risks preventing later standardisation in this area.
- Do we define the coroutine API in terms of the [P2052] *Making modern C++ i/o a consistent API experience from bottom to top* abstraction currently employed by the reference implementation? This is a superset of [P2300] `std::execution`, and therefore implementations would have to extend P2300 with proprietary extensions which seems risky.
- Do we enhance [P2300] `std::execution` so it gains a sufficient feature set? This risks delaying P2300 past the 26 IS, not least because we really would need a reference implementation of this now quite novel design before we can standardise, and that is probably a man-year of work (or more if you want a production grade implementation with real world use experience).

In order to prevent obstructing later standardisation work in this area, I am currently of the opinion that it is best to not standardise the coroutine API at this time. The hope would be that P2300 is either extended or replaced in future C++ standards with something sufficiently capable to implement high performance i/o, and at that time somebody who isn't me can propose a coroutine API for standardisation.

A poll result from WG21 to see if anybody else agrees with me would be useful.

4 Acknowledgements

I would like to thank those senior members of LEWG who encouraged me to design this API and write this paper. I won't mention them by name as it's easier if I don't.

I would like to also thank the Reddit `/r/cpp` community for giving such valuable feedback to earlier drafts of this paper.

5 References

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions: Throwing values
https://wg21.link/P0709

[P1028] Douglas, Niall
SG14 `status_code` and standard `error` object
https://wg21.link/P1028

[P1031] Douglas, Niall
Low level file i/o library
https://wg21.link/P1031

[P1183] Douglas, Niall,
file_handle and mapped_file_handle
https://wg21.link/P1183

[P1861] Christensen, Alex; Bastien, JF; Herscher, Scott,
Secure Networking in C++
https://wg21.link/P1861

[P2052] Douglas, Niall,
Making modern C++ i/o a consistent API experience from bottom to top
https://wg21.link/P2052

[P2300] Dominiak, Michal; Evtushenko, Georgy; Baker, Lewis; Teodorescu, Lucian Radu; Howes, Lee; Shoop, Kirk; Garland, Michael; Niebler, Eric; Adelstein Lelbach, Bryce,
std::execution
https://wg21.link/P2300

[P2464] Voutilainen, Ville,
Ruminations on networking and executors
https://wg21.link/P2464