SG14 2026/03/11 (P4007)

Attendees:
Andrew Lumsdaine
Andrzej Krzemienski
Bryan St. Amour
Daniel Sinkin
Detlef Vollman
Dietmar Kuhl
Dmitry Arkhipov
Guy Davidson
Harold Bott
Ian Petersen
Inbal Levi
John McFarlane
Lauri Vasama
Mark Hoemmen
Matheus Izvekov
Matthew Borland
Michael Levine
Michael Vandeberg
Michael Wong
Mungo Gill
Paul Bendixen
Samuel Obeng
Steve Gerbino
Ville Voutilainen
Vinnie Falco
Vishal Ozal

Chair: Michael Wong
Scribe: Mark Hoemmen

Vinnie Falco (VF) (presenting): Slide “The Sender Sub-Language” (P4014). It shows Continuation Passing Style (CPS).

Mark Hoemmen (MH): I thought P4007 was on the agenda, not P4014.

VF: This is background for what follows, just a couple more slides.

VF: I wasn't a fan of c++20 coroutines when they landed. Peter Dimov challenged me to "Just Use Coroutines." The result was that coroutines are great for I/O.

VF: (P4007): "std::execution is fine engineering. It's ready to ship. Let's do it." What about std::execution::task.

VF: (Presenting example) async_read_array. Compares with P2300's async_read_array example. Presents poll that removed Networking TS. Shows coroutine version of this example. "This is a third direction [a 'coroutine-native approach'] that was not polled." (See Appendix A.3 in the paper.)

MH: Could you please speak more about symmetric transfer from your paper P4007?

Michael Wong (MW): That is a NB comment.

VF: P2583R3 documents the "symmetric transfer gap" that exists in the sender pipeline itself. Structurally incapable of passing the coroutine handle … earlier revisions of our paper identify the gap. Subsequent iterations proposed a fix. R3 update proposes wording to address a national body comment.

MH: Stating for the record that P2583R0 was published in Feb 2026 despite the paper number.

Vishal Oza (VO): Does this mean that std::execution is going to be delayed or is it only the std::execution::task that would be delayed for C++26?

MW: We're not saying anything is delayed right now. This is a technical discussion.

VF: std::execution is already voted in. There is a gap, it's up to the committee to decide… std::execution is good for the domain … the task is where we feel the … has some problems. C++ Alliance has built libraries based on coroutines. That's why our recommendation is to delay task.

VO: Is there a purpose for making coroutines … right now C++26 has coroutines as part of the generators; would there be an idea of putting coroutines as a package task in the language as well? Or some way to write coroutine functions without having all the mechanisms created if you're trying to use a coroutine out of the box and not a generator?

VF: I think you're asking what options for users would be for a packaged task out of the box?

VO: … rather than the generator framework?

VF: 2 cases: A user who wants to author the task, user who wants to use them. For the first, tasks use the awaitable protocol from c++20 coroutines. Many libraries implement tasks. Second: what are users' options for task types? Users can use the many libraries that already exist; they can use those types.

Ian Petersen: A central part of your concern about task design is about error handling in network scenarios, esp. Error handling. Is that the central concern?

VF: Several gaps. Allocator timing gap, ergonomics gap, "complicated success" outcome (error code plus value / partial result) . co_yield is a surprising interface for utilizing the error channel. Symmetric transfer gap [see above]. P4007 covers the problems for I/O that are very important.

IP: It feels to me like the choice of whether to represent complicated success as value or error is about the design of the … rather than the design of the task type. Dietmar's task could return a pair of error_code and result through the error channel. You can represent partial success with sender as a value of a pair like that. You listed several problems: the specific one about complicated success. Why is this networking API design challenge a reason to hold back a task that is more general than networking?

