Asynchronous Value Sequences

Draft Proposal

Document #:	D0000R0
Date:	2023-04-11
Project:	Programming Language C++
Audience:	SG1 - parallelism and concurrency
Reply-to:	Kirk Shoop <kirk.shoop@gmail.com>

Contents

1 Introduction **2**
- 1.1 Basic polling . . . 2
- 1.2 Bulk processing . . . 2
- 1.3 Web Requests . . . 2
  - 1.3.1 Connect to a firehose and parse . . . 3
  - 1.3.2 Satisfying Web Service Retry contract . . . 3
- 1.4 User Interface . . . 3
  - 1.4.1 Build a set of reducers . . . 3
  - 1.4.2 Apply a set of reducers to a model . . . 4
  - 1.4.3 Build a set of renderers . . . 4
  - 1.4.4 Apply a set of renderers to a model . . . 5

2 Design **5**
- 2.1 Consuming a Sequence . . . 6
- 2.2 `set_next()` . . . 6
- 2.3 Sequences . . . 7
  - 2.3.1 Lock-Step . . . 7
  - 2.3.2 External Event . . . 8
  - 2.3.3 Parallelism . . . 9

3 Algorithms **10**
- 3.1 then_each . . . 10
- 3.2 filter_each . . . 10
- 3.3 take_while . . . 11
- 3.4 distinct . . . 11
- 3.5 ignore_all . . . 11
- 3.6 generate_each . . . 12
- 3.7 iotas . . . 12
- 3.8 fork . . . 12
- 3.9 merge_each . . . 13
- 3.10 scan_each . . . 13
- 3.11 sample_all . . . 14
- 3.12 timeout_each . . . 14

4 References **14**

1 Introduction

This was the end goal all along.

Sender/Receiver, as described in [P2300R5], can represent a single asynchronous value.

Today, using a range-v3 generator, a `range<Sender>` can represent a set of asynchronous values, where the range generates each sender synchronously.

This paper is about sequence-senders that can asynchronously provide `0..N` value-senders. This paper also describes some of the algorithms that operate on sequence-sender.

Some of the features provided by this design are:

- back-pressure, which slows down chatty producers
- no-allocations, by default
- parallel value senders

1.1 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());
```

1.2 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);
    }
  }));
```

1.3 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();
  }));
```

1.3.1 Connect to a firehose and parse

```
struct Tweet;

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

1.3.2 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();
  };
```

1.4 User Interface

This is 'ported' from a twitter application.

1.4.1 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);
    };
  });
```

1.4.2 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
```

1.4.3 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));
```

