# Issues and questions with p2300

A number of open issues identified by the BSI P2300 review group

## Kudos where kudos are due

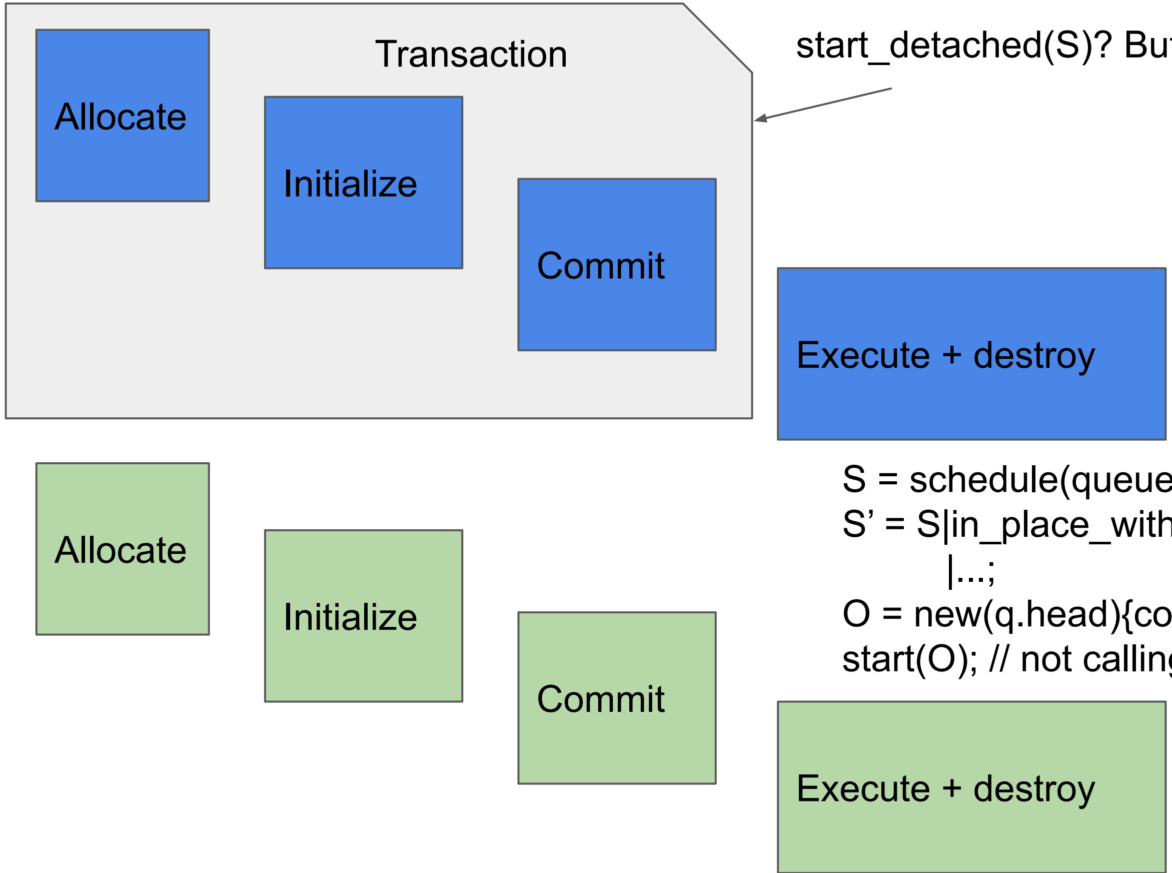
- P2300 is an awesome paper with a lot to like
  - Compositionality model is enticing
  - All-in-one-paper, and not a mess of addenda (great success)

## How should we do this?

(compendium paper is nice)

- We can have another paper salad
- Or all the authors can hash it out together and keep bringing a single paper with a list of open issues

# Consider a head-of-line blocking erase-on-execute queue



start_detached(S)? But how in primitives?

```
S = schedule(queue);
S’ = S|in_place_with_buffer(init, 512)
        |...;
O = new(q.head){connect(s, r?)}; // init(buf)
start(O); // not calling this breaks program
```

## Missing algorithms:

We have when_all(), when_any()

We also need:

- when_any_success() (ignores errors)
- when_any_error()
- when_all_successes() waits for all, but only propagates successful ones

## Stoppable_token callbacks

- Stoppable token seems to require an unlimited number of type-erased callbacks (registration can only throw if init() throws)
- Could we allow attaching a finite, algo-specific number of callbacks and replace set_done() with them? The registered callback can then transition execution contexts manually if needed.