VF: (1) You're saying "just route everything through set_value." Consequence ist hat ⅔ of the sender outcomes are unused. Thus downstream pipeline senders can't use upon_error or upon_stopped. We would be shipping c++26 with a task that, when used with I/O, leaves sender algorithms structurally unreachable. (2) Idea that P2300 is UNIVERSAL, answer to everything. I think back when we had the poll and experience with it, that could have been reasonable. I like universal things. Now we have several years of evidence questioning whether it's universal. Gaps are trying to fit I/O for a model that's not tuned for it. Users aren't at our level of sophistication. They will use task and do a network call. Errors will get turned into exceptions. They will try to return an error_code. C++ doesn't ship with documentation. 6 years we've been using co_yield for the generator. Now user told to use co_yield with error. I do wish we would keep back task to address these gaps.

MW: Segue to next speaker, Dietmar Kuehl. We are not here to relitigate; we are here to address the technical domain.

Dietmar Kuehl (DK): First question: You presented a number of issues claiming there are with task. Which one was actually a problem with task? You answered some in questions. I don't think anything you presented has anything to do with task. 3 value channels – any task that operates with senders needs 3 value channels. I'm unclear how any alternative design would not do that. That ship on co_yield hasn't sailed. But which thing is a task problem?

VF: It's nice to finally meet you Dietmar. The task work has been impressive. I respect your intellectual honesty and rigor. With respect to P3…'s task, the allocator timing gap is a prboelm, because the coroutine frame allocation must be controlled. Users must be able to specify at the launch site which allocator they used.

DK: Note that you did not mention the allocation gap in your presentation.

VF: My presentation was designed to give a little flavor of the problem and not intended to be complete. (Displays header file with coroutines.) Every user who writes a coroutine declaration will need to put (std::allocator_arg_t, Allocator) in their declaration. The recycling allocator is mandatory, else C++ won't have performant I/O. Users must supply "std::allocator_arg, alloc" everywhere. Does that answer your question?

DK: No it doesn't answer my question. (Shares "Responses to the Issues": https://github.com/bemanproject/task/blob/89e367f75d1cd0cdbac0952f3e041137afbcc778/examples/task-sender.cpp#L138) For frame allocation specifically, I do agree that the approach which task supports uses a parameter list instead of thread-local storage. That does not necessarily conflate to the use side. You can create a thing that does not require passing the allocator as an argument. If you create the defer_frame thing with an argument … (presents).

VF: Look at what you wrote: co_await defer_frame. That lambda is a coroutine. Where's the allocator?

DK: This is the point. This is NOT a coroutine.

VF: You're using co_return that makes it a coroutine.

DK: It is NOT invoked; this is a coroutine factory function, not a coroutine. You need to invoke it to get the coroutine.

VF: What happens at the launch site for connect? How do we get allocator into operator new in the promise?

DK: operator new comes … read_env gets allocator from the environment.

[argues about the semantics]

VF: At the top level, the first coroutine happens at connect. Show me that part when the allcoator is propagated.

DK: The top level is a sender.

VF: That's not connect.

MW: I'll stop this back and forth.

DK: I create a sender and you connect it. There's no magic. The sender being returned is the let_value thing. That has a connect function. That gets an environment.

MW: That is the answer. Vinnie, you can come back to it.

Ville Voutilainen (VV): What's the new term for "partial success"?

VF: "complicated success"

VV: I thought I saw in your presentation that the coroutine returns a pair fo the data and the error. Is that correct/

VF: We're using structured bindings. "Auto [ec, n] = co_await sock.read(buf)"

VV: Your sock.read returns a bindable thing. Criticism with senders, if you use just value channel, you can't use upon_error algorithms. But that's the same thing with this coroutine; you can't use upon_error algorithm. The pair comes from value channel, not error channel. The same issue remains. This doesn't seem to actually avoid that problem; it has the same problem.

VF: I disagree. What follows after getting the pair is a statement "if (ec)". In senders, pipe operator separates statements. In C++, semicolon separates statements. In regular C++ I can do whatever I want here. In senders you lose the compile-time graph properties, a retry operation. If you use upon_error and retry – coroutine can never reach it. That's the problem with using set_value.