1.4.4 Apply a set of renderers to a model

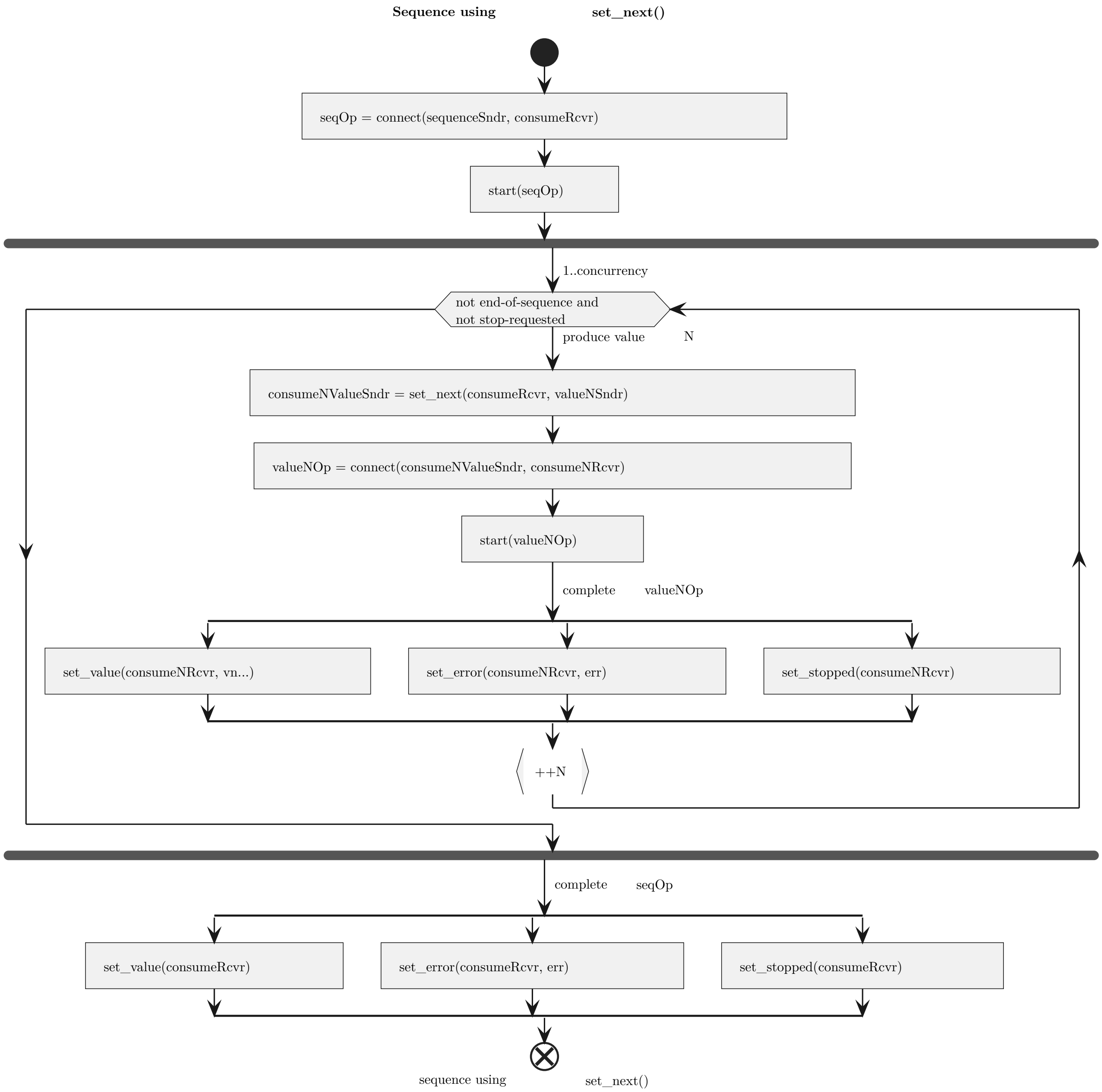
```
async_scope scope;
scope.spawn(iterate(renderers) |
  merge_each() |
  ignore_all());

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

2 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.

2.1 Consuming a Sequence



2.2 `set_next()`

`set_next` is a new customization point object for the receiver. `set_next` applies algorithms to the given sender of a value. a sequence-receiver concept subsumes a receiver concept. a sequence-receiver is required to only produce a void `set_value` to signal the end of the sequence. a