Document	**P3143R0**
Date	**2024/02/13**
Reply To	**Lewis Baker <lewissbaker@gmail.com>**
Audience	**LEWG**

An in-depth walk-through of the example in P3090R0

Introduction

The paper P3090R0 contains an example of some sender/receiver code and gives a brief walk through of how the example works.

This paper takes a deeper dive into the workings of the example for those people wanting more of the details of how the various senders, receivers, operation-states, schedulers, queues, etc. all fit together.

Example

The code here is a slightly modified version of the code from P3090R0, with the following changes:

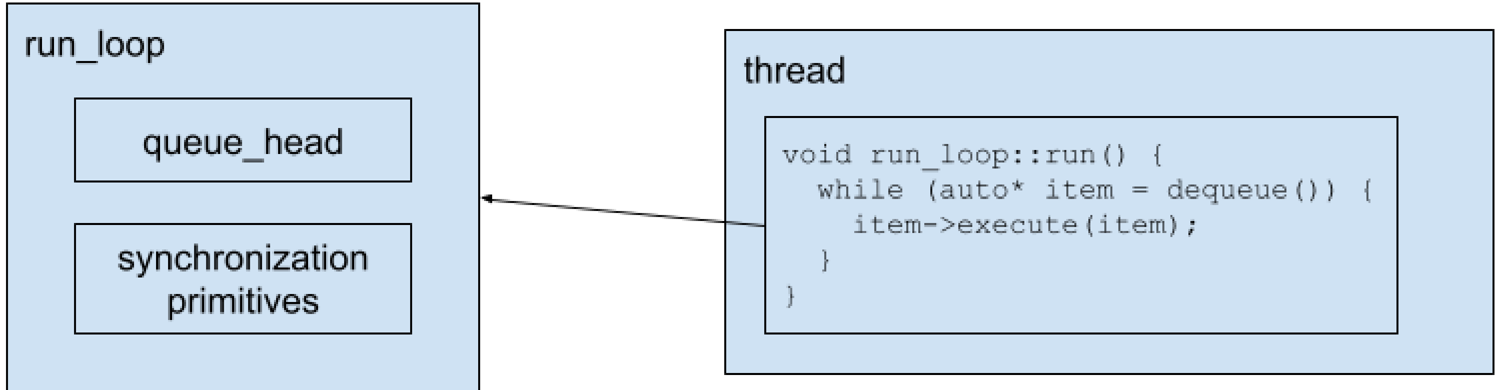
- The 'loop' and 'worker' object is moved as a local variable of 'main' instead of a global variable
- Uses `std::jthread` to automatically finish the loop and join the thread on exit, even in the presence of exceptions.
- Added `std::move()` when composing/using senders declared in previous statements to avoid copies.

Compiler Explorer: https://godbolt.org/z/4M94zGdoc

```
#include <thread>
#include <iostream>
#include <execution>
using namespace std::literals;
namespace stdex = std::execution;

int main()
{
   stdex::run_loop loop;

   std::jthread worker([&](std::stop_token st) {
      std::stop_callback cb{st, [&] { loop.finish(); }};
      loop.run();
   });

   stdex::sender auto hello = stdex::just("hello world"s);
   stdex::sender auto print = std::move(hello)
                            | stdex::then([](auto msg) {
                                 std::puts(msg.c_str());
                                 return 0; // This will be returned as the
                                           // result of the async op
                              });

   stdex::scheduler auto io_thread = loop.get_scheduler();
   stdex::sender auto work = stdex::on(io_thread, std::move(print));

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

   return result; // return 0
}
```

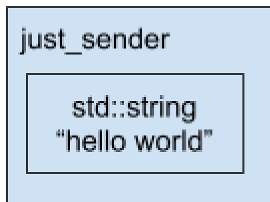
Step 1: Create the run_loop and launch background thread to execute items

This initializes the ‘`loop`’ and ‘`worker`’ variables.



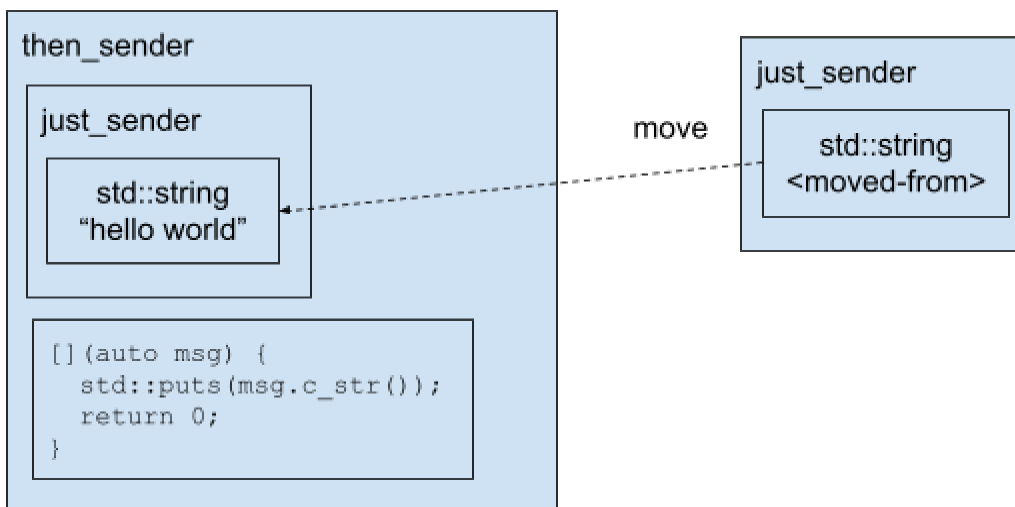
Step 2: Create just-sender

This is the local variable ‘`hello`’



