Asynchronous Value Sequences

Draft Proposal

Document #:	D0000R0
Date:	2024-09-26
Project:	Programming Language C++
Audience:	SG1 - parallelism and concurrency

Reply-to: Kirk Shoop
<kirk.shoop@gmail.com>
Maikel Nadolski
<maikel.nadolski@gmail.com>

Contents

1 Introduction **2**

2 Examples **2**
2.1 Basic polling . . . 2
2.2 Bulk processing . . . 2
2.3 Web Requests . . . 3
2.3.1 Connect to a firehose and parse . . . 3
2.3.2 Satisfying Web Service Retry contract . . . 3
2.4 User Interface . . . 3
2.4.1 Build a set of reducers . . . 3
2.4.2 Apply a set of reducers to a model . . . 4
2.4.3 Build a set of renderers . . . 4
2.4.4 Apply a set of renderers to a model . . . 5
2.5 Async Resources . . . 5

3 Design **5**
3.1 Consuming a Sequence . . . 6
3.2 Sequence Receiver . . . 6
3.3 Sequence Sender . . . 7
3.4 Sequences . . . 8
3.4.1 Lock-Step . . . 8
3.4.2 External Event . . . 9
3.4.3 Parallelism . . . 10

4 Algorithms **11**
4.1 then_each . . . 11
4.2 filter_each . . . 12
4.3 take_while . . . 12
4.4 distinct . . . 12
4.5 ignore_all . . . 12
4.6 generate_each . . . 13
4.7 iotas . . . 13
4.8 fork . . . 13
4.9 merge_each . . . 14
4.10 scan_each . . . 14
4.11 sample_all . . . 15

4.12 timeout_each 15

5 References **15**

1 Introduction

This was the end goal all along.

`std::execution`, as described in [P2300R5], introduced the sender and receiver concepts which represent and compose asynchronous operations. These abstractions provide a powerful mechanism for building composable and efficient asynchronous code.

Currently, by employing a range-v3 generator, a `range<Sender>` can represent a group of asynchronous values that generate each sender synchronously. Nevertheless, it is unable to represent potentially infinite sequences of values that might arrive in parallel. Thus, this proposal proposes to broaden the existing sender and receiver concepts by incorporating the sequence sender concept. It is a sender that emits a sequence of values over time.

This paper also elaborates on some of the algorithms that operate on sequence senders. Among the features provided by this design are back-pressure, which decelerates chatty producers, no-allocations by default, and parallel value senders.

2 Examples

2.1 Basic polling

Polling works well for sampling sensors.

This particular example is simple, but has limited use as a general pattern. User interactions are better represented with events.

```
sync_wait(generate_each(&::getchar) |
  take_while([](char v) { return v != '0'; }) |
  filter_each(std::not_fn(&::isdigit)) |
  then_each(&::toupper) |
  then_each([](char v) { std::cout << v; }) |
  ignore_all());
```

2.2 Bulk processing

```
auto counters = std::map<std::thread::id, std::ptrdiff_t>{};
sync_wait(itoas(1, 3000000) |
  on_each(pool, fork([](sender auto forked){
    return forked | then_each([](int v){
      return std::this_thread::get_id();
    });
  })) |
  then_each([&counters](std::thread::id tid) {
    ++counters[tid];
  }) |
  ignore_all() |
  then([&counters](){
    for(auto [tid, c] : counters){
      std::print("{} : {}\n", tid, c);
    }
  }));
```

2.3 Web Requests

This is 'ported' from a twitter application.

```
auto requesttwitterstream = twitter_stream_reconnection(
  defer_construction([=](){
    auto url = oauth2SignUrl("https://stream.twitter...");
    return http.create(http_request{url, method, {}, {}}) |
      then_each([](http_response r){
        return r.body.chunks;
      }) |
      merge_each();
  }));
```

2.3.1 Connect to a firehose and parse

```
struct Tweet;

auto tweets = requesttwitterstream |
  parsetweets(poolthread) |
  publish_all(); // share
```

2.3.2 Satisfying Web Service Retry contract

```
auto twitter_stream_reconnection =
  return [=](auto sender chunks){
    return chunks |
      timeout_each(90s, tweetthread) |
      upon_error([=](exception_ptr ep) {
        try {
          rethrow_exception(ep);
        } catch (const http_exception& ex) {
          return twitterRetryAfterHttp(ex);
        } catch (const timeout_error& ex) {
          return empty<string>();
        }
        return error<string>(ep);
      }) |
      repeat_always();
  };
```

2.4 User Interface

This is 'ported' from a twitter application.

2.4.1 Build a set of reducers

```
struct Model;
using Reducer = std::function<Model(Model&)>;

vector<any_sender<Reducer>> reducers;

// produce side-effect of dumping text to terminal
reducers.push_back(
```

```
  tweets |
  then_each([](const Tweet& tweet) -> Reducer {
    return [=](Model&& model) -> Model {
      auto text = tweet.dump();
      cout << text << "\r\n";
      return std::move(model);
    };
  });

// group tweets, by the timestamp_ms value
reducers.push_back(
  tweets |
  then_each([](const Tweet& tweet) -> Reducer {
    return [=](Model&& model) -> Model {
        auto ts = timestamp_ms(tweet);
        update_counts_at(model.counts_by_timestamp, ts);
        return std::move(model);
    };
  });

// group tweets, by the local time that they arrived
reducers.push_back(
  tweets |
  then_each([](const Tweet& tweet) -> Reducer {
    return [](Model&& model) -> Model {
        update_counts_at(model.counts_by_arrival, system_clock::now());
        return std::move(model);
    };
  });
```