## The kinds of cancellation

In general, we have three guarantee types when it comes to cancellation (analogous to exception safety)

- Terminal (no guarantees - valid but unspecified, you can't retry, in any case)
- Partial (you can retry if no parts of the op succeeded - partial read / weakish guarantee) - you can also resume the remainder
- Total (you can retry with impunity - similar to strong guarantee)

Might be important to understand the state of the execution context that the scheduler runs on [Gordon Brown] - mechanism to provide this kind of info is welcome

set_done(how much succeeded)

Eric: some way to query a sender which cancellation modes it supports?

Chris: reliably querying is difficult (layers of state machines - provide different cancellation guarantees)

I went with "when you request cancellation, you indicate what kind you want. That builds on P2175 - cancellation is advisory and can be ignored. I'm asking for a kind of cancellation, if you don't support it, I'm alright if you ignore it, just don't give me a cancel I didn't ask for".

## Tail calls?

The paper states: because all the work is done within callbacks invoked on the completion of an earlier sender, recursively up to the original source of computation, the compiler is able to see a chain of work described using senders as a tree of tail calls, allowing for inlining and removal of most of the sender machinery.

But set_value() can throw, and propagate up the stack until some sender redirects to set_error(). Due to this stack unwinding, are these really tail-calls? Do these tail-call optimizations depend on wrapping every set_value() in a try block?

## In general - how do we handle re-entrancy?

AsyncReader

- async_read_some() -> ar.set_value(bytes), get partial, re-register
- ar.set_value(some more bytes), re-register
- check stop_token, ar.set_done()?
- The on_done(func) algorithm completion_scheduler - how does it propagate?

## Possibly eager has additional synchronization requirements

If connect() can commit the data to the queue, we need start/completion feedback.

this_thread::sync_wait() is not easy to implement efficiently (best lock-free I can think of is a spinloop)

## Error handling

How do we distinguish between expected errors (other side hanging up) and weird errors (another thread close()'d the socket while we were read()ing it)?

Network card goes away, hotpluggable accelerator goes away

## Functionality gap: P2300 executor model

- Event-driven execution

# Current known open issues (list)

- Stoppable_token is both heavy and insufficiently general
- Operation states need a separate allocation for runtime-sized data
- Operation states can’t be allocated inside the queue due to lifetime model (right?)
- Toplevel operation states that *can* still need to return something from connect()
- “Work counting” - how do we prevent thread pools joining because of composited but not start()’d work?
- Slight inelegance in having to curry (and thus decay-copy) arguments that are consumed when starting work in all cases. Lazy/eager should be user choice but only some algos get lazy versions.
- When_all and when_any: missing wait_one_success, wait_one_error
- Partial results are results - don’t throw away work.
- Insufficient distinction between exceptions and “expected errors”
- sync_wait/fiber_wait need to be parametrized with *execution context*-specific info (how do I schedule a fiber wait?)
- Prefer tag-dispatching (set_value_t, set_error_t, set_done_t) on operator() vs three named channels - would be nice to be able to *if constexpr* between them in a lambda.
- Scheduler equality is defined as equality on execution contexts (Identified as a bug last week). If not bug, means that a threadpool with priorities presents an execution context per-priority even if runtime.

## Stoppable token composition

- Chris Kohlhoff has a (very new) signal/slottish lighter-weight cancellation mechanism that at a glance can do everything stoppable_token + set_done() can do together.
    - This will be a paper. Has the potential to replace set_done() as a full cancellation feature - but we need some time. Promise not to take long - bird in hand applies.
- Cancellation can be partial (some success has been achieved), and needs to be **pluggable by algorithms without allocation** of callback erasures.
- Cancellation over *via* and *on* may need to transition execution contexts - can’t run handler in previous exec context!

## Stoppable Token is insufficiently general

- How do we support different notions of cancellation-safety (”exception safety”) - restore state so ops can be re-run, or just leave it as-is because nobody will try to do it again? Partial reads from a socket are not unrollable, reversing partial reads from a file is a matter of resetting the cursor.
- Propose 4 levels of cancellation:
  - Terminal (never going to retry this)
  - Partial (Leave the system as “basic exception guarantee” analogue)
  - Total (As if the op was never started)

## Operation state memory issues

- operation_state objects have several memory/placement-related issues:
  - Need separate allocation for runtime-sized data
  - Can’t be allocated directly into a queue which frees memory immediately after execution
  - (both of these issues are **serious** performance blockers for line-rate UDP handling)
  - Our code deals with queues exposing std::allocator-like interfaces with tight constraints on freeing memory (no out-of-order freeing, order is set_value(), ~task(), free(), no buts.)