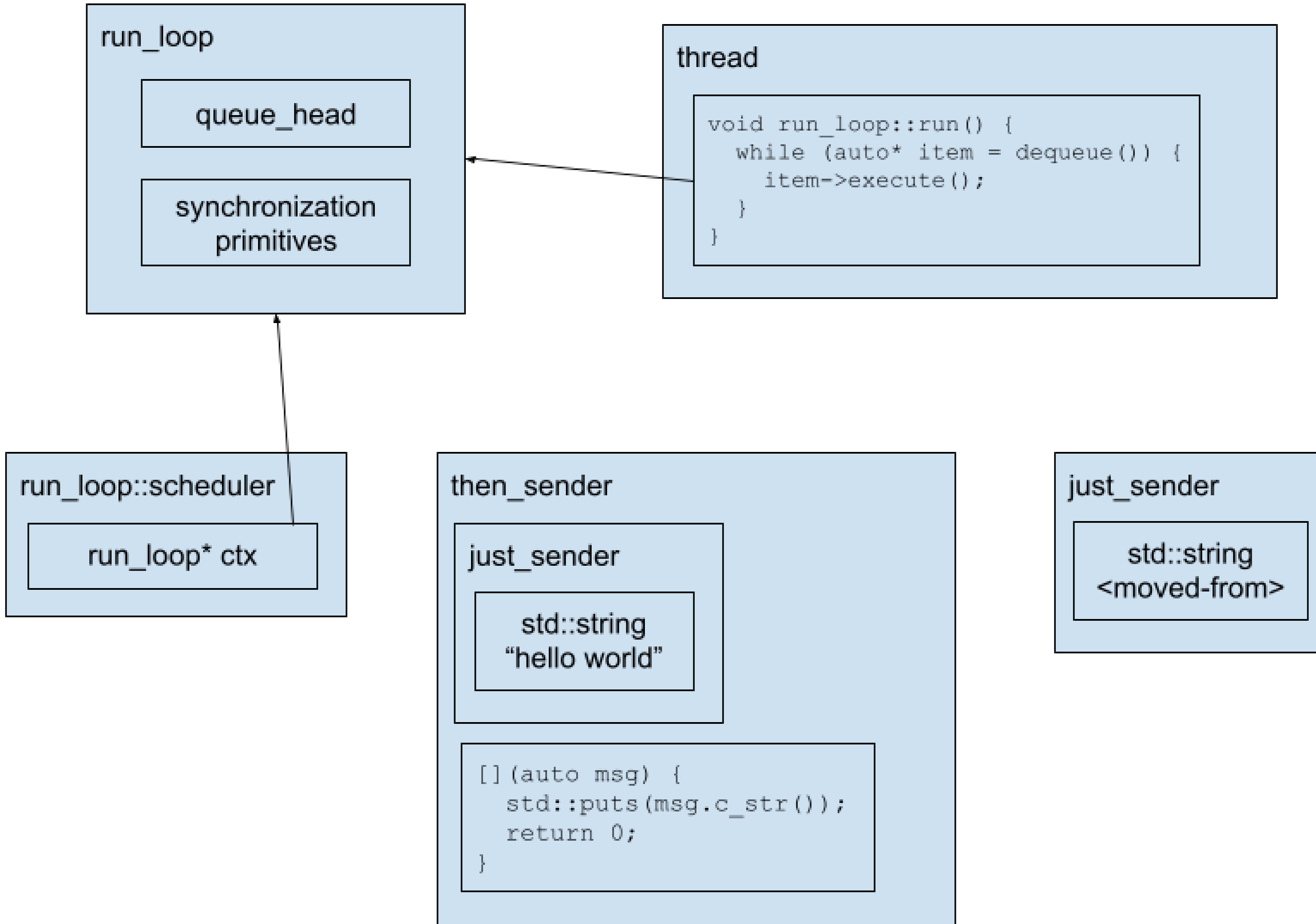
Step 3: Create then-sender

This is the local variable ‘`print`’.



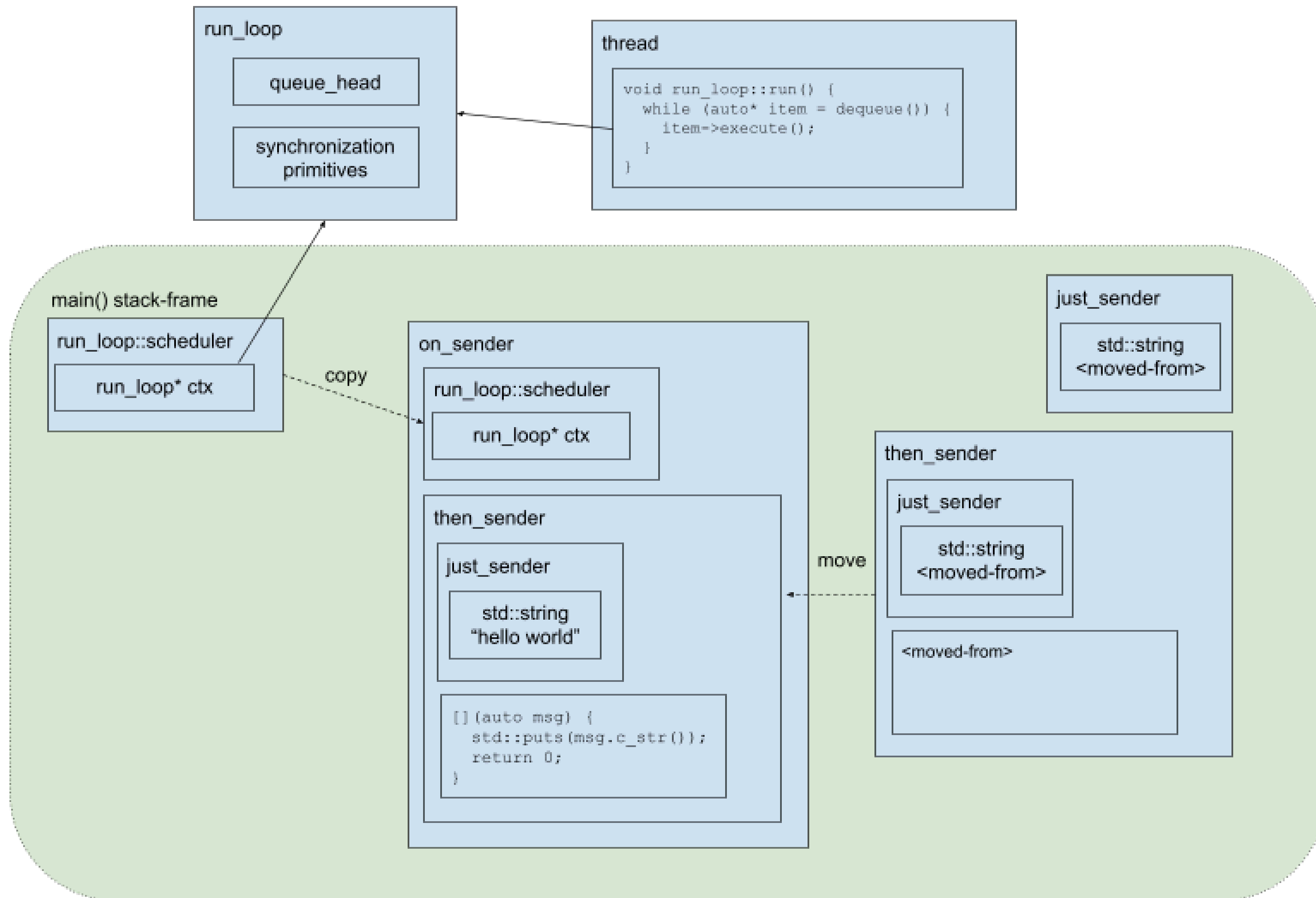
Step 4: Get run_loop scheduler

This initializes the local variable ‘`io_thread`’.



Step 5: Create on-sender

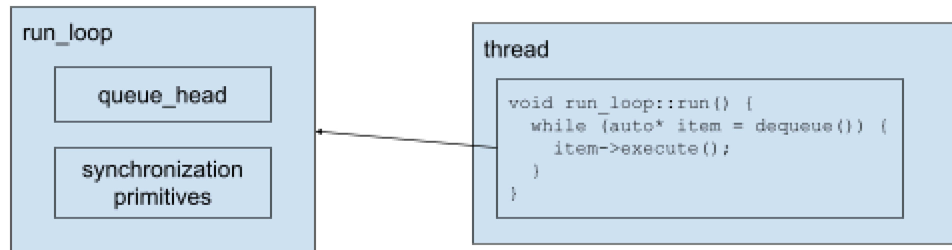
This initializes the local variable ‘work’.