2.4.2 Apply a set of reducers to a model

```
// merge many sequences of reducers
// into one sequence of reducers
// and order them in the uithread
auto actions = on(uithread, iterate(reducers) | merge_each());

auto models = actions |
  // when each reducer arrives
  // apply the reducer to the Model
  scan_each(Model{}, [=](Model&& m, Reducer rdc){
    auto newModel = rdc(std::move(m));
    return newModel;
  }) |
  // every 200ms emit the latest Model
  sample_all(200ms) |
  publish_all(); // share
```

2.4.3 Build a set of renderers

```
vector<any_sender<void>> renderers;

auto on_draw = screen.when_render(just()) |
```

```
    with_latest_from(models);

renderers.push_back(
  on_draw |
  then_each(render_tweets_window));

renderers.push_back(
  on_draw |
  then_each(render_counts_window));
```

2.4.4 Apply a set of renderers to a model

```
async_scope scope;
scope.spawn(iterate(renderers) |
  merge_each() |
  ignore_all());

ui.loop();
scope.request_stop();
sync_wait(scope.on_empty());
```

2.5 Async Resources

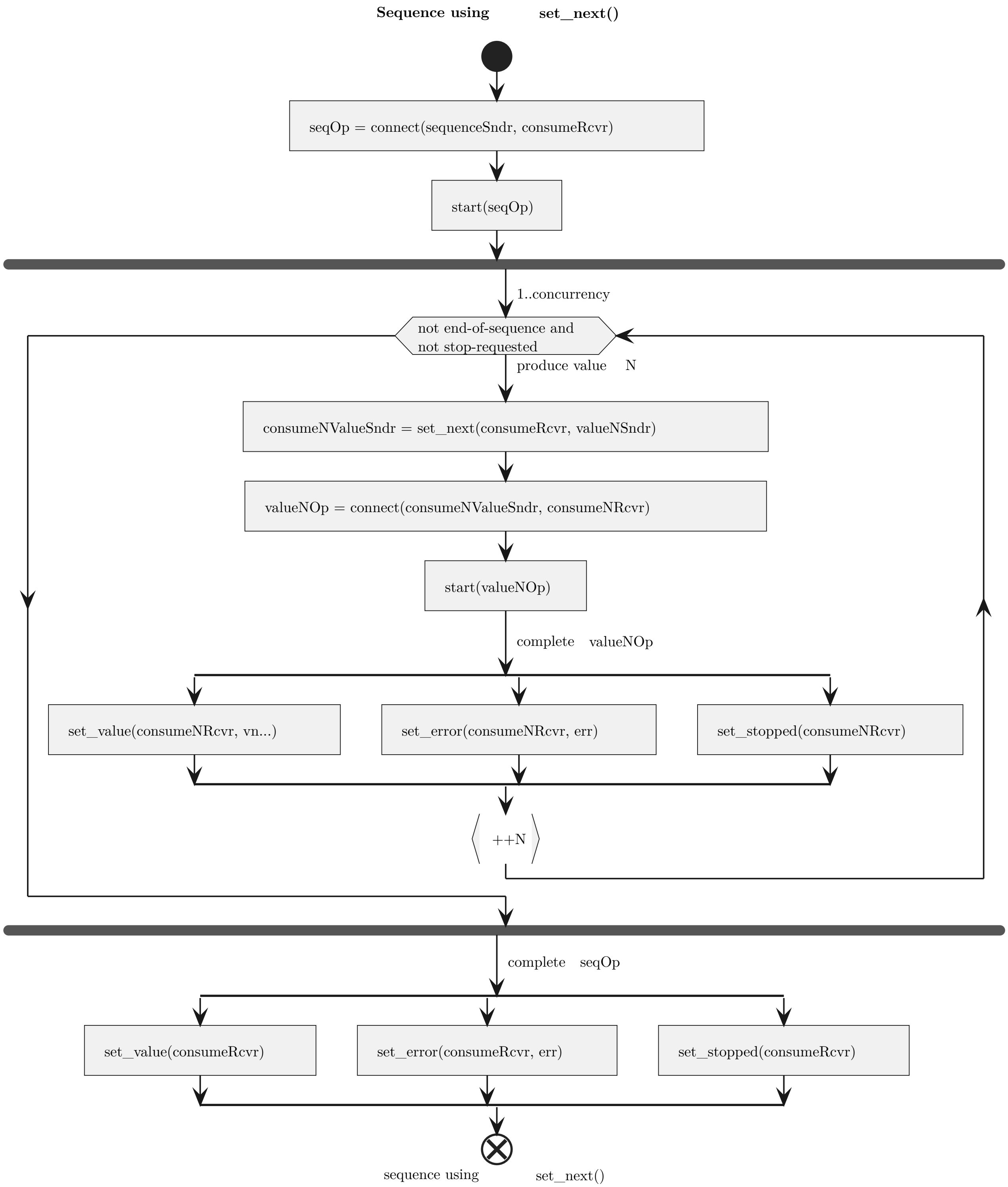
The proposal D0000R0 (TBD), introduces three customization point objects, `async_resource::open`, `async_resource_token::close` and `async_resource::run`, which define async resources within the sender and receiver framework. `async_resource::open` returns a sender that completes with a handle to the acquired resource and is used as a channel to perform any user code on the acquired resource. The `async_resource_token::close` customization point object completes when the resource have been released and `async_resource::run` does the actual work of acquiring and releasing the resource. Both `async_resource::open` and `async_resource_token::close` merely act as signals to start either operation. `async_resource::run` returns a sender of no value and completes when an acquired resource have been released. When the `async_resource::run` sender is stopped or an error occurs, it releases any acquired resources. Combining the three customization point objects D0000R0 proposes an `use_resources` algorithm which provides a safe way to acquire and release resources in a concurrent environment.