VV: If you … the same problem occurs. That's the point. It's criticizing without solving the problem. It solves the problem in the scope of that single coroutine function. But then when you compose that further, the same issue occurs again. We are going to be doing that – composing – widely using sender pipelines to deal with coroutines. We will do scheduler transfers with sender pipelines. … followed by then in order to bind a continuation to a coroutine. People won't use any other mechanism like it's currently done. … some coroutine-based-specific way to get continuation. They will use the general mechanism and that involves binding the coroutine to a sender pipeline anyway. Then the original problem you're criticizing resurfaces.

We'll have to deal with error info coming through the value channel anyway. We already deal with that. We have things in the wild where you can't make sure that errors occur on value channel. Qt … turns signals into senders. If you have a signal that reports a success and a separate signal that reports an error, and a combo signal that reports both, all of those are mapped into set_value anyway, and then you need to write subsections of your pipeline that deal with that .. and possibly transform those into completions on the value channel. … later take that "complicated success" and then use … from that point onwards into completions into separate channels. That will happen anyway in … real application code. I find the whole rationale for that criticism imprecise and the solution insufficient.

VV: I have further comments on the whole direction of a courtine-native I/O library that I'll talk about later but I'll give Inbal a chance.

VF: If you could write this analysis in an e-mail list message I could process it. What you're saying is true to some extent, but users who do I/O, they don't need to interact with std::execution; they can write a coroutine. They don't need to route through sender sublanguage.

VV: Do you have a preference for mailing list?

VF: LEWG.

Inbal Levi (IL): To respond to more general things that came up: I don't have comments on technical issue. I'll defer to experts. Implementation experience: In LEWG github, we have section dedicated to Senders. Multiple implementations here. Dietmar's is one. Eric Niebler's is another. Libunifex is an earlier version of the model. (See github.com/cplusplus/LEWG/wiki/Senders/…). Ian and experts wording in practice with stdexec (Eric's implementation). 3 channels not new. For I/O, please see Robert Leahy's Core C++ 2024 talk on networking which he is using in production. We had multiple discussions on std::execution::tasks 9 months ago. Symmetric transfer is here. Task by Dietmar implemented in beman project 9 months ago. cppreference, I wrote the documentation for std::execution, copied from the expert sources, but still. We have an issue "support symmetric transfer" which is why we will talk about this in Croydon. I hear claims that we are looking at this for the first time. Senders is not only about coroutines. Task is the glue we're trying to give to our users to use coroutines, to improve usefulness. I don't have a horse in this race, but I wanted to bring up that we've had this topic on the table in the spotlight for at least 9 months. NB comments time, starting to talk about error / value channel, not a good time to start.

VF: The reason I filed these papers late, no particular timing. I posted a video to the reflector which explains that. It was really just luck. Documentation: cppreference is informal. There is no official documentation for C++. We have to rely on the ecosystem for tutorials.

IL: This was particularly important to me. I can send print screens from 2 years ago poking sender / receiver authors on this topic. Senders are very documented in talks - that I've encouraged.

VF: You're right, it's very well documented. The C++ ecosystem doesn't have the pedagogical framework to deliver … consistently to users. WG21 puts out the Standard; they don't put out the teaching tools. P2300R10 claims to be a universal async model. P2300 is very complex, very big … the phrase "symmetric transfer" does not appear once in P2300. Even though corotuines were delivered in C++20, the phrase doesnt' appear once. That means coroutines weren't considered.

MW: They were delivered at different times as well.

IL: Senders is NOT only about coroutines. Task is. It's a useful model and framework. You're correct that it doesn't appear there because it's not mainly about coroutines.

Lauri Vasama (LV): It seems like this problem is specific to coroutines and not to any particular coroutine task type, and it's not clear to me how the coroutine native solution you showed would solve the all9ocater problem better without passing allocators manually.