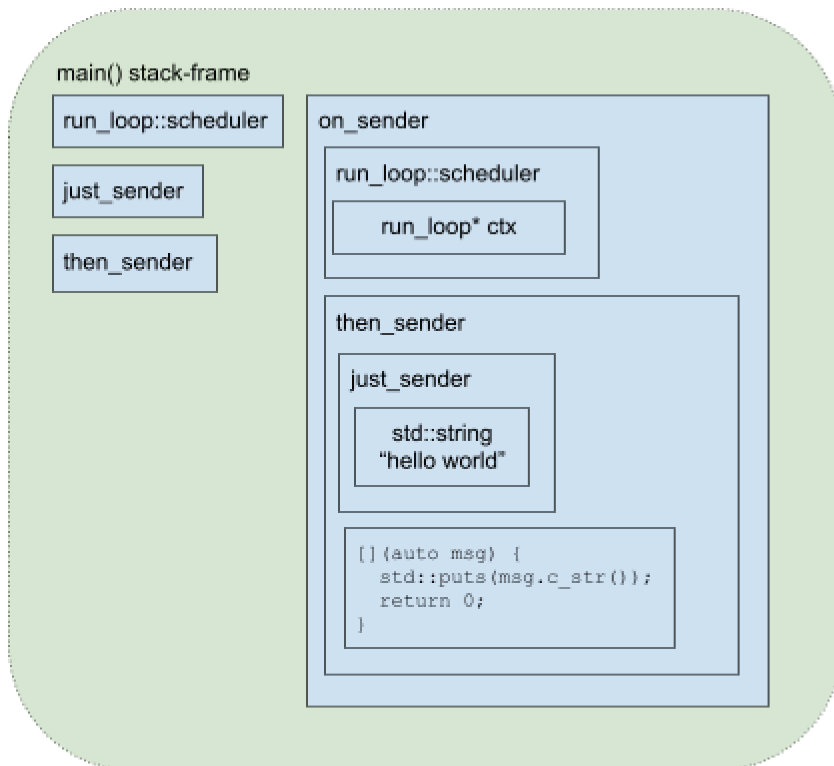


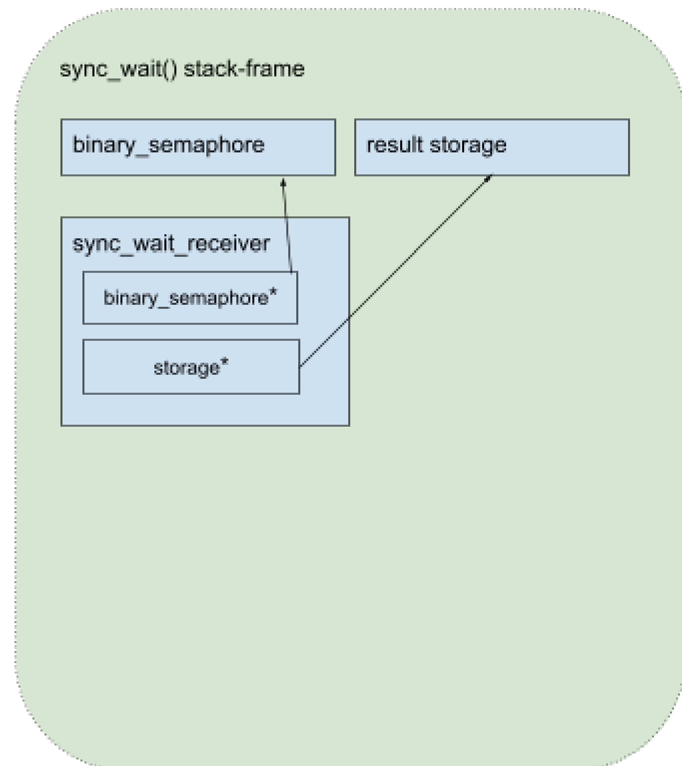
Step 6: Call sync_wait

Internally, `sync_wait()` initializes some synchronization primitives and storage for the eventual result.

It then constructs a receiver of type `sync_wait_receiver` that has a reference to these.