However, using sequence-senders, we can simplify this model since sequence-senders naturally provide an additional value channel and an opportunity to start a final operation upon completion of the sequence. In this case, only one customization point object, `async_resource::run`, would be needed to safely use resources in a concurrent environment. `async_resource::run` could return a sequence sender that sends only one value, which is the handle to the acquired resource. Whenever the sequence sender completes, it would automatically release the acquired resource.

3 Design

The basic progression, for a sequence of values, is to have a sender that completes when the sequence ends and a separate sender for each value.

3.1 Consuming a Sequence



3.2 Sequence Receiver

`set_next` is a new customization point object for the receiver. `set_next` is similar to `set_value`, but instead of completing the receiver, it signals the arrival of a new element in the sequence. `set_next` applies algorithms to a given sender of values and its function signature reads

```
auto set_next(receiver auto& rcvr, sender auto&& item)
  -> max-one-valued-sender-of<set_value_t()>;
```

Using a sender-based function signature instead of `void set_next(rcvr, values...)` allows `set_next` to be called without having values ready and gives fine-grained control over the elementwise operations for the sequence-operation. A sequence-receiver is required to provide `set_next` for all the item-senders produced by the sequence-operation.

A sequence-sender or operation calls `set_next(receiver, valueSender)` with a sender that will produce the next value. `set_next` returns a resulting *next-sender*. The only valid `set_value` completion signature for *next-senders* is `set_value_t()`.

The completion functions of a sequence-receiver can only be called once all started operations of the *next-senders* have completed. This agreement is referred to as the sequence-receiver contract.

A sequence operation connects and starts the sender returned from each call to `set_next`.

For so-called *lock-step* sequences, the receiver that the sequence operation connected the sender to will call `set_next` from the `set_value` completion.

> NOTE: To prevent stack overflow, there needs to be a trampoline scheduler applied to each value sender. A tail-sender will be defined in a separate paper that can be used instead of a scheduler to stop stack overflow.

3.3 Sequence Sender

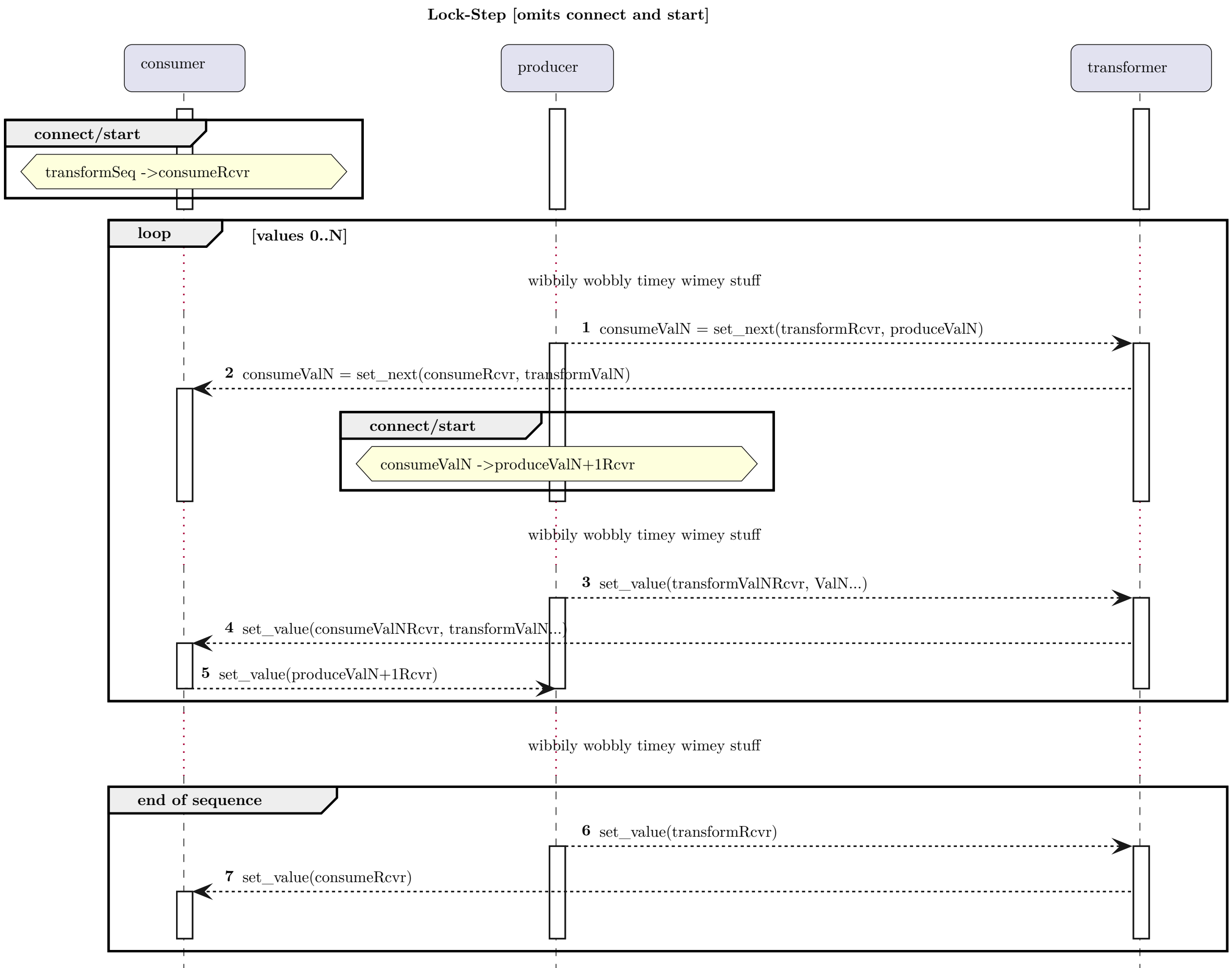
Each sequence-sender is also a sender and provides an implementation of the `get_completion_signatures_t` customization point object. However, instead of using `connect` to connect to receivers, sequence senders use the `sequence_connect` customization point object to connect to sequence-receivers. `sequence_connect` returns an operation state. All sequence-senders complete with `set_value_t()` on their success path and the `completion_signatures` of a sequence sender describe the `completion_signatures` of the value sender passed to `set_next`.