VF: I wrote P4007 because we first impl'd coroutines and solved the … problem and then I wanted to "steal techniques" from Dietmar. I tried his trampoline "affine_on." … When I looked closer, that's when I discovered the gaps. The allocator timing gap is solved at with double dispatch and thread-local variable. Coroutines … don't try to invent 3 channels, a self-inflicted problem that doesn't appear in regular C++. You use co_return and co_yield as intended.

MW: The only reason I'm taking this on is because I don't want to have a 2-hour discussion in Croydon.

MH: I strongly object to the characterization of std::execution as a GPU or NVIDIA thing. More than half of the work in std::execution is not that at all. [Scribe note: It puts down the extensive contributions of non-NVIDIA contributors who are not looking into heterogeneous programming.]

VF: If you object to particular sections of the paper, please point them out.

IP: A number of claims you made I think are false, in support of arguments that I think don't follow. What is the best way to resolve this. It is interesting that you have production

experience that coroutines suit the domain of SG14, because my experience with video calling (not HFT but soft real time) is that the performance characteristics of coroutines could be a problem. Maybe the reason is that we didn't integrate with allocation. I'm interested in your discoveries. But I think we're speaking past in each other. I'm not convinced that we understand each other. I appreciate the contribution.

VF: … coroutines have gotten a bad rap. You can eliminate all allocations in the hot path and steady state if you use a recycling allocator. We implemented the feature in c++20.

MW: Ian, if you could articulate the points on paper.

IL (via chat): Please go over Robert's S/R networking talks, as an example for Networking using S/R: https://www.youtube.com/watch?v=S5v7f8rjSVQ

VV: On going coroutine native for an I/O framework expected to be scalable: I understand you can get rid of frame allocations and you don't need to do global allocation for the frame, and if you're disciplined enough, none of your coroutines throw either. But with a sender-based pipeline, it's much easier to do that, because you don't have to worry about allocators at all. Because you can trivially easily build I/O pipelines that do no allocation whatsoever. I'm firmly in the camp that believes the best allocation is no allocation whatsoever, vs. allocation that is amortized or doesn't appear in heap profiles. As far as avoidance of exceptions goes, that's not necessarily a question of how fast exceptions fly, but that the support code required for exceptions is too large for some applications. I need the discipline for both so that I don't use exceptions. The point is that coroutines aren't superior in that sense. I need the same discipline for both. As far as frame allocations go, with senders no allocations at all. For that scaling down you do need to be disciplined about how you do amortization .. whereas with sender pipeline it's easy to avoid allocations at all. The whole issue doesn't come up. I find it implausible that a coroutine-based I/O framework is the best way to go. It can be nice when you look at smaller isolated parts of code, if you're just an "I/O programmer" it may be a nicer way to do things, but then the further compositions are palatable even if they don't use the "coroutine-native" approach, but the point is that those further compositions WILL happen. In your isolated fortress of solitude, you can make sure your I/O doesn't cause problems, but if it's a sender pipeline through and through none of those issues come through anyway.

VF: This format is not very good for me because so many things were said that I'm not able to respond.

MW: VV had a point about structural guarantees vs. discipline – the strongest technical argument for Sender pipeline.

VF: OK, maybe VV is right and he can avoid allocations. However, it's a trade-off. Look at what you have to pay: everything is a template and in the header file. ABI. Can't use normal C++ technique … ABI stability with coroutines, not senders. Users need a choice….

MW: I've run out of hands. Do you want to do those polls? I don't think we should do them. I want to ask Inbal and Guy's guidance. Are they within SG14's purview?

Guy Davidson: It's up to the chair. Whether polls should be taken for max benefit to the Standard, that depends… if you want to gain direction on how to proceed, you could ask the chair of the most relevant group to gauge the opinion of the room. It looks like you're asking someone to do more work (e.g., "open design questions").

VF: I'm not asking anyone to do more work. We'll take on more work.