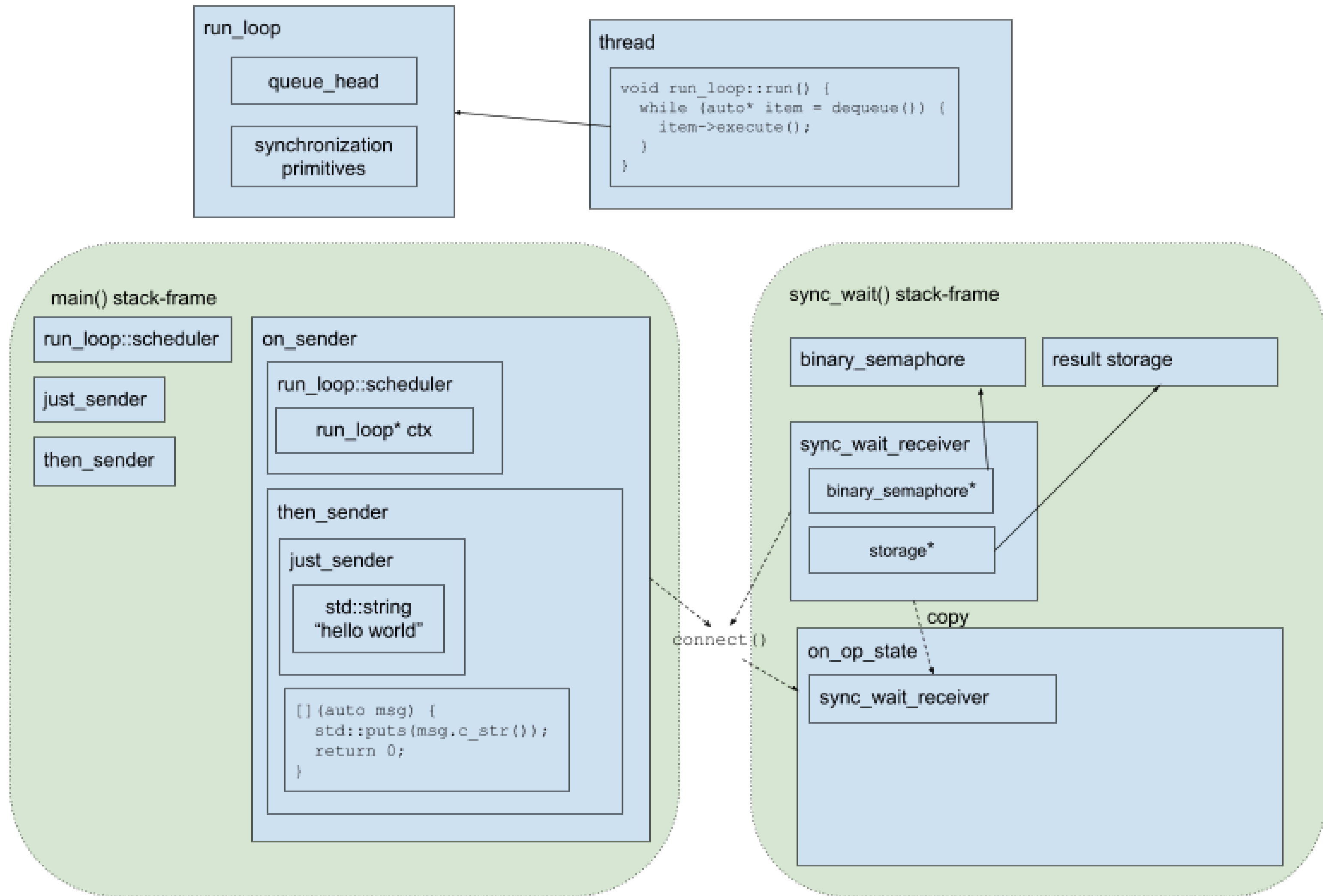




Step 7: sync_wait connects the sender

Part 7a

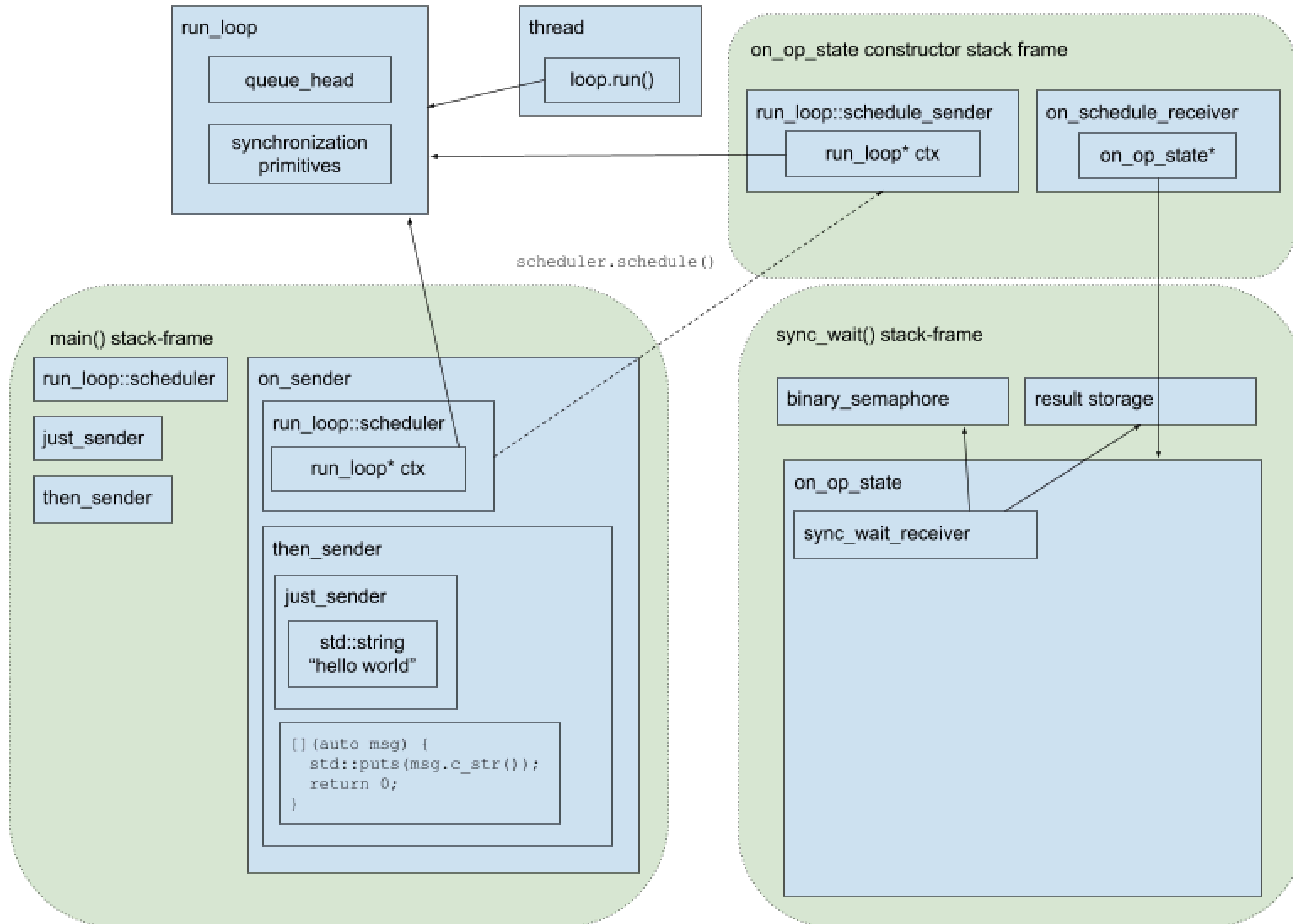
Then connects this receiver with the on_sender, creating an operation-state, which is stored as a local variable within sync_wait()



Part 7b

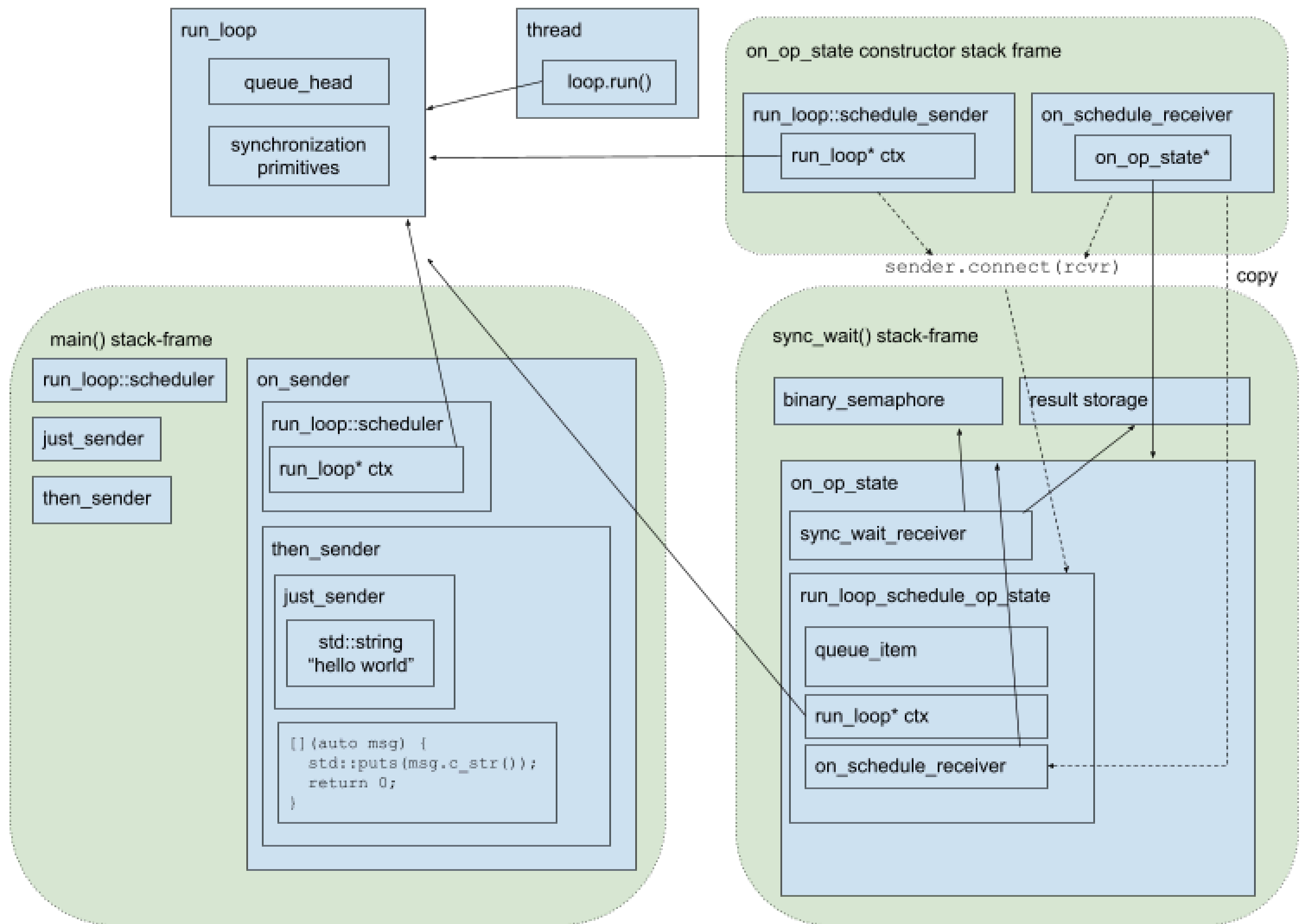
The initialization of `on_op_state` then needs to construct a child op-state for the scheduler's schedule operation in order to transfer execution to the `run_loop` scheduler.

First the `on_op_state` constructor obtains a schedule-sender by calling `schedule()` on the `run_loop`'s scheduler.
Then it creates an `on_schedule_receiver` that holds a pointer to the `on_op_state` that will handle the completion of the schedule operation. These objects are just temporaries on the `on_op_state` constructor stack frame.



Part 7c

The `on_op_state` constructor then calls `.connect()` on the `run_loop::schedule_sender` and uses copy-elision to initialize the result of the call to `connect()` in-place in a data-member of the `on_op_state` object.