3.4 Sequences

sequence diagram for lock-step set_next() w/connect+start

3.4.1 Lock-Step

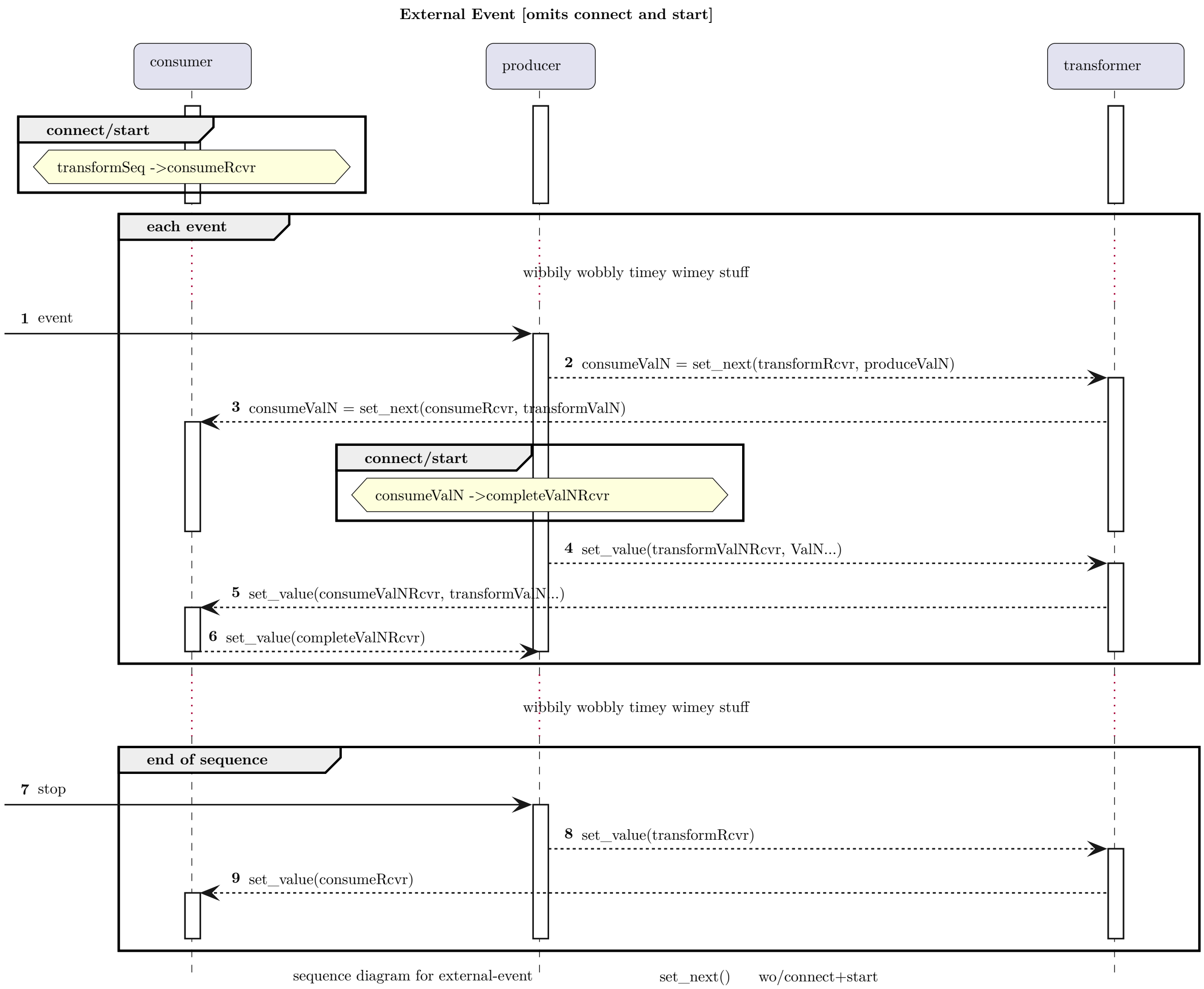
lock-step sequences are inherently serial. The next value will not be emitted until the previous value has been completely processed.