sequence-receiver is required to provide `set_next` for all the value senders produced by the sequence operation.

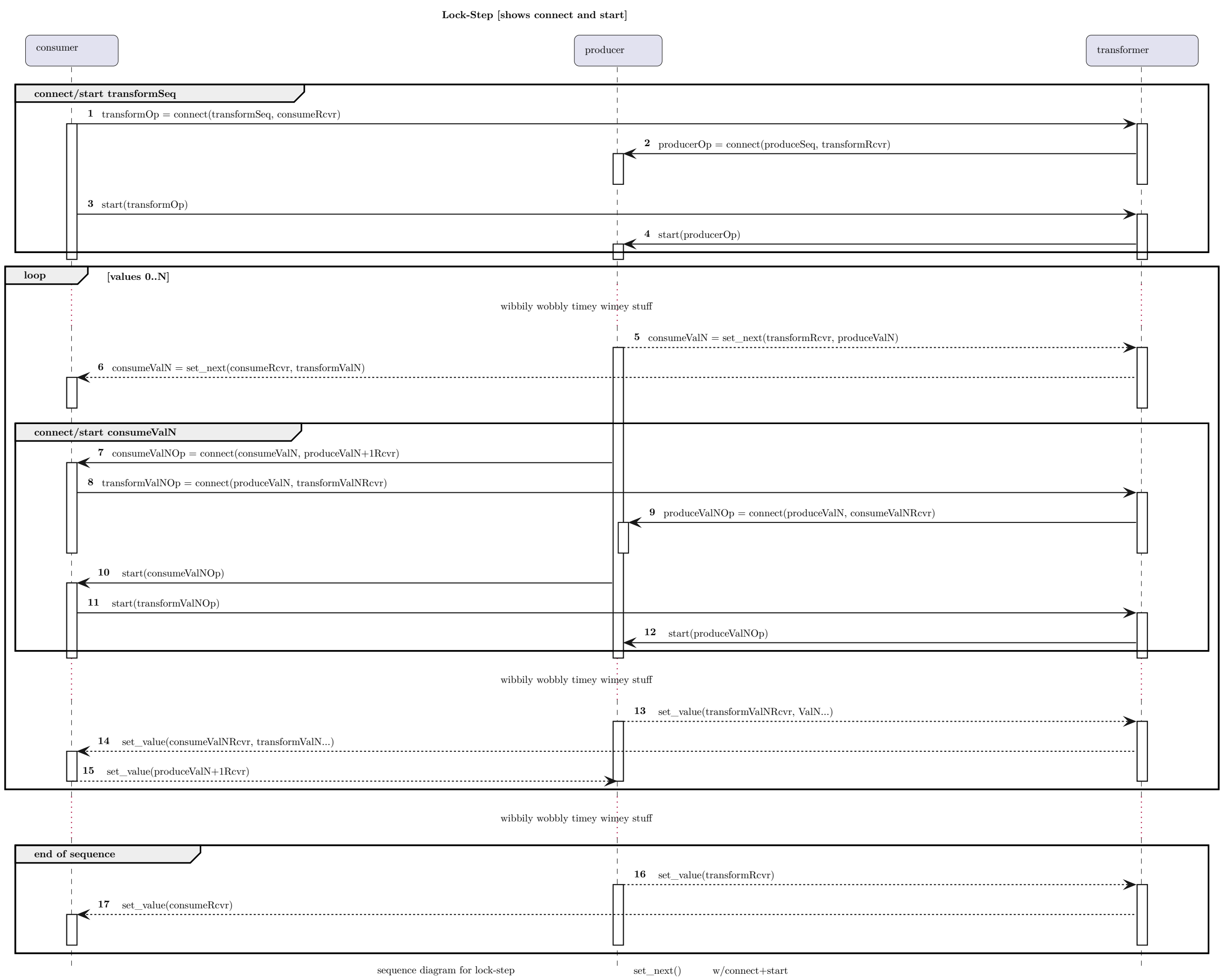
A sequence operation calls `set_next(receiver, valueSender)` with a sender that will produce the next value. `set_next` applies algorithms to the `valueSender` and returns the resulting sender.

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

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

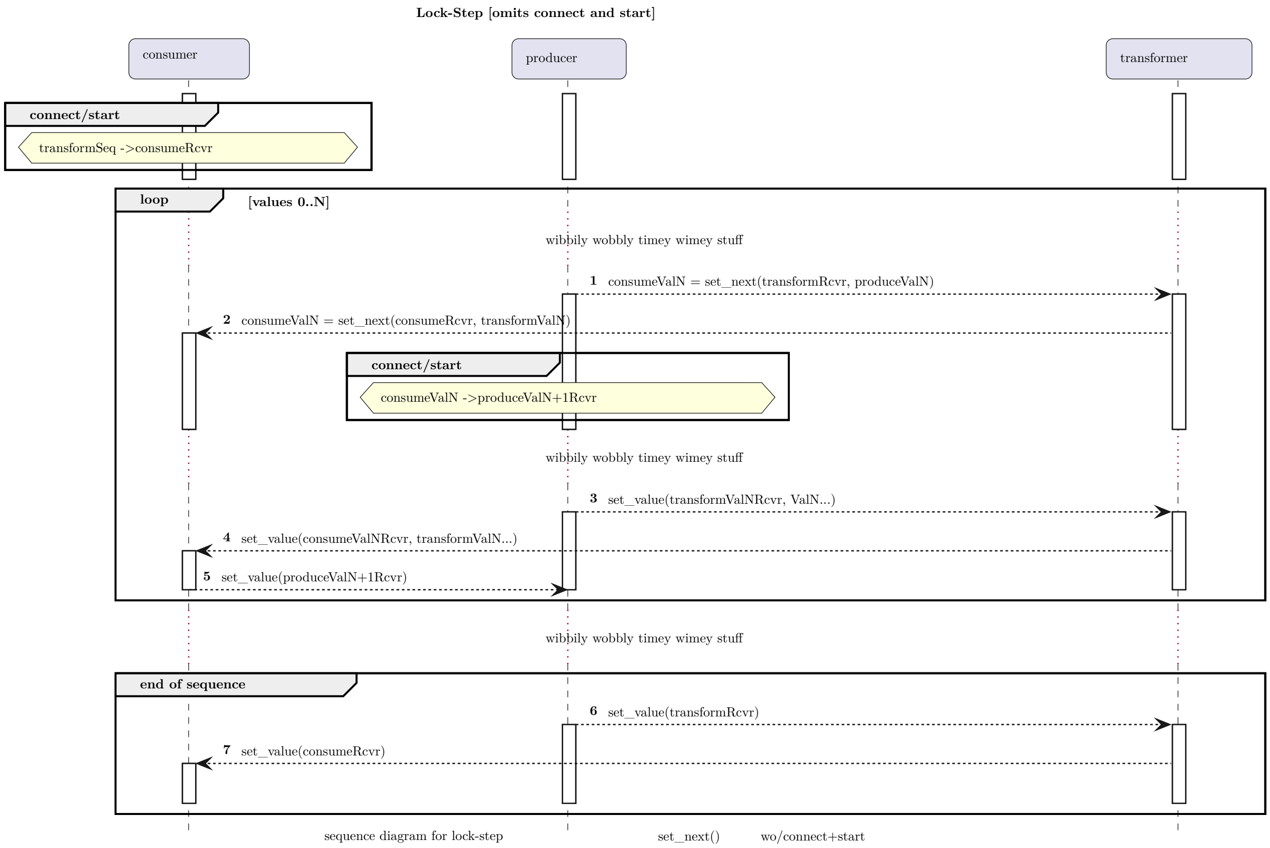
NOTE: To prevent stack overflow, there needs to be a trampoline scheduler applied to each value sender. A tail-sender will be defined in a separate paper that can be used instead of a scheduler to stop stack overflow.

