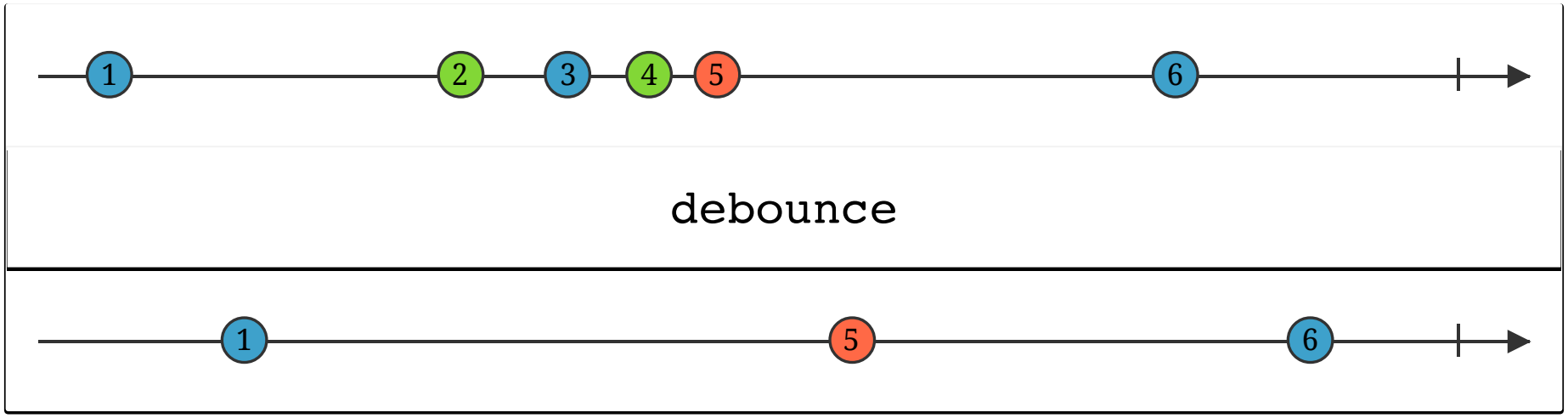
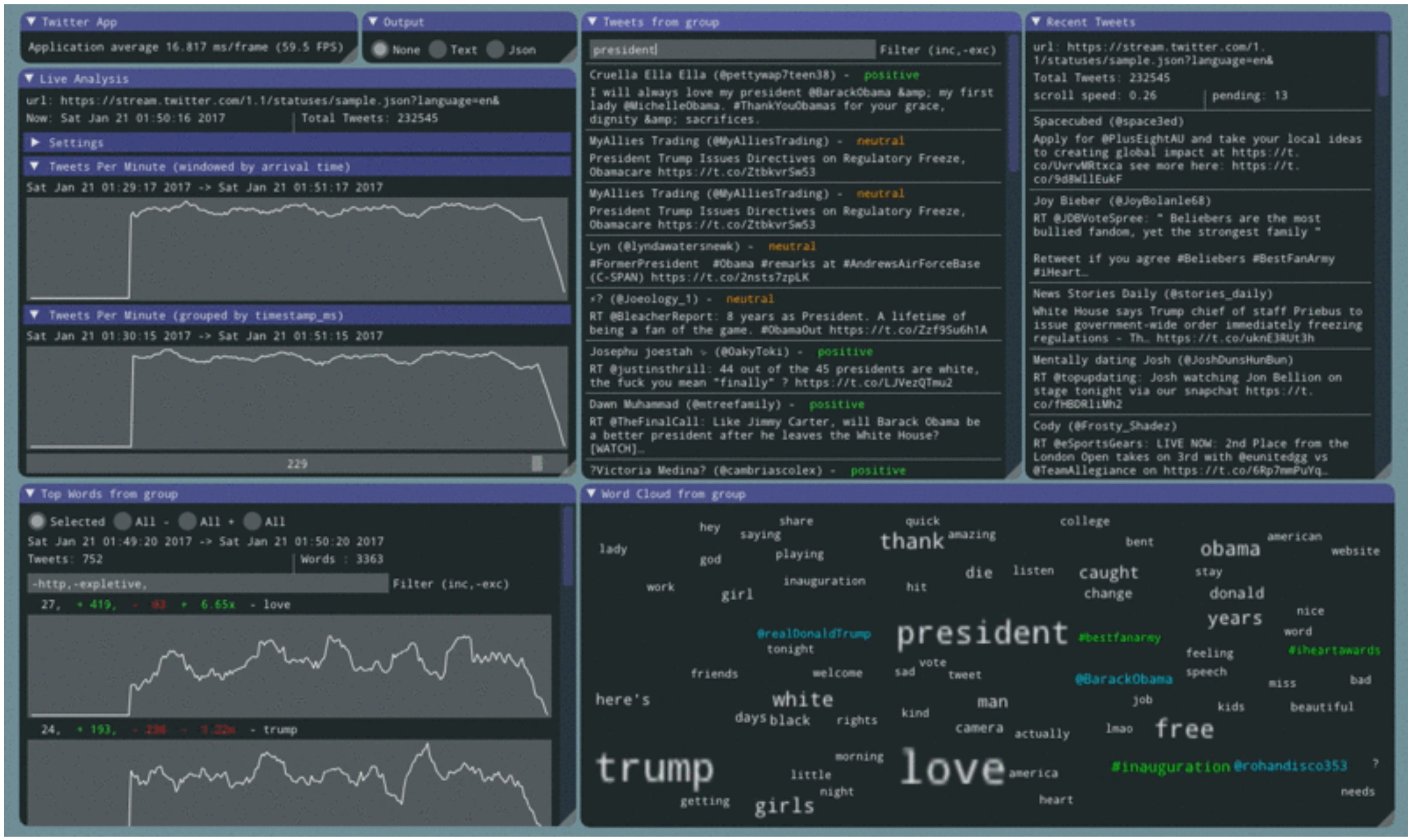
No raw std::thread!