sequence diagram for lock-step set_next() wo/connect+start

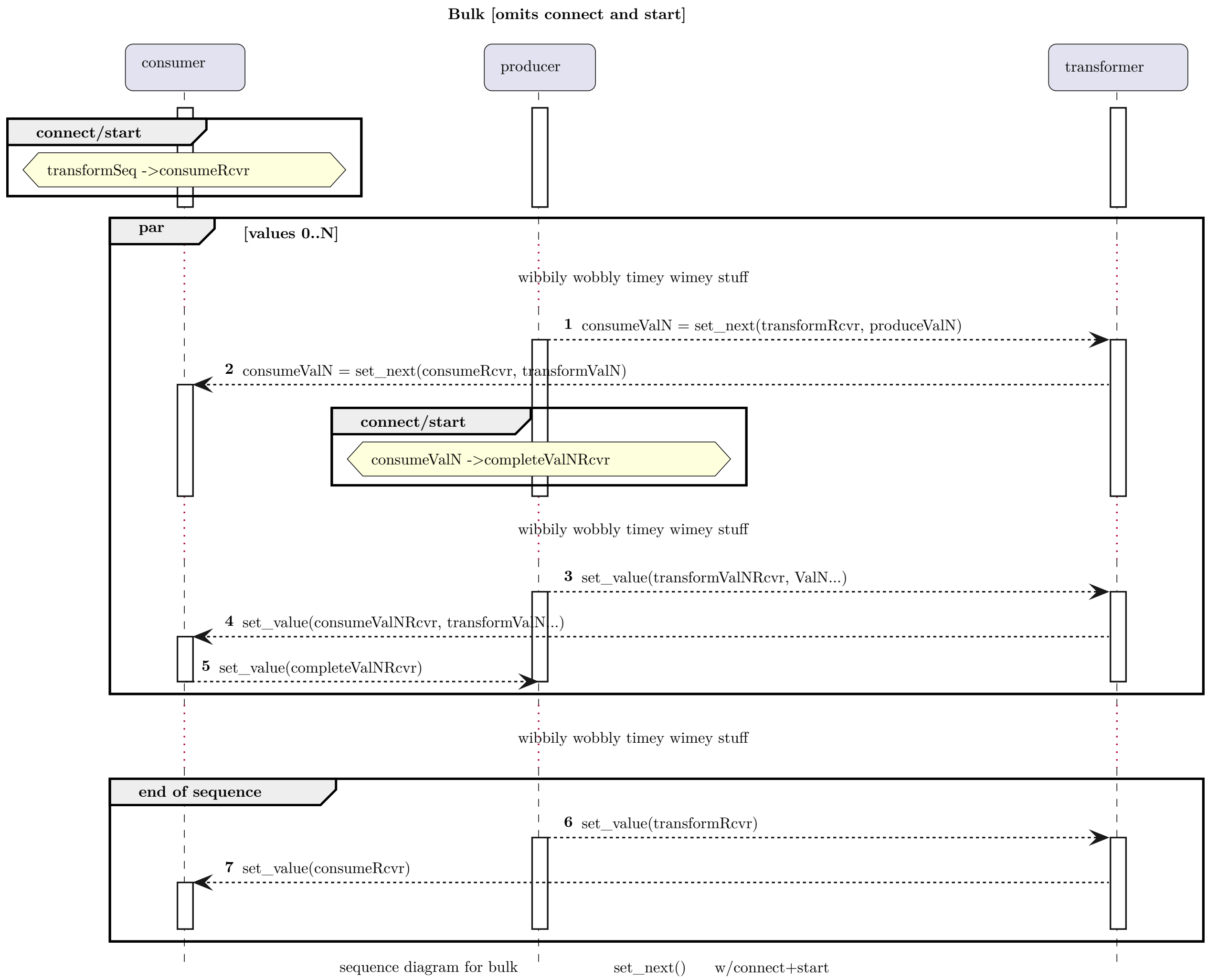
3.4.2 External Event

External events are very common. User events like pointer-move and key-down and sensor readings like orientation and ambient-light, are examples of events that produce sequences of values over time.



3.4.3 Parallelism

Sequences may be consumed in parallel. Be it network requests or ML data chunks, there is a need to use overlapping consumers for the values.



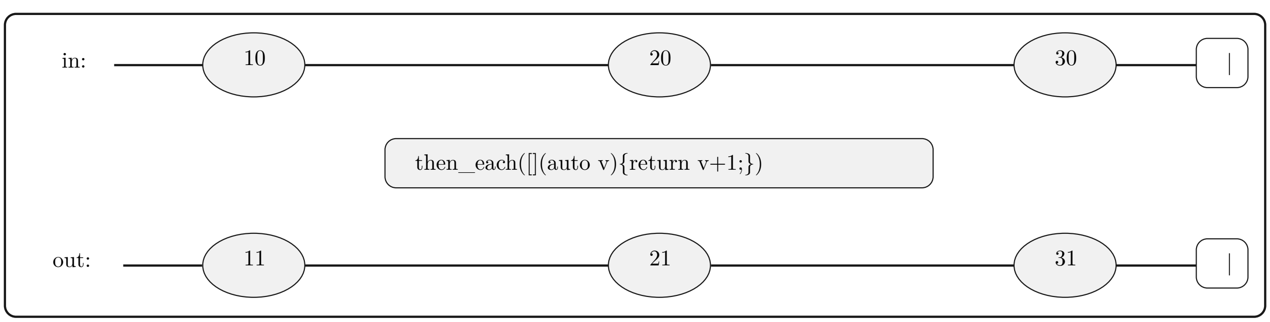
sequence diagram for bulk set_next() w/connect+start

4 Algorithms

Marble diagrams are often used to describe algorithms for asynchronous sequences.

4.1 then_each

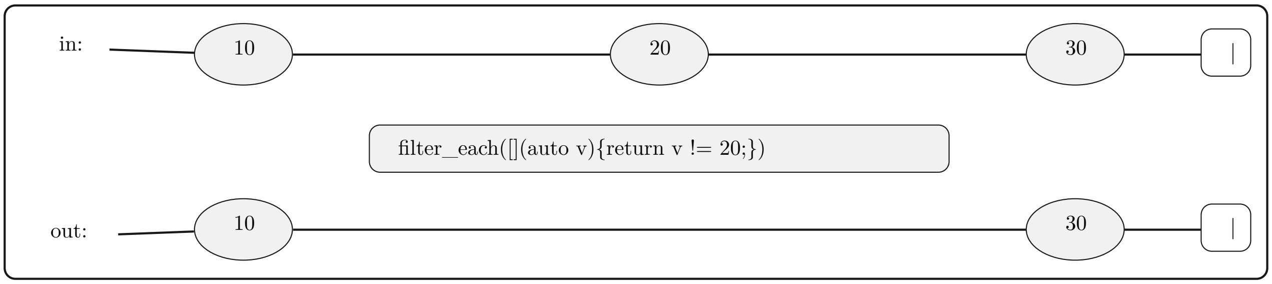
`then_each` applies the given function to each input value and emits the result of the given function.



