Executors Naming

Document #:	P2213R1
Date:	December 15, 2020
Project:	Programming Language C++
Audience:	LEWG
Authors:	Amir Kirsh <kirshamir@gmail.com> Inbal Levi <sinbal2l@gmail.com> Dan Raviv <dan.raviv@gmail.com> Ran Regev <regev.ran@gmail.com> Dvir Yitzchaki <dvirtz@gmail.com> Andrei Zissu <andrziss@gmail.com>
Contributors:	Yehezkel Bernart <yehezkelshb@gmail.com> Michael Peeri <michael.peeri@protonmail.com>
Reply-to:	Amir Kirsh <kirshamir@gmail.com>

1 Motivation

We believe that some of the names in the current Executors proposal [P0443R14] are confusing and would like to propose alternative names that, as we believe, better express the essence of the operations or entities. We have received indications from the authors of P0443 that name suggestions are welcome. The paper collects comments made by different members of the Israeli NB, on names which appear in executors proposal. Educational purposes will also be best served by names which make the most sense - this being especially important since executors clearly present a steep learning curve.

This revision is influenced by feedback from LEWG on the names suggested in revision 0. We kept only the name suggestions that had support in the discussion and moved other suggestions which did not gain LEWG's support to the Changelog section at the end of this document. Furthermore, we moved name suggestions which relate to "P1897: Towards C++23 executors: A proposal for an initial set of algorithms" to a separate proposal - P2252R0.

2 Name Suggestions

We believe that names should be clear to the users of a library, the same as they are for the initial creators of the library, and therefore propose reconsidering some of the names related to executors library, as described in the alternatives below.

2.1 operation_state

2.1.1 Rationale for the original name:

From [P1525R1]: The operation state is created when connecting a sender and a receiver. It is the object associated with, as well as holds the state of an async operation. It typically keeps the receiver alive as a data member. The operation_state concept has a single basis operation called ‘start()‘ that takes the operation state as an argument and ensures the operation is started, possibly by en queuing it for execution in some execution context. The caller of ‘start()‘ is required to keep the execution state alive for the duration of the async operation. Once one of the receiver functions has been called, the operation state can be assumed to be destroyed.

From LEWG’s meeting: The operation_state represents the storage of the state machine – would correspond to the stack frame of the invocation of the function you’re about to call. "Connect" prepares the state machine; "start" transitions to the initial state of the state machine. Invocation of a receiver is the transition of the state machine to the terminal state.

2.1.2 Proposed alternatives:

operation, packaged_operation.

From LEWG’s meeting: async_operation, execution_connection, activation_frame.

2.1.3 Rationale for the proposed alternatives:

Reasoning for “operation”:

As mentioned above, operation_state contains both the data and logic required to complete the operation, same as ‘std::vector‘ contains the data and logic required to implement a vector. In [libunfix], at the time of writing, there are 40 types implementing this concept which are called ‘operation‘, which signals it is indeed the natural name.

Reasoning for “packaged_operation” / “async_operation”:

Both names come from "future" terminology (“packaged_operation” similar to “std::packaged_task”, “async_operation” similar to “std::async”). Both names convey the message that there is an operation to be run in a certain context which is different from the current context.

One of this paper’s co-authors raised a concern about the word “async” being potentially misleading, as in implying strictly asynchronous execution. Main terminology contention point: does “async” mean **strictly** or rather **potentially** asynchronous? ‘std::async‘ seems to indicate the latter but ‘std::launch::async‘ and the optional ‘std::launch::sync‘ might indicate the former.

Reasoning for “execution_connection”:

This name was suggested in LEWG meeting. It describes the established connection. Some of the authors of this proposal feels that this name is still not descriptive enough, as the word “execution” doesn't add much for types inside “executors” namespace and the word “connection” fairly describes the connection between sender and receiver established for generating this type, but is not the essence of the type.

Reasoning for “activation_frame”:

This name was suggested in LEWG meeting, as being the activation frame of an async operation – connect the sender and receiver, and get back an activation frame for suspended operation; starting it is analogous to co-awaiting.

2.1.4 Rationale for the rejected alternatives:

Reasoning for “startable”:

It was mentioned as an historical note that ‘operation_state‘ was called or proposed to be called ‘startable‘, but the discussion about concept names was caught up in the action of whether concepts are nouns or adjectives. Some members of LEWG strongly oppose the name "startable", claiming it's non-descriptive.