GD: Then you don't need to take that poll. ((3) "WG21 should explore coroutine-native I/O designs…"

VF: Cites "Networking should support only a sender…" poll.

GD: If you can improve motivation… if you can come back with better motivation.

VF: My understanding is that if I bring P4003, someone could raise a process objection because we said no in 2023.

GD: That's kind of how it works, but if you bring new information and improved motivation, it behooves us to look and see if it's indeed new information.

MW: 2023 consensus means you need strong motivation, not that the door is permanently closed.

GD: It would be foolish of us to ignore new discoveries.

VF: Should I reword the poll?

GD: No – you could do the work.

VF: You're saying I could revise the poll….

GD: You can do what you like. You have to convince the chair and the room….

GD: Poll (2) is a strange poll, because you're asking for temperature of room. It's very vague and not very direct.

IL: We WILL look at this paper in LEWG because we have open issues for tasks. There's no action item…. It would be helpful to get guidance from low latency people regarding different aspects of your proposal. If different tradeoffs … would be useful feedback for LEWG. Anything else would be discussed in LEWG.

VF: I don't how to word these things. Intention of (2) is to defer task to C++26 to resolve the issues.

IL: That would be a LEWG discussion.

MW: I agree with Inbal's comment that (2) should be deferred to Croydon.

GD: (2) is canvassing opinion. More specific is better. Don't read the room; put open design questions in your paper and present the paper. That would be the clear thing to do.

VF: "The committee should defer task to C++29."

GD: OK, that's an actual poll but I don't feel qualified to answer it.

IL: We are in NB comments time now. The way for us to pull task, is if we can't resolve the NB comments, it could be an option. By SG14 saying pull this out of the standard… however if thre are actual concerns we want to poll, like task does not handle allocation well, etc. Remind you that we already have a comment on symmetric transfer. If any technical concerns SG14 could forward to us with strong consensus….

GD: Remind everyone that we have a draft that has gone through a ballot. If you want to make changes to the paper, they need to be a response to an NB comment. If you bring a paper, it needs to come with wording that says, change the standard in this way. If you do this, the NBs have already issued their comments. After Croydon, we'll ask the NBs to vote. If substantial changes, that's a matter of dishonesty. Be careful what changes you seek to me. It sounds like you propose a significant change in comparison to the NB comments that have already be issued. We are right up against the finish line here.

IL: I was planning to reach out to VF and ask for wording change, but the wording change is just remove task. I agree with everything being said. We owe to our users the best C++ version we can have. But we are past the finish line. We don't want to scare people. Multiple discussions

here that task part of senders has been widely considered. Let's not do anything in a rush. I don't want people to take the wrong impression that those things haven't been considered.

VF: I think it's unfair to consider this as a drive-by, because for an I/O user, errors turned into thrown exceptions is a real problem. The conservative option is to defer task. What Guy and Inbal are saying, is that C++26 is about to ship and we can't make significant changes. That's rushing, "ship pressure." This has resulted in not-so-great outcomes. That sounds like something you do with a shrink-wrapped product, not something a standards body should do.

MW: That's not what they said.

VO: Regarding the Boost trading card game, I got that from the last CppCon, is there a ruleset to make it easier to understand [how to play].

VF: Message me offline.

VV: Continuing on what VF just said: Hypothetically, it's not necessarily about this particular issue, if someone finds a stinker at any point regardless of whether it's an NB comment, we have the ability to fix that stinker, even if it requires removing half of the things we've adopted in C++26 timeframe. That's not a process problem. We can fix a significant enough defect at any stage. If any expert finds a significant problem, we can deal with it. It becomes a question of what problems ARE significantly significant. Shipping pressure does exist. But we could make whatever significant changes if the problem is big enough to justify. Question of whether this set of issues rises to that level.

IL: Repeat, I did say that is a possible outcome of LEWG's discussion.

MW: It's always a judgment on the severity. Can you go back to your slide VF, to look at first poll?