2.3 Sequences



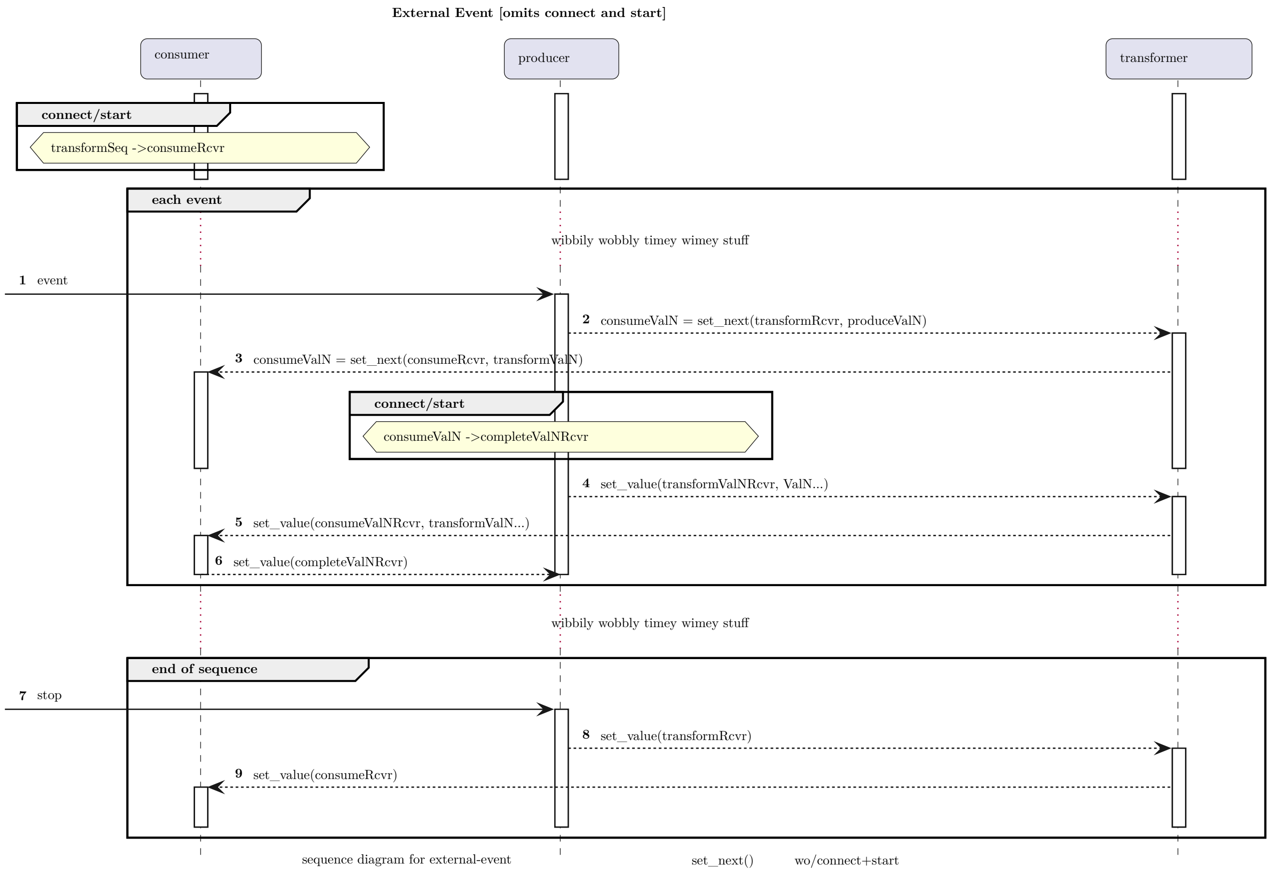
2.3.1 Lock-Step

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



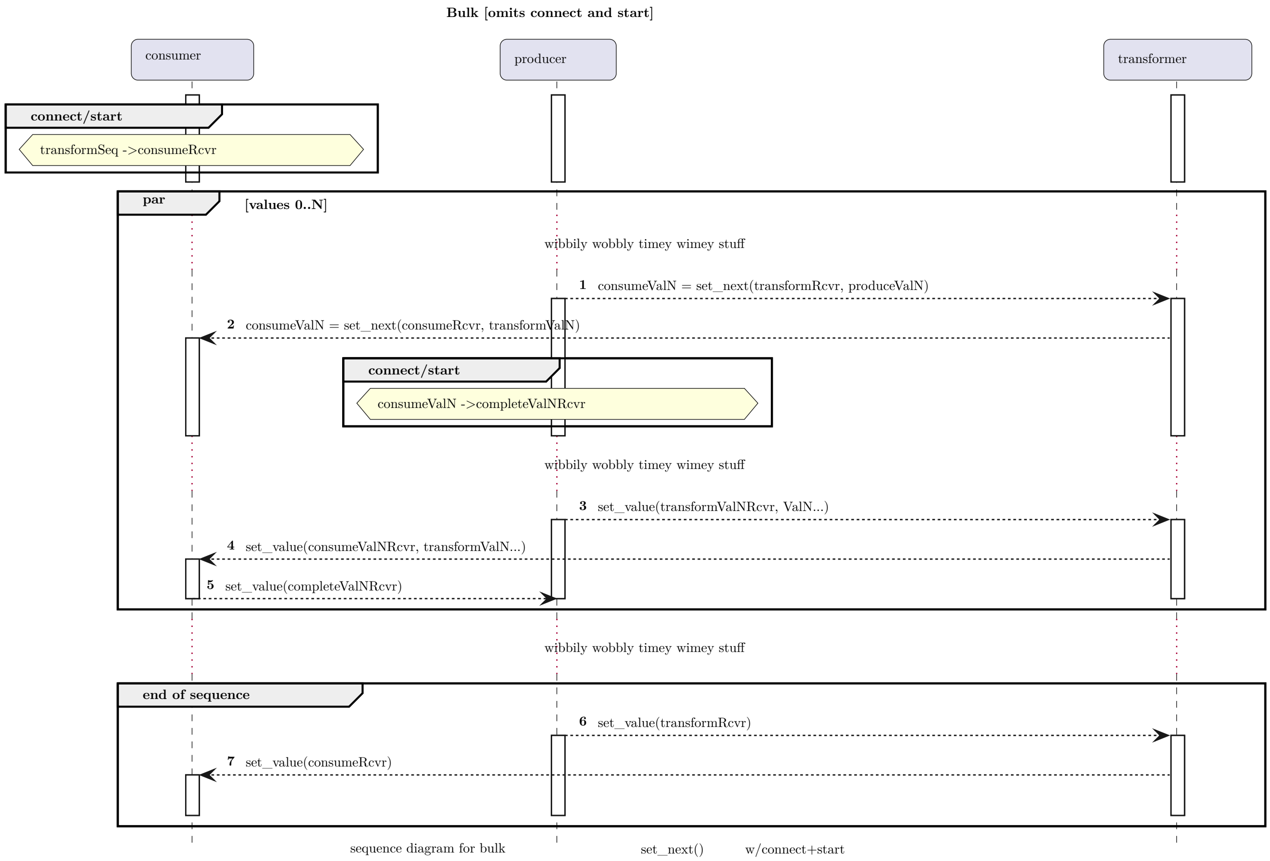
2.3.2 External Event

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.



2.3.3 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.



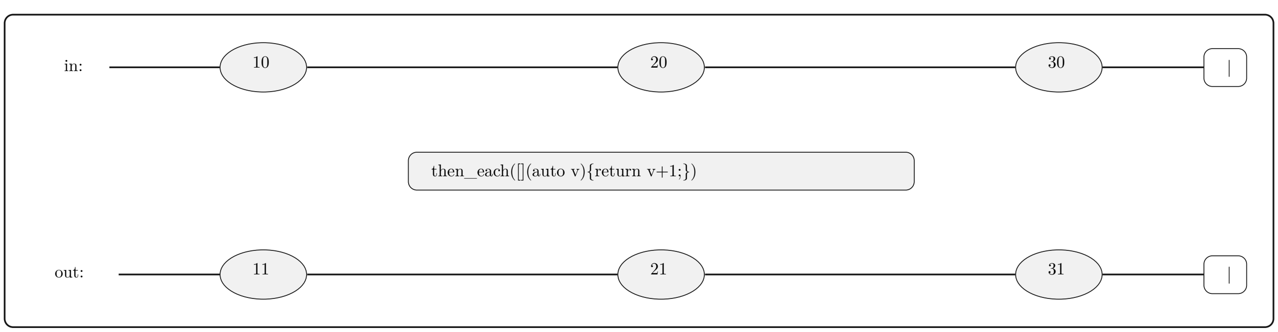
sequence diagram for bulk set_next() w/connect+start

3 Algorithms

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

3.1 then_each

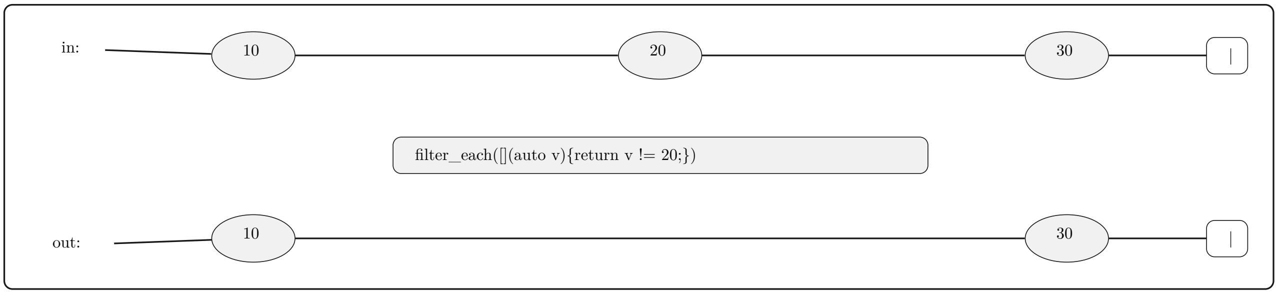