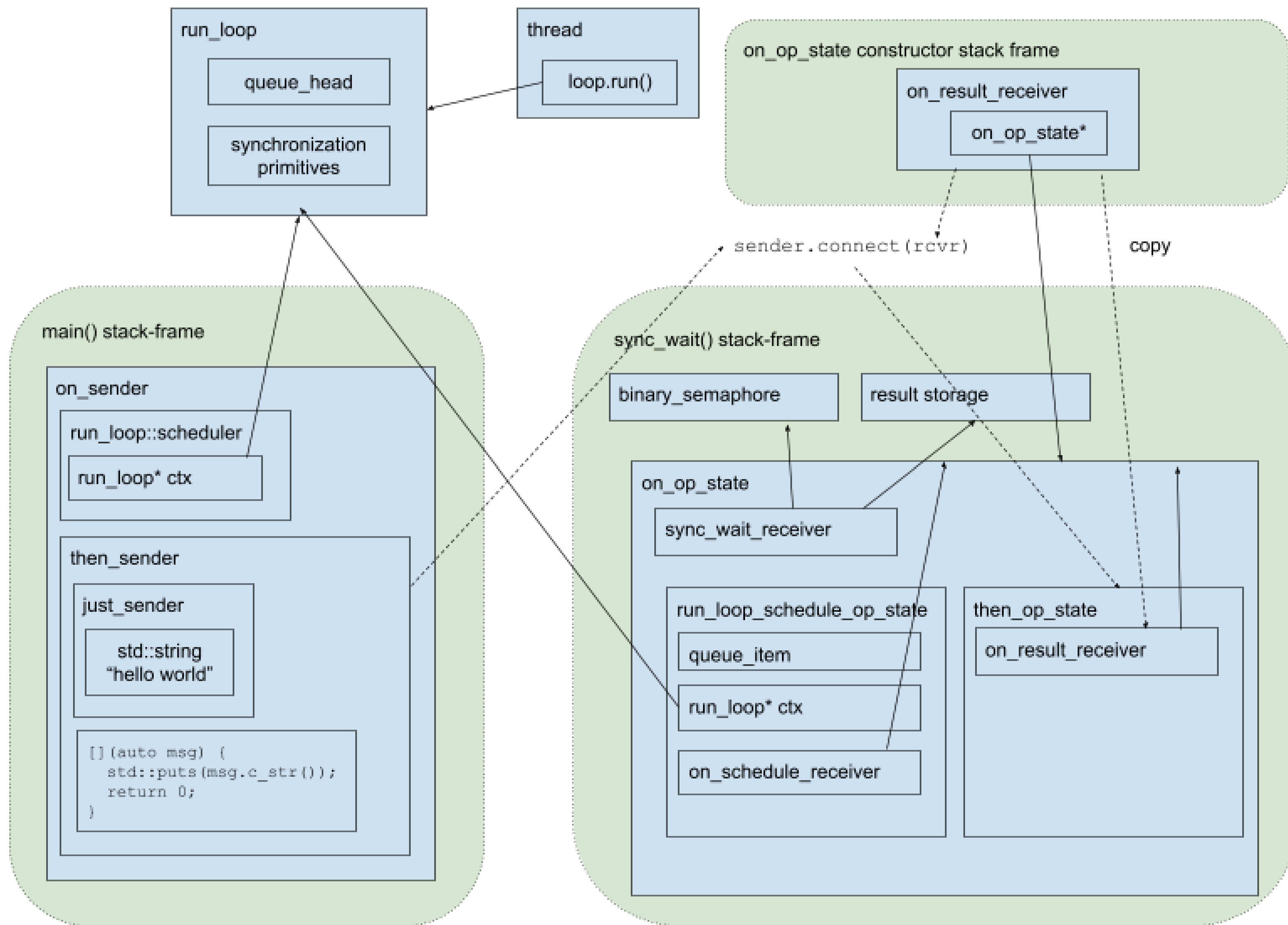


Note that the `run_loop_schedule_op_state` includes storage of a `queue_item` sub-object - this will be used by the `run_loop` queue to implement an intrusive list of queue items. This will be covered later.

Part 7d

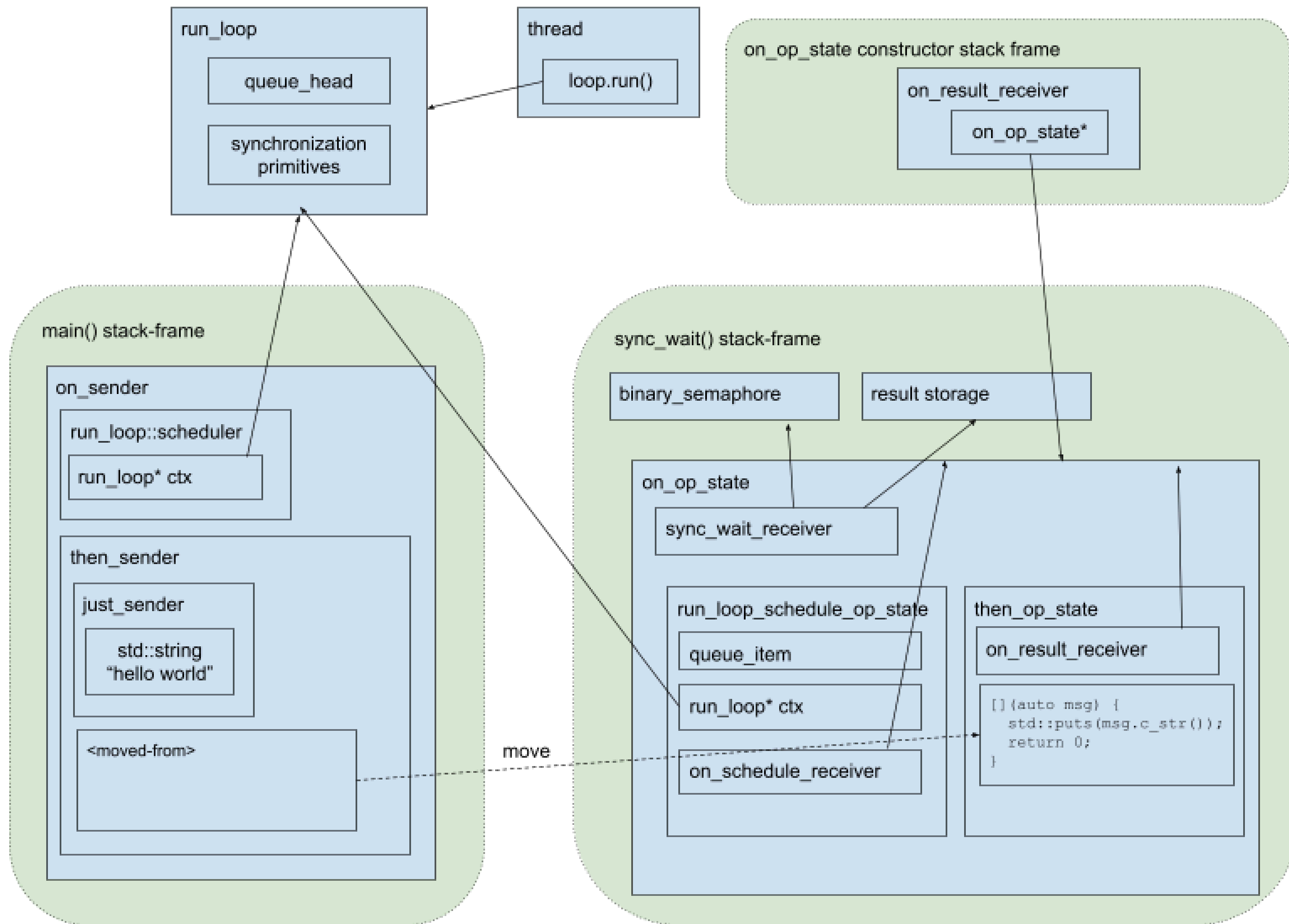
Next, the `on_op_state` constructor creates another receiver, an `on_result_receiver`, this time to receive the result of the `on_sender`'s child operation - in this case a `then_sender`.

It then calls `connect()` on the then_sender, passing the on_result_receiver, creating a then_op_state sub-object of on_op_state.
The then_op_state holds a copy of the on_result_receiver, which contains a pointer back to the on_op_state object.