VF: Context: Ideal solution would be where you have a complicated success outcome that has an error code, and then the extra data. If that could be transported safely into the sender pipeline, and you could use all 3 channels, so as not to throw out the data or do anything weird like routing through set_value. Users will want to implement their own algorithms. We want I/O to be able to route … through to those algorithms. Poll's purpose is that I/O outcome hasn't been given the first-class treatment that it deserves.

IP: I suppose we can poll this, but I think this statement is false. I think it's a technical argument. It doesn't feel like a good (?) poll. I can think of 2 ways to solve this problem. (1) Complicated success is a kind of success, communicated through set_value. You can complete successfully

many different ways through function overloading. (2) Create a struct that represents your error and data and pass that to the error channel. NO need to think of error channel as throwing exceptions; it's calling a function. There is an API design feature here, how to design a good networking API on top of S/R model, but that's not a problem with the model.

MW: I think Poll 1 is appropriate for SG14 to take. It directly affects low-latency and it provides useful signal to LEWG for concrete design tension. I will let either side position their argument. Ian has already.

VF: I think we should say Yes, that it's a design challenge, because Peter Dimov, Ian, etc. have solved the problem in a way that preserves the properties of P2300. All solutions make it so the senders downstream don't get the richness of the 3 channels.

IL: Can I jump in and show one screen that proves otherwise? Shows "epoll" slide from Core C++ 2024, Robert Leahy's "Evolving C++ Networking with Senders & Receivers [part 1]." You are suggesting that networking doesn't use those channels; that's not true. If we're taking this poll, let's make sure that the poll doesn't capture wrong technical arguments.

VF: That slides shows 2 things: set_value with events, set_error with exception pointer. That proves the case why you should say yes to Poll 1.

IL: No; set_value gives it information. I just wanted to show a slide that shows networking implementation using those channels. Please record in the minutes my concern about the phrasing of the poll that it is phrased as if the 3 channels are not used for I/O: https://youtu.be/S5v7f8rjSVQ?si=b7fPYkujOc7qOcOt&t=2163

VF: On LEWG, show me what you think the solution is, and I'll show you that I think there's a cost.

VV: Of course there are certain challenges with 3 channel and complicated result (success). Those are surmountable because you can process that in your pipeline. We actually have mechanisms for dealing with that, not in C++26 now, but I recommend we look at libunifex materialize and dematerialize algorithms. What materialize does, is it transforms a sender so it takes the separate … with a tag that tells you which one it is… split that completion. It's an example of an existing tool that significantly helps with dealing with that problem. The point is that we can provide algorithms that assist with handling that part, which tells us that we don't have a fundamental, certainly not insurmountable problem with 3-channel model, even for the complicated result, because we have similar ways of dealing with it.

VF: Why in P2300 and Robert's Leahy's adapter that every S/R … throws away the data? Why not put materialize / dematerialize in the paper?

MH: To me, the parts of this paper that aren't the 3-channel thing are far more interesting and worthy of a poll.

VF: I didn't address those. We want to defer task because [it] doesn't resolve the issues. I didn't go into the other parts of the paper because task is the pressing need. That's way more important than delivering goodies to SG14. We're about to ship a task to users that will give them a bad experience and that's the priority for me.

MW: We will take this poll. Either side may make a comment.

VF: I respect the work that went into P2300. It's really great for its domain. For I/O the complicated success doesn't fit into those 3 channels.

MW: Anyone else want to comment? Are people clear on the vote itself?

"I/O completions that carry both an error code and a byte count present a design challenge for the three-channel completion model."

SF | F | N | A | SA
8 | 2 | 0 | 2 | 7

Number of participants: 25
Author's position: SF
Outcome: No consensus

MW: I'm calling it as clear consensus either way for me. I need ⅔ either way for me. This is kind of a temperature poll. Concluding statements?

IL: I do want to say to Michael, please use the reflector as much as possible. We do have answers to at least some of the things that came up. We want to get the things we don't have answers for out of this if possible.