`then_each` applies the given function to each input value and emits the result of the given function.



marble diagram for then_each

3.2 filter_each

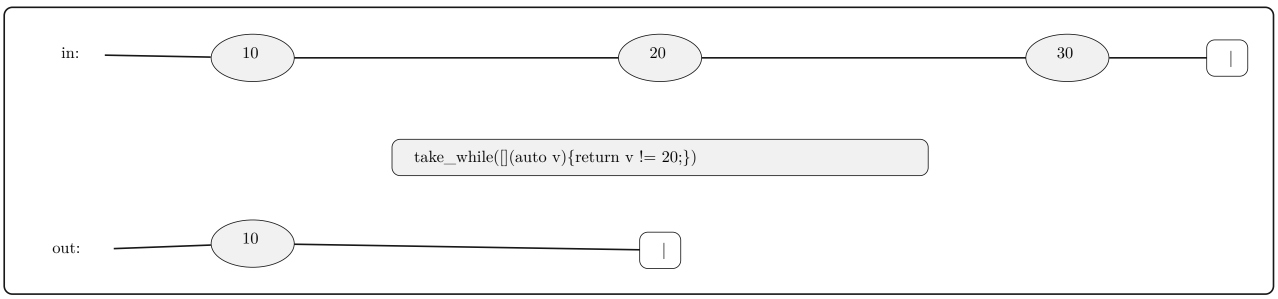
`filter_each` applies the given predicate to each input value and only emits the value if the given predicate returns `true`.



marble diagram for filter_each

3.3 take_while

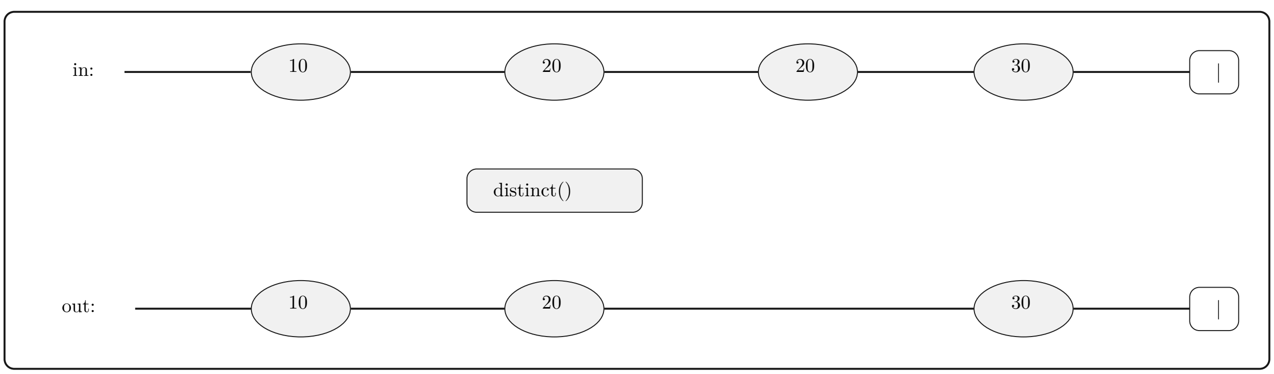
`take_while` applies the given predicate to each input value and if the given predicate returns `true` cancels the input and emits no more values.



marble diagram for take_while

3.4 distinct