marble diagram for then_each

4.2 filter_each

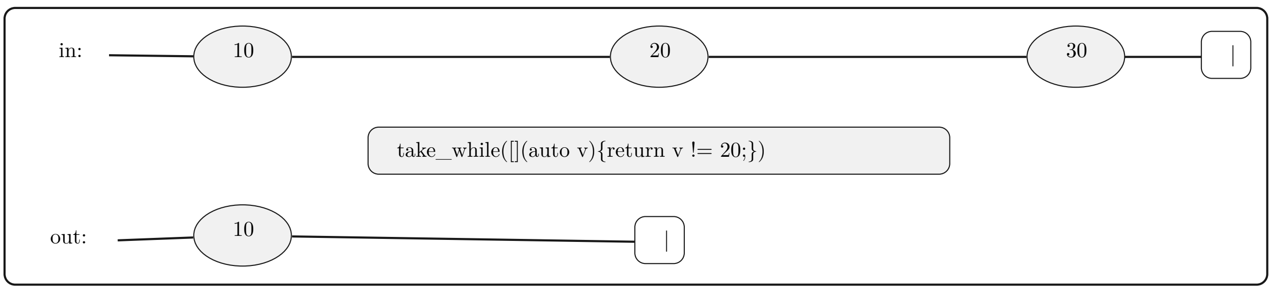
`filter_each` applies the given predicate to each input value and only emits the value if the given predicate returns `true`.



marble diagram for filter_each

4.3 take_while

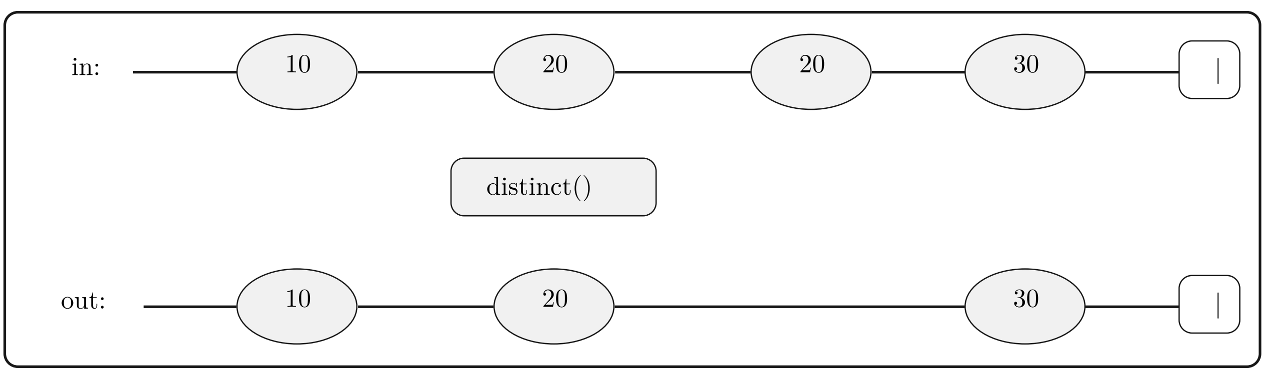
`take_while` applies the given predicate to each input value and if the given predicate returns `true` cancels the input and emits no more values.



marble diagram for take_while

4.4 distinct

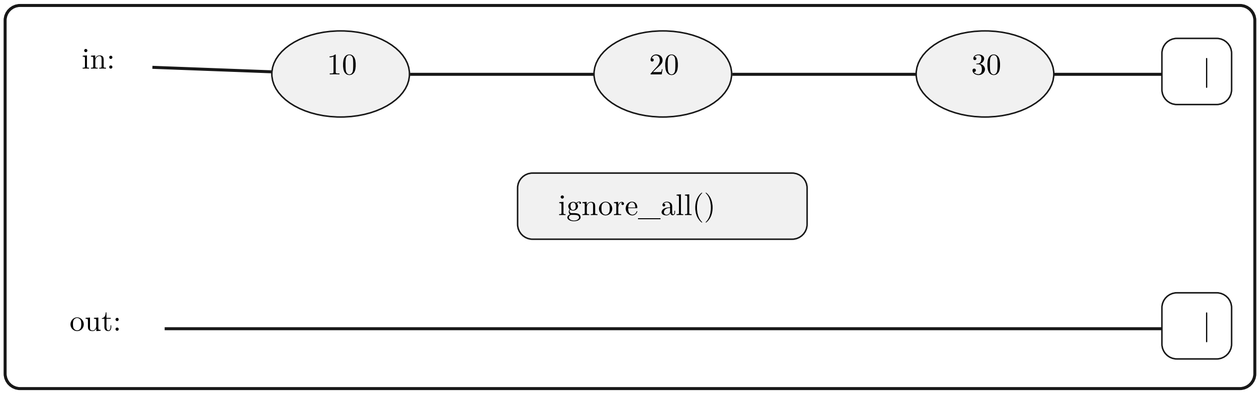
`distinct` compares each input value to a stored copy of the previous input value, if the input value and the previous input value are not the same replace the stored copy with the input value and emit the input value, otherwise do not emit the input value.