Part 7e

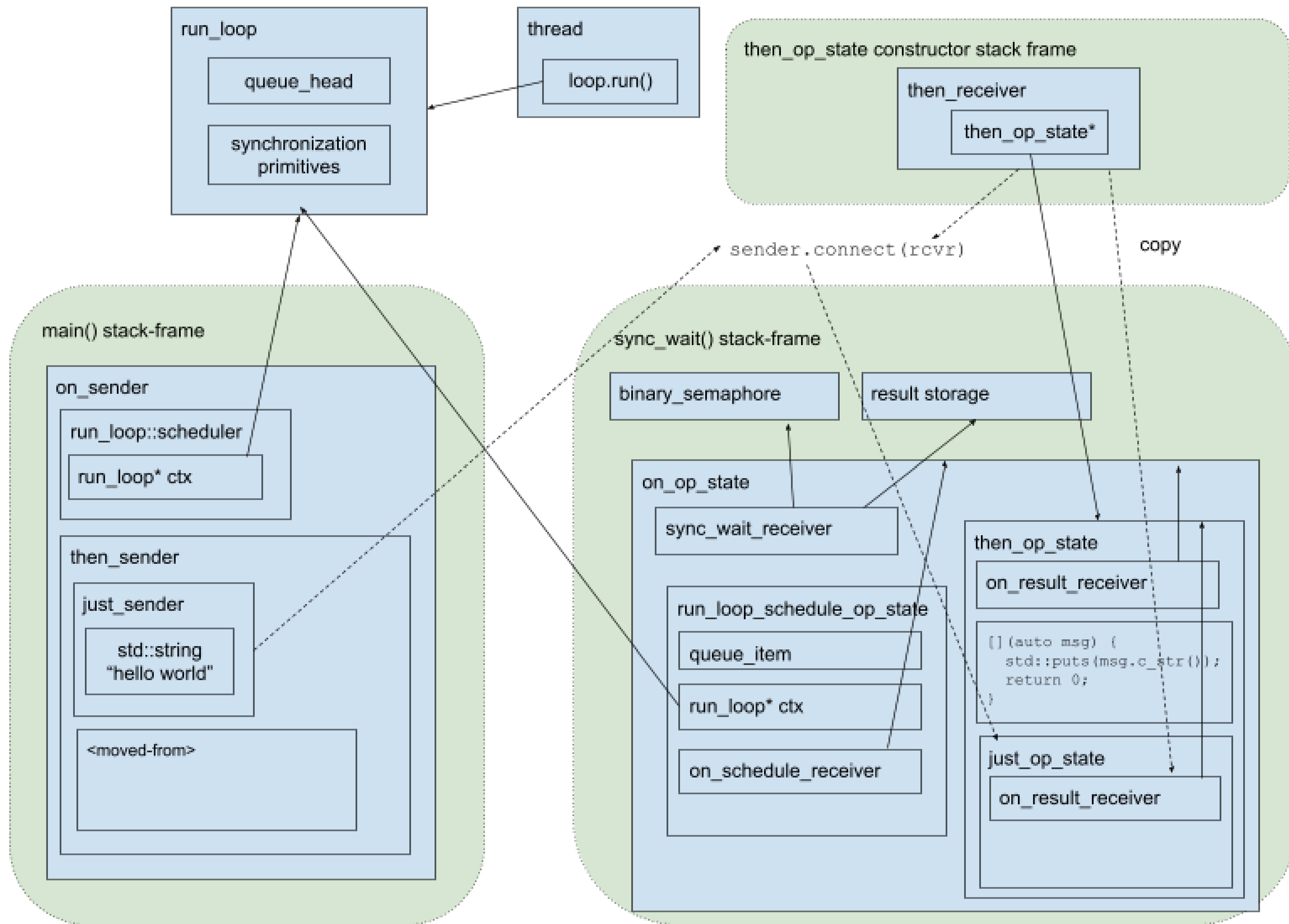
Next the `then_op_state` constructor then moves the function object (in this case the printing lambda) into the `then_op_state`



Part 7f

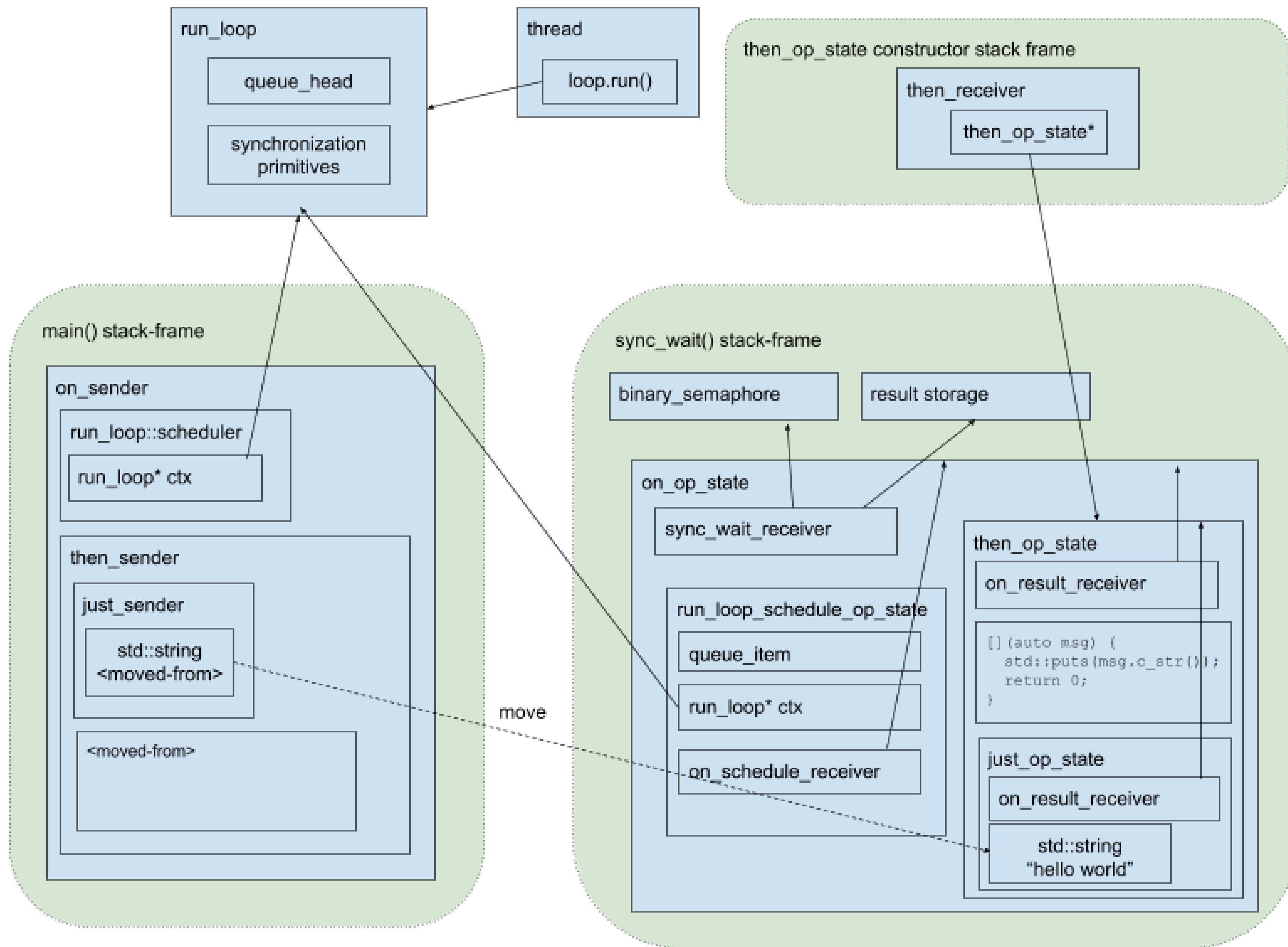
The `then_op_state` then creates a new `then_receiver` object, which contains a pointer to the `then_op_state`, and then passes this receiver to a call to `connect()` on the `then_sender`'s inner sender - in this case a `just_sender`.