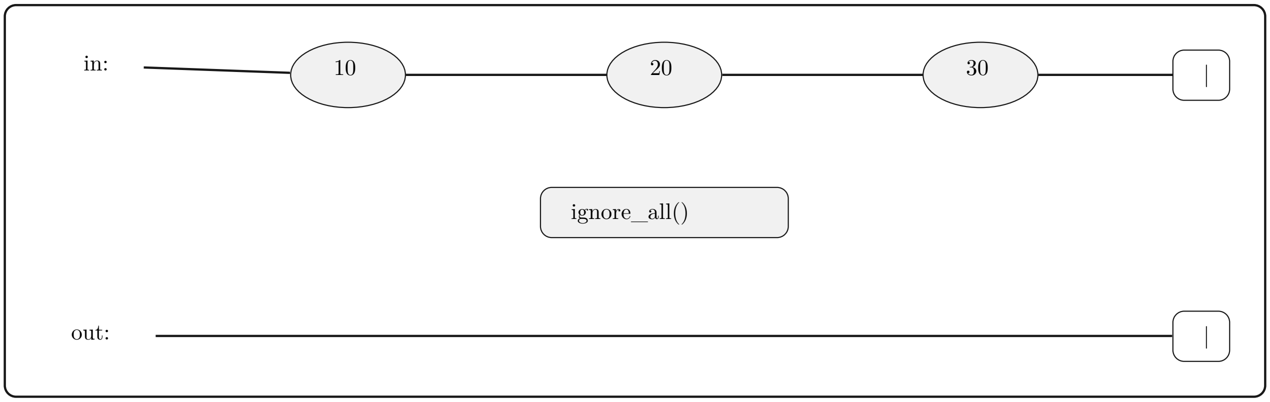
`distinct` compares each input value to a stored copy of the previous input value, if the input value and the previous input value are not the same replace the stored copy with the input value and emit the input value, otherwise do not emit the input value.



marble diagram for distinct

3.5 ignore_all

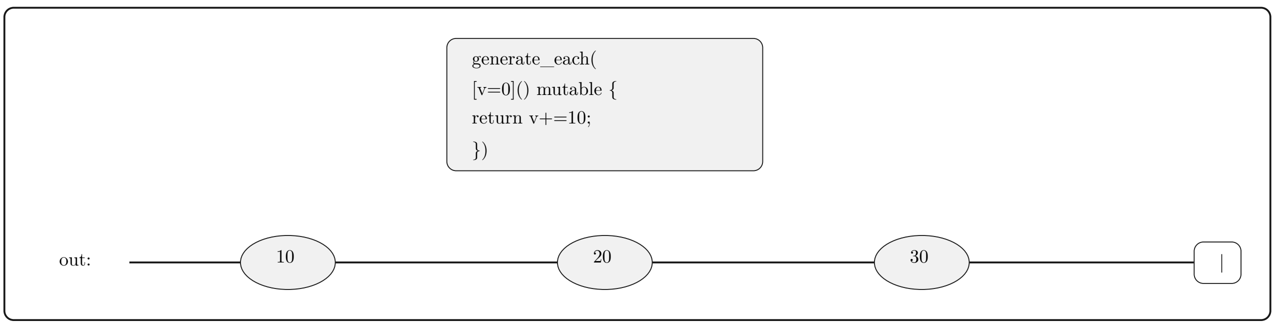
`ignore_all` does not emit any input values. This converts a sequence of values to a sender-of-void that can be passed to `sync_wait()`, etc..



marble diagram for ignore_all

3.6 generate_each

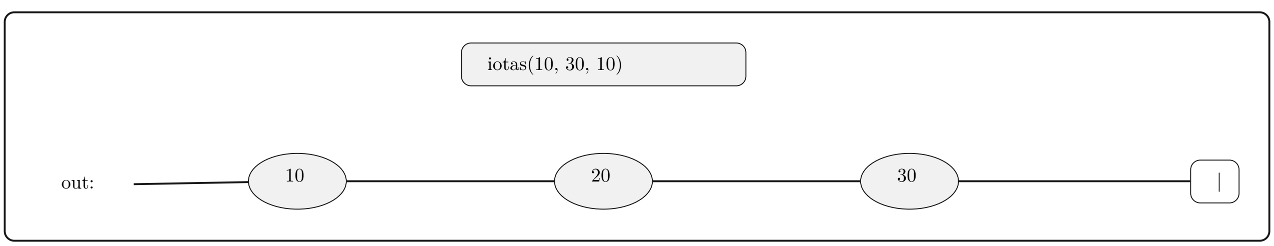