marble diagram for distinct

4.5 ignore_all

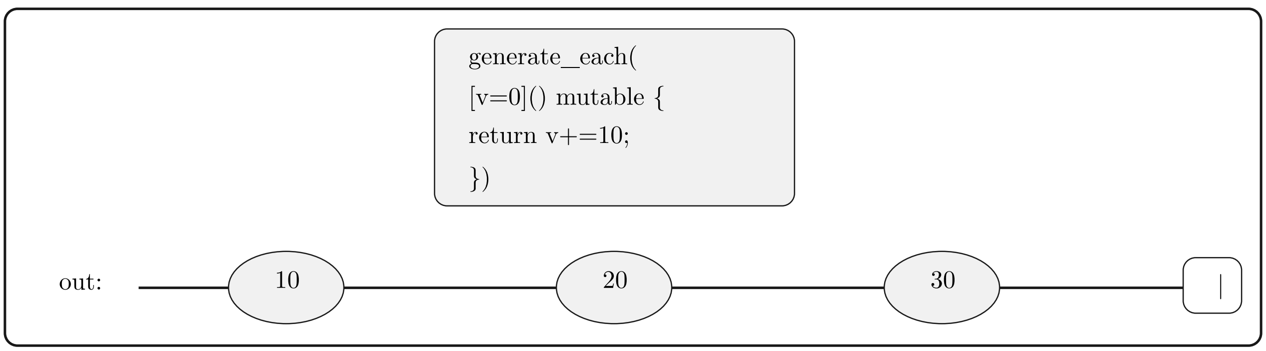
`ignore_all` does not emit any input values. This converts a sequence of values to a sender-of-void that can be passed to `sync_wait()`, etc..



marble diagram for ignore_all

4.6 generate_each

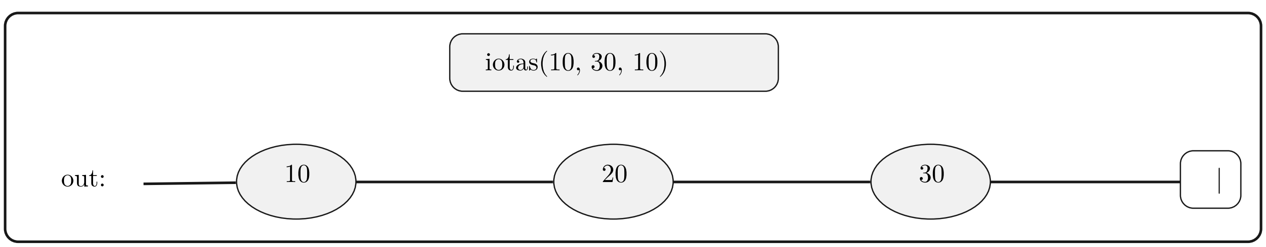
`generate_each` repeatedly calls the given function and emits the result value.



marble diagram for generate_each

4.7 iotas

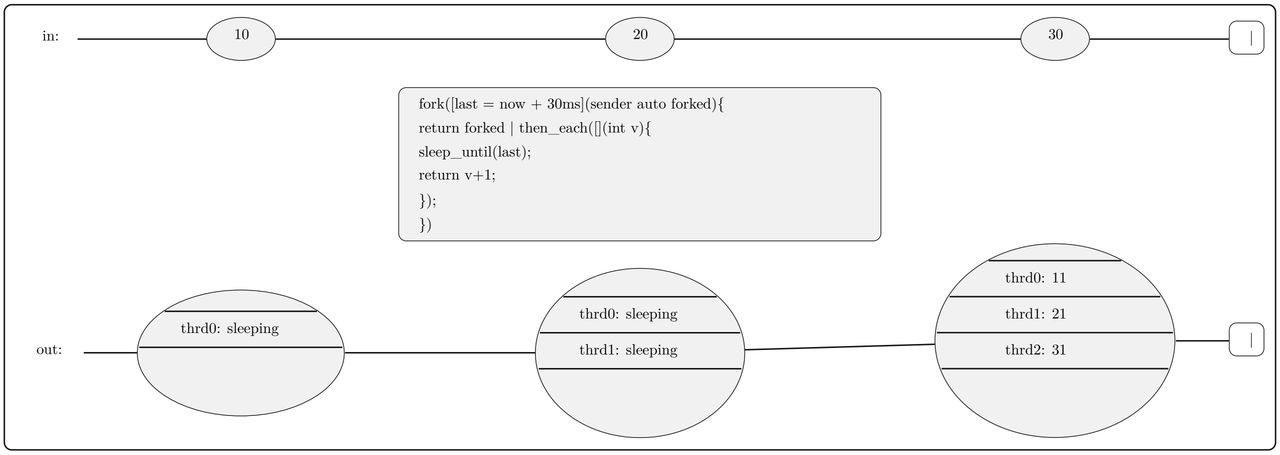
`iotas` produces a sequence of values from the given first value to the given last value with the given increment applied to each value emitted to get the next value to emit.



marble diagram for iotas

4.8 fork

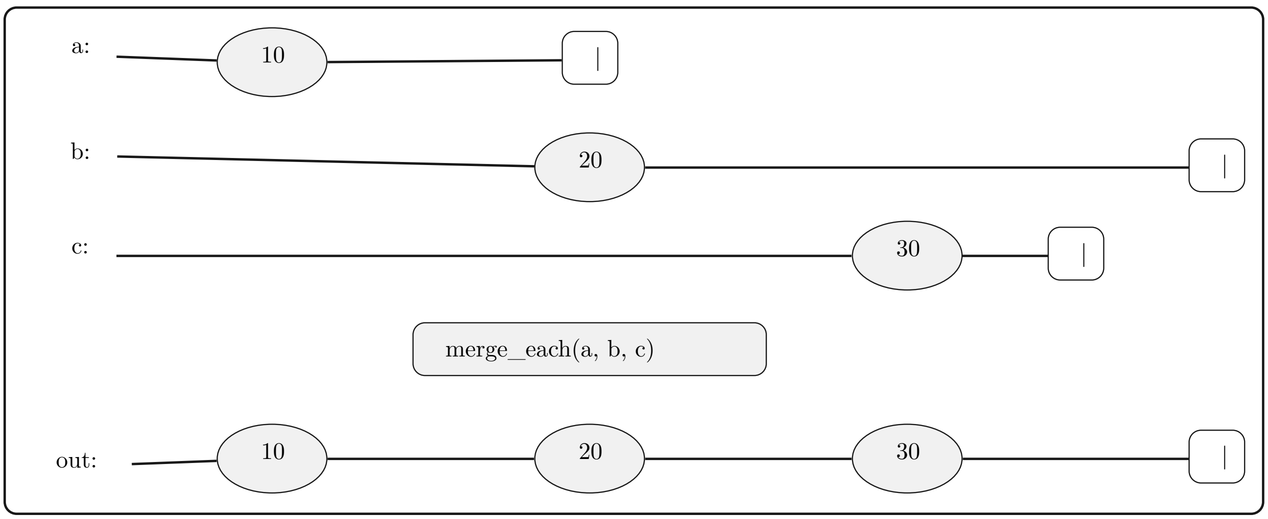
`fork` takes values from the input sequence and emits them in parallel on the execution-context provided by the receiver's environment.