Live Tweet Analysis in C++



DEMO



topics

pick the pattern

- primitives are too primitive
- handling Tweets
- values distributed in time
- write an algorithm

use the pattern

- virtuous procrastination
- opt-in thread-safety
- adapt existing sources
- algorithmic trancendence

primitives are too primitive

primitive technology

https://www.youtube.com/channel/UCAL3JXZSzSm8AlZyD3nQdBA

primitives are too primitive

C++ Seasoning

https://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning

primitives are too primitive COMPLETED

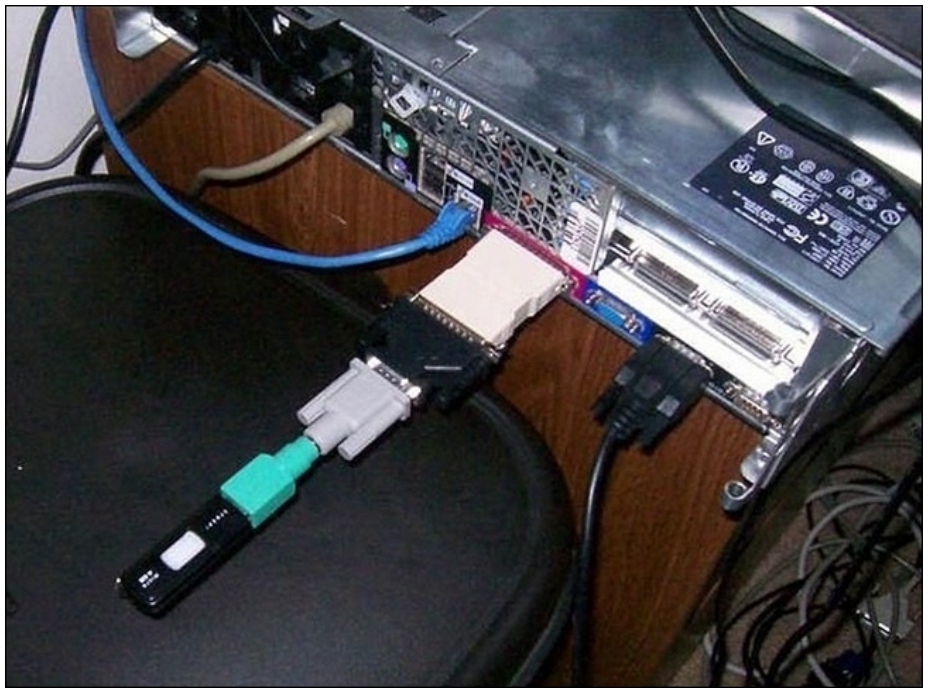
- avoid reimplementing algorithms
- avoid using synchronization primitives

next >> handling Tweets

handling Tweets

handling Tweets

callback per Tweet



handling Tweets

promise per Tweet

https://www.youtube.com/watch?v=JyVSmP-MKgY

handling Tweets