`generate_each` repeatedly calls the given function and emits the result value.



marble diagram for generate_each

3.7 iotas

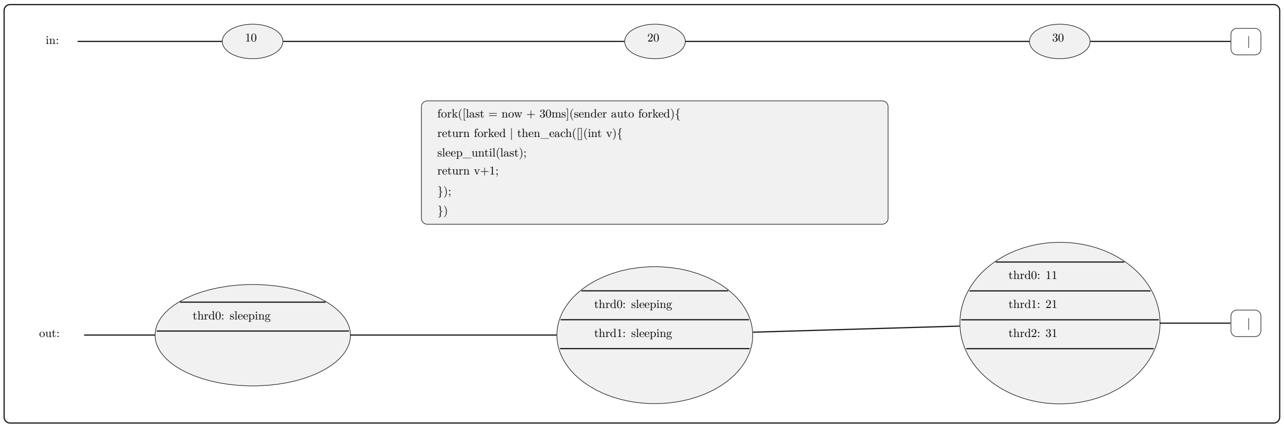
`iotas` produces a sequence of values from the given first value to the given last value with the given increment applied to each value emitted to get the next value to emit.



marble diagram for iotas

3.8 fork

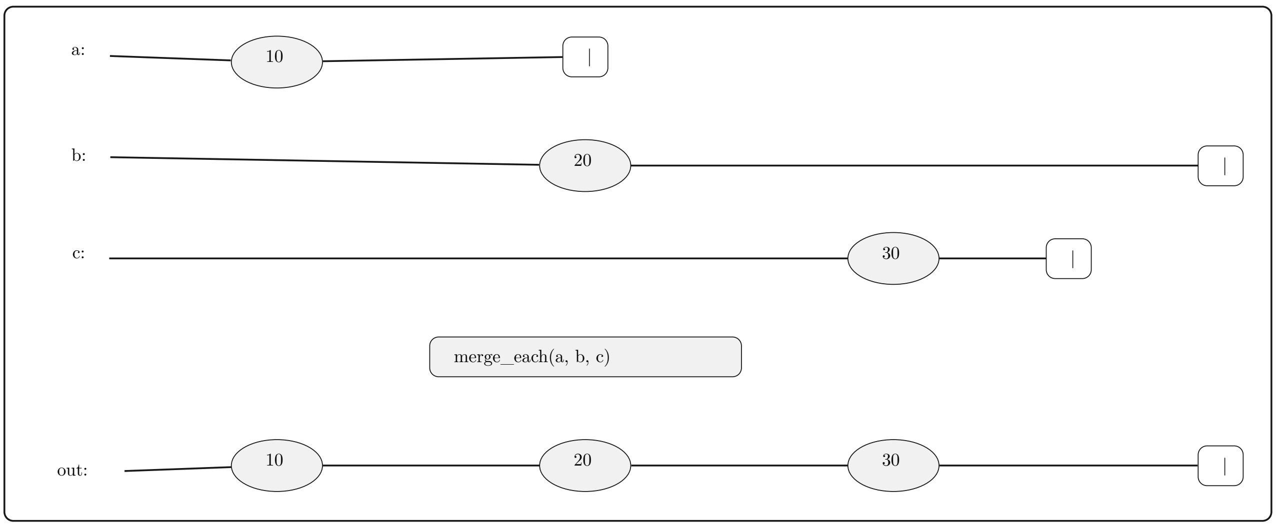
`fork` takes values from the input sequence and emits them in parallel on the execution-context provided by the receiver's environment.



marble diagram for fork

3.9 merge_each

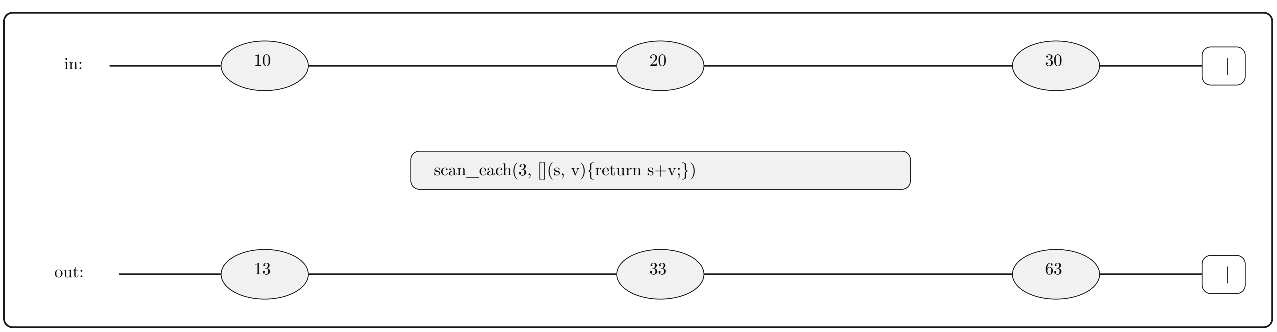