2.2 ::set_done

2.2.1 Rationale for the original name:

Calling set_done (along with set_error, set_value) can be done at most once, and is determining the operation_state and passing control to the receiver. The contract between sender and receiver suggests that at least one of the operations mentioned above will be called.

According to [P1194R0] (2018): The name done is chosen for two reasons:

1. Calling it cancel may lead to confusion since it's a finished and not canceled task.

2. The Sender and Receiver concepts naturally extend to multi-value asynchronous computations, also known as streams. In the stream case, a Sender calls the receiver's value API multiple times, terminating the stream by calling done on the Receiver to let it know that no additional values are forthcoming. [Streams are a planned extension]

According to Facebook's [libunifex] documentation:

If an operation completes early because its result is no longer needed or because the higher-level goal has already been met then it is idiomatic for a sender to complete by calling the receiver's set_done() method. The set_done() method indicates that the operation did not run to completion and so may not have satisfied its post-conditions. However, it did not satisfy the post-conditions (...) typically because its result is no longer required. Consumer code (...) should be cancelled (by not executing it) and should generally propagate the cancellation result as its result (after performing any necessary cleanup). The exception to this rule is for the operation that actually requested the child operation to stop. In this case, the operation will typically handle the 'done' signal and then produce its result (eg. an 'error' or 'value').

2.2.2 Proposed alternatives:

::set_canceled, ::set_stopped, ::set_discarded, ::set_end.

2.2.3 Rationale for the proposed alternatives:

Reasoning for “set_canceled” / "set_stopped":

The name ‘set_done‘ might be misleading as it is usually called when the operation was not done but rather stopped / canceled. Our understanding is that the ‘set_done‘ name was left over from a previous version where it was allowed to be called even after a successful ‘set_value‘ call. As in the current P0443 version, it is used only to convey cancellation, it should be renamed accordingly. We realize that ‘set_done‘ can be called in case the operation was fulfilled in another context, but we believe suggested name alternatives would fit this scenario as well, while also handling actual cancellation / stop of the pipe (when the operation was not fulfilled).

The addition of the prefix ‘set_‘ as opposed to just ‘execution::stop‘ or ‘execution::cancel‘ is in order to make it conform with ‘set_value‘ and ‘set_error‘.

In libunifex, set_stopped is called as an acknowledgment of stop_requested returning true.

Reasoning for “set_discarded”:

Current design seems to indicate that calling set_done may **only** be viewed as a manifestation of a higher level goal having been satisfied, also known as serendipitous success [P1677R2]. This would clarify that the pending result is no longer required, i.e. has been discarded. Such a name would also precisely indicate the state of the sender without assuming anything about the receiver (unlike e.g. set_done).

Reasoning for “set_end”:

In LEWG it was raised that “set_stopped” and “set_canceled” may have connotations that we may want to avoid, the option of “set_end” tries to capture the action of ending the receiver for further work without saying if this action is triggered by having a result from another context or due to a cancel request.

2.3 scheduler (concept)

2.3.1 Rationale for the original name:

The scheduler concept describes a handle to an execution context that produces senders which can lazily be used at a later time, thus the name ‘scheduler‘ tries to convey that it may be used for running jobs later.

In addition, as [P0443R14] states, subsumptions of the scheduler concept are expected to add the ability to postpone or cancel execution until after some time period has elapsed, which brings this concept closer to familiar schedulers. Indeed, libunifex has a [TimedScheduler] subsumption with those abilities.

2.3.2 Proposed alternatives:

execution_context, sender_context

2.3.3 Rationale for the proposed alternatives:

The ‘scheduler’ concept does not deal with scheduling but rather storing an execution context. As stated in [P0443R14] - a scheduler is a factory for creating senders from a known execution context. A suitable alternative might wisely incorporate wording from 1.5.3 in [P0443R14], namely terms such as “execution” and “context”. We’ve deemed the word “context” as must-have, as it successfully describes both a sender factory and an environment in which senders execute. Therefore we’ve come up with two suitable alternatives. In both cases the action would be called “create_sender” and would be perceived as a natural verb when combined with either proposed noun.

execution_context - Context of execution. Has the advantage of best succinct description. It naturally creates senders since they are the executing agents.

sender_context - Opts in favor of specifying the execution agent. Constitutes an even better match with the associated verb and guarantees least astonishment in the coupling thereof.