This results in creating a new `just_op_state` sub-object of the `then_op_state` object.



Part 7g

Finally, the just_op_state constructor moves the value from the just_sender into the just_op_state.



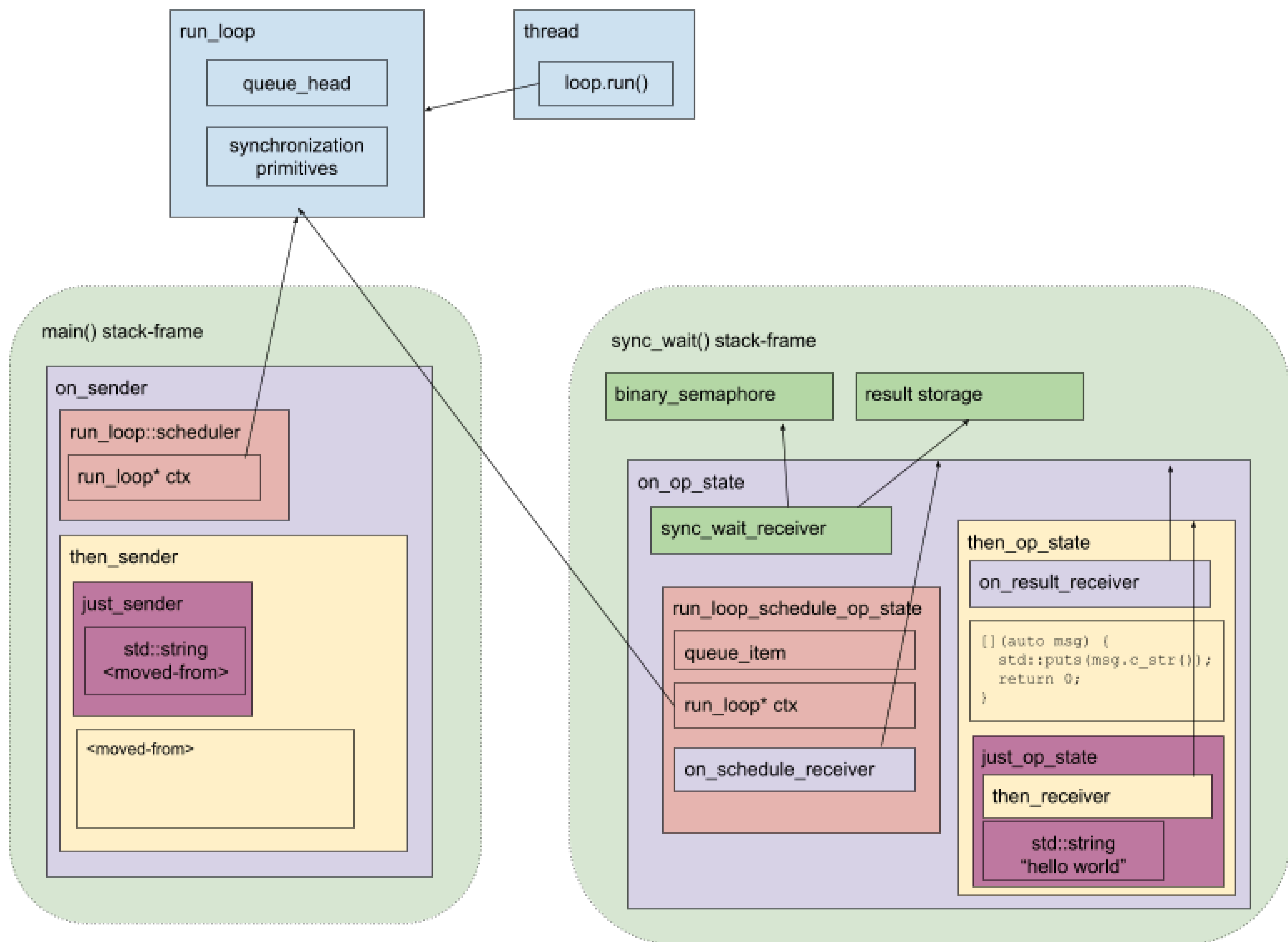
And this concludes the call to connect() from sync_wait().
There is now a fully-initialized operation-state object sitting as a local variable of the sync_wait() stack frame.

Stepping back now connect() is done

Stepping back, let’s now look at what we’ve just done.

We have taken a sender expression tree on the left, where child objects don’t know anything about the parent object within which they are contained, and now constructed a hierarchy of operation-state objects that mirrors the structure of the sender expression tree, but now where child operation-states have a link to the parent operation states via the receiver passed by the parent operation-state when initializing the child operation states.

This becomes clearer if we color-code some of the boxes.



Of course, now that everything has been connected, the original sender is left with just an empty-shell and is no longer needed.

Step 8: starting the operation

The next thing that sync_wait() does, now that the operation-state has been initialized, is to start the operation.

It does this by calling start() on the on_op_state object.

The implementation of on_op_state::start() then calls run_loop_schedule_op_state::start() and this is where things "start" to get interesting.

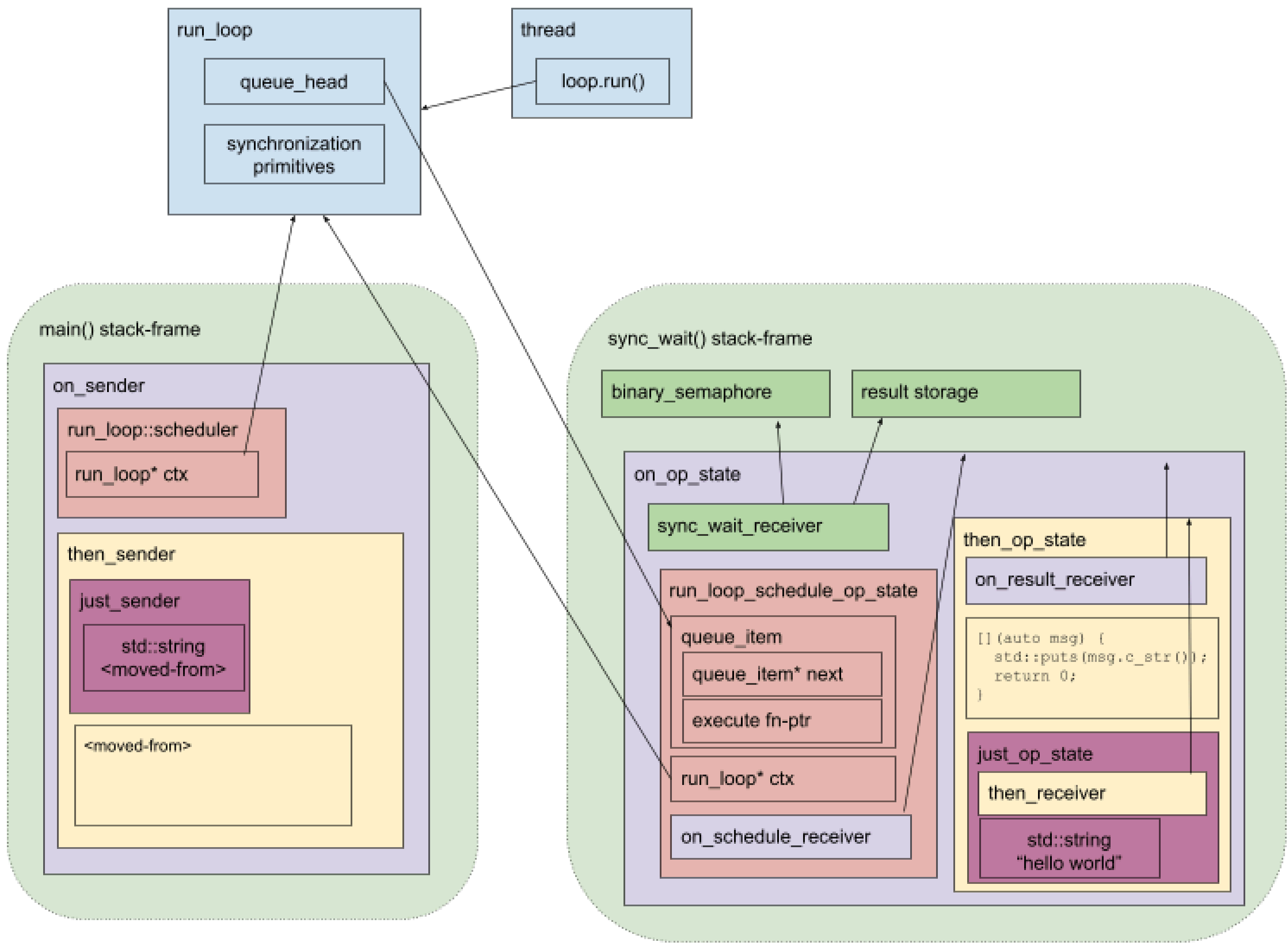
The run_loop_schedule_op::start() function needs to enqueue the operation-state to the run_loop object's queue.