`merge_each` takes multiple input sequences and merges them into a single output sequence.



marble diagram for merge_each

3.10 scan_each

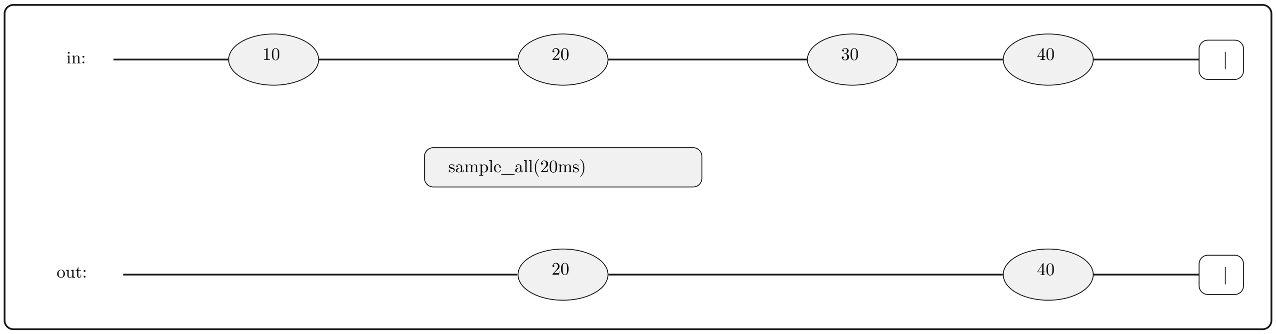
`scan_each` is like a reduce, but emits the state after each change.



marble diagram for scan_each

3.11 sample_all

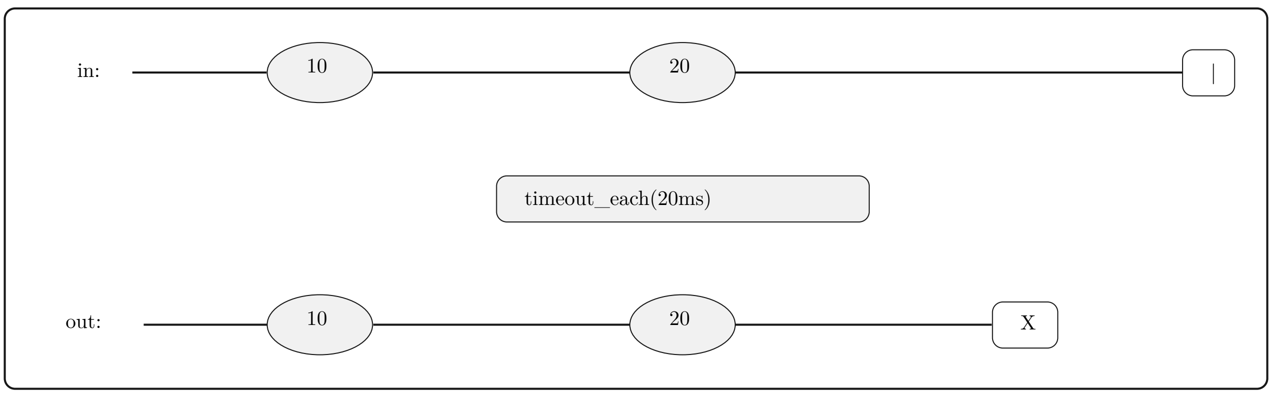
`sample_all` emits the most recent stored copy of the most recent input value at the frequency determined by the given interval.



marble diagram for sample_all

3.12 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

4 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