marble diagram for fork

4.9 merge_each

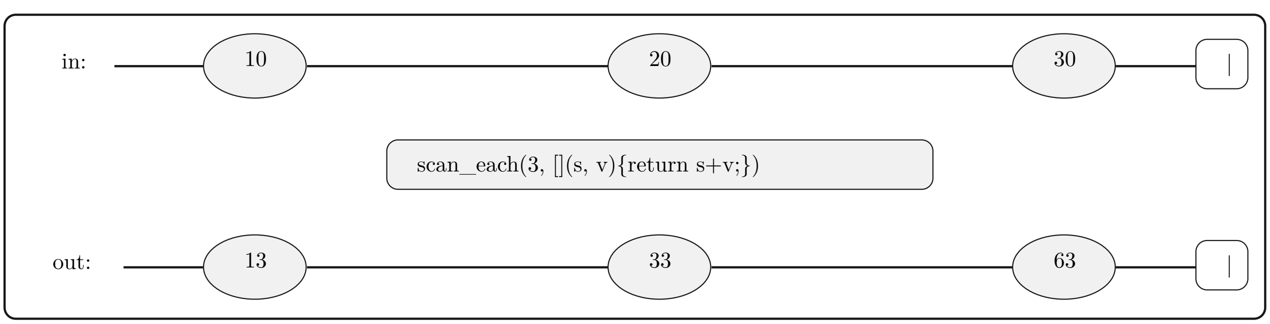
`merge_each` takes multiple input sequences and merges them into a single output sequence.



marble diagram for merge_each

4.10 scan_each

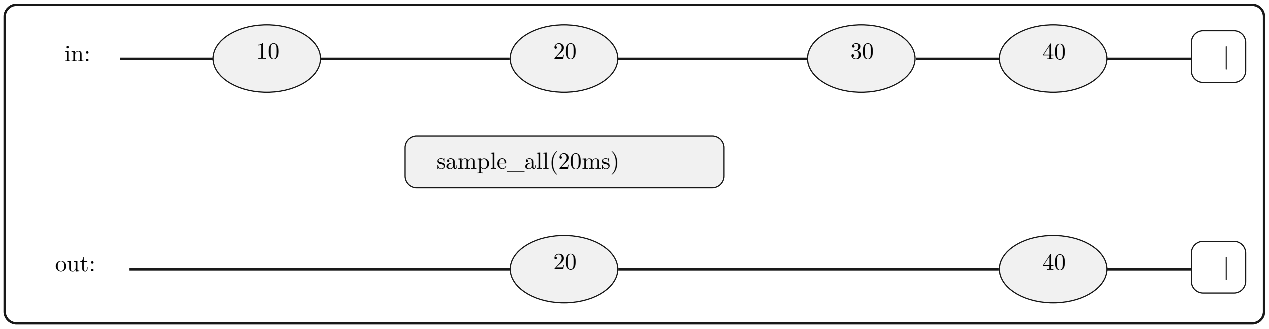
`scan_each` is like a reduce, but emits the state after each change.



marble diagram for scan_each

4.11 sample_all

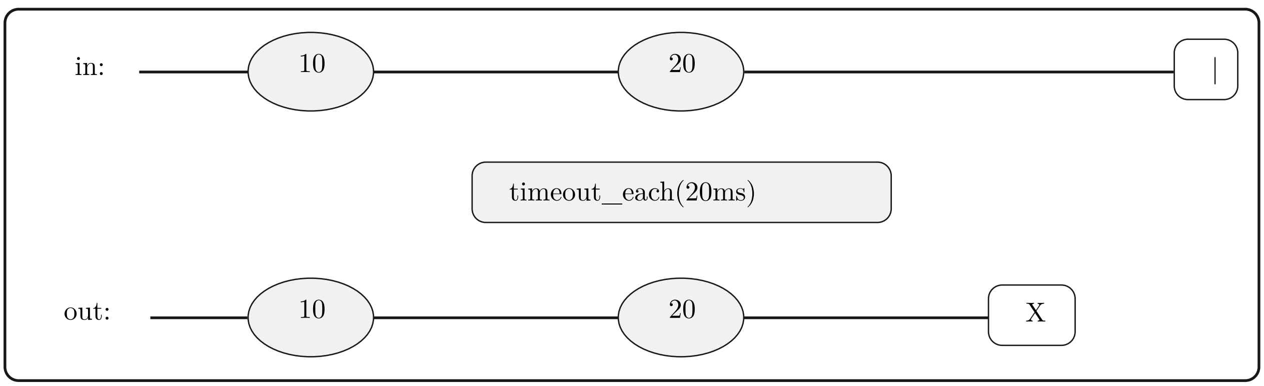
`sample_all` emits the most recent stored copy of the most recent input value at the frequency determined by the given interval.



marble diagram for sample_all

4.12 timeout_each

`timeout_each` completes the sequence with a `timeout_error` if any two input values are separated by more than the given interval.



marble diagram for timeout_each

5 References

[P2300R5] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. 2022-04-22. 'std::execution'.
https://wg21.link/p2300r5