2.4 scheduler::schedule

2.4.1 Rationale for the original name:

This is the single operation provided by the scheduler concept.

2.4.2 Proposed alternatives:

execution_context::create_sender / sender_context::create_sender

2.4.3 Rationale for the proposed alternatives:

As presented in the example at section 1.6.1 of [P0443R14], calling ‘schedule‘ will create a sender from a given execution context. Similarly to the previous section, We believe the operation name ‘schedule‘ is confusing, as it does not schedule something but rather creates a sender.

2.5 thread_pool::scheduler

2.5.1 Rationale for the original name:

Having the scheduler concept, this is the reasonable name for asking a thread pool for a scheduler allowing scheduling jobs on that thread pool.

2.5.2 Proposed alternatives:

thread_pool::execution_context, thread_pool::sender_context

2.5.3 Rationale for the proposed alternatives:

The same rationale as for renaming the scheduler concept. The function is not related to actual scheduling, it rather returns the execution context of the thread pool to be passed on for later execution of tasks in that context.

2.6 thread_pool::attach

2.6.1 Rationale for the original name:

This name was chosen as this function attaches a thread to the thread pool, so it will act as other worker threads managed by this thread pool. We believe that thread_pool::attach is a good name and raise this item only because currently [P0443R14] suggests an alternative name that we believe is misleading.

2.6.2 Proposed alternatives:

thread_pool::attach (e.g. **keep as is**)

From LEWG's meeting: thread_pool::attach_this_thread

2.6.3 Rationale for the proposed alternatives:

Reasoning for "thread_pool::attach" (keep as is):

An alternate name suggested in [P0443R14] for this function is 'join()'. We believe that the name 'attach' better describes the purpose, as the method attaches the current thread to the pool, while 'join' usually means the opposite: waiting for the object being called upon to join the current thread.

Reasoning for "thread_pool::attach_this_thread":

In the LEWG discussion 'thread_pool::attach_this_thread' was raised. It is a more verbose option, as the function doesn't get any parameter, thus making it a bit obfuscated (who is being attached?).

To conclude, it was mostly agreed that "join" doesn't make much sense for the "attach" operation and 'attach' (current name) or 'attach_this_thread' are better.

3 Changelog

3.1 Revision 0

Adds rationale for the existing names as required by LEWG.

Move the suggested names which relate to P1897 to a separate proposal - P2252R0: just, let_value, let_error

The following suggested alternatives did not gain support by LEWG, we document it below to avoid repeating the same debate in the future.

3.1.1 ::connect => ::pipe

The action is to prepare a 'pipe' of a sender and a receiver, whereas the term 'connect' expresses the technical operation but not the semantics of the operation.

Arguments against 'pipe': a lot of the executors facilities will be used with networking TS. 'connect' well describes what is being done. "Pipe" may be confused with Unix pipes, and it's not clear weather it's a verb or a noun.

3.1.2 ::submit => ::start

The ‘::submit‘ operation is same as ‘::start‘ for the user, but with ‘::connect‘ operation. We suggested overloading the same name for what we argued is, in a way, the same operation.

Arguments against: ‘submit‘ and ‘start‘ should be separate CPO’s. During the development of executors proposal there was a debate on whether operations with distinct semantics should have unique names or be overloads – e.g., single-shot set value vs. potential future multi-shot set value.

3.1.3 sender, receiver => producer, consumer

The names “producer” and “consumer” better convey the nature of executing jobs.

Arguments against: sender / receiver names do not overload terminology in the spaces of concern – e.g., concurrency space (e.g., task, thread). The paper intentionally avoided using producer / consumer to avoid clashes. producer / consumer are ambiguous on weather they are a single-shot or a communication channel. One could argue that "receiver" overloads a networking term.

sender / receiver has correspondence to push / pull programming models. They also fit from an educational perspective.

4 References

[1] P0443R14: A Unified Executors Proposal for C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html

[2] P2006R1: Eliminating heap-allocations in sender/receiver with connect()/start() as basis operations
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2006r1.pdf

[3] P1525R1: One-Way execute is a Poor Basis Operation
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1525r1.pdf

[4] P1194R0: The Compromise Executors Proposal: A lazy simplification of P0443
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1194r0.pdf

[5] P1677R2: Cancellation is serendipitous-success
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1677r2.pdf