To do this without any allocations it makes use of the fact that it was able to reserve some storage in the run_loop_schedule_op_state structure for a queue_item object and stashes some pointers in there to implement an intrusive list of queue-items.

The queue_item structure will look something like this:

```
struct queue_item {
  queue_item* next;
  void(*execute)(queue_item* item) noexcept;
};
```

After enqueueing the item, the run_loop::queue_head pointer will contain a pointer to the queue_item from the run_loop_schedule_op_state object.



The enqueue operation then notifies the thread executing loop.run() to wake up if necessary so that it can dequeue an item from the now empty queue.

If there were existing items in the queue then the 'next' pointer would be updated to point to other items.

Step 9: Waiting for the result

The sync_wait() function now just needs to wait for the result to be available and so blocks on the binary_semaphore with a call to `semaphore.acquire()`.

Note that the actual sync_wait implementation drives an event-loop while waiting so that it can use the current thread to make forward progress on the operation completing, however for simplicity we just assume that sync_wait() blocks in this example, as the event loop is not required for things to work.

Step 10: run_loop executes the queue_item

Now the thread we launched to execute loop.run() will wake up, see there is an item in the queue, dequeue it and then call the execute() function pointer, passing the queue_item pointer itself as the parameter.

The call to execute() dispatches to a function that casts the queue_item pointer back to a run_loop_schedule_op_state pointer (doing this safely without UB by making sure the op_state type inherits privately from queue_item) and then calls set_value() on the run_loop_schedule_op_state's receiver - in this case the on_schedule_receiver.

```
template<typename Rcvr>
void run_loop_schedule_op_state<Rcvr>::execute_impl(run_loop::queue_item* item) noexcept {
  auto& self = *static_cast<run_loop_schedule_op_state*>(item);
  std::move(self.receiver).set_value();
}
```

Calling `set_value()` here indicates that the schedule operation is complete - a schedule operation completes when the item is dequeued by some thread associated with the scheduler's execution context. i.e. a thread has scheduled the work successfully.

Step 11: on_schedule_receiver starts the wrapped operation

The on_schedule_receiver::set_value() method is implemented to start the wrapped operation - now that the schedule operation has completed and we know we are executing on the right execution context.

In this case, the wrapped operation is the then_op_state object and so it will call start() on this object.

```
template<typename Rcvr, typename Scheduler, typename Sndr>
void on_op_state<Rcvr, Scheduler, Sndr>::on_schedule_receiver::set_value() noexcept {
  this->result_op->start(); // calls then_op_state::start()
}
```

The then_op_state::start() function then in turn calls start() on its wrapped operation - in this case, the just_op_state object.

```
template<typename Rcvr, typename Func, typename Sndr>
void then_op_state<Rcvr, Func, Sndr>::start() noexcept {
  inner_op_state.start(); // calls just_op_state::start()
}
```

Step 12: just_op_state start()

The implementation of the just_op_state::start() function completes synchronously with the value - in this case the std::string containing the value “hello world” - by passing the value to the call to set_value() on the op-state’s receiver.

```
template<typename Rcvr, typename Value>
void just_op_state<Rcvr, Value>::start() noexcept {
  std::move(this->receiver).set_value(std::move(this->value));
}
```

The receiver in this case is the then_receiver provided to it by the then_op_state.

Step 13: then_receiver::set_value()

The implementation of then_receiver::set_value() then invokes the stored function object (in this case our printing lambda) with the value passed to set_value().

This lambda then invokes std::puts() with the string “hello world” and the string is output to stdout.

The lambda then returns the ‘int’ value 0 and control returns to then_receiver::set_value() which then invokes set_value() on its receiver (in this case the on_result_receiver), passing the return value of the lambda as the value argument.

If the call to the lambda were to throw an exception then we would call set_error() on the receiver instead.

```
template<typename Rcvr, typename Func, typename Sndr>
template<typename... Values>
void then_op_state<Rcvr, Func, Sndr>::then_receiver::set_value(
    Values&&... values) noexcept {
  try {
    std::move(this->op_state->receiver).set_value(
      this->op-state->func(std::forward<Values>(values)...);
  } catch (...) {
    std::move(this->op_state->receiver).set_error(std::current_exception());
  }
}
```

Step 14: on_result_receiver::set_value()

Now the on_result_receiver::set_value() is invoked and so the on_op_state operation itself has now completed and so just forwards the result to its receiver - in this case the sync_wait_receiver - by calling its set_value() method.

```
template<typename Rcvr, typename Scheduler, typename Sndr>
template<typename... Values>
void on_op_state<Rcvr, Func, Sndr>::on_result_receiver::set_value(
    Values&&... values) noexcept {
  std::move(this->op_state->receiver).set_value(
    std::forward<Values>(values)...);
}
```

Step 15: sync_wait_receiver::set_value()

Now we finally get to the end of the chain and the sync_wait operation has its result.

However, the result is provided to it on another thread (the background thread executing run_loop::run()) and so we need to stash the result in the storage reserved on the stack-frame of sync_wait (which the receiver conveniently contains a pointer to) and then signals that the result is available by calling semaphore.release() on the binary_semaphore also stored on the sync_wait stack-frame (which the receiver also, conveniently, has a pointer to).

Once this is done, the sync_wait_receiver's job is done, and it returns from set_value(), unwinding the stack all the way back to return to the loop inside the background thread's run_loop::run() call.

Step 16: sync_wait wakes up

The call to `semaphore.release()` above has now unblocked the sync_wait thread which now returns from its call to `semaphore.acquire()`. It knows that there is now a result stored in the storage that it reserved on the stack and so inspects the storage (typically a variant of the possible results) to obtain the result and then simply returns this value as the result of the `sync_wait()` call.

The return-type of `sync_wait()` is an optional (the empty optional corresponds to the set_stopped() completion result), so the result is wrapped up in an optional<int>.

As part of the return from the sync_wait() function, the operation-state object that was stored in a local variable now goes out of scope and is destroyed, releasing all of the resources used by the operation-state.

Step 17: Execution now returns to main()

The `sync_wait()` function now returns to main, which unwraps the result, obtaining the value 0 that was returned by the lambda.

This then calls `run_loop::finish()`, signaling that the background thread calling `run_loop::run()` should now return from run() and let the thread run to completion.

It then waits for the thread to finish before returning the integer result from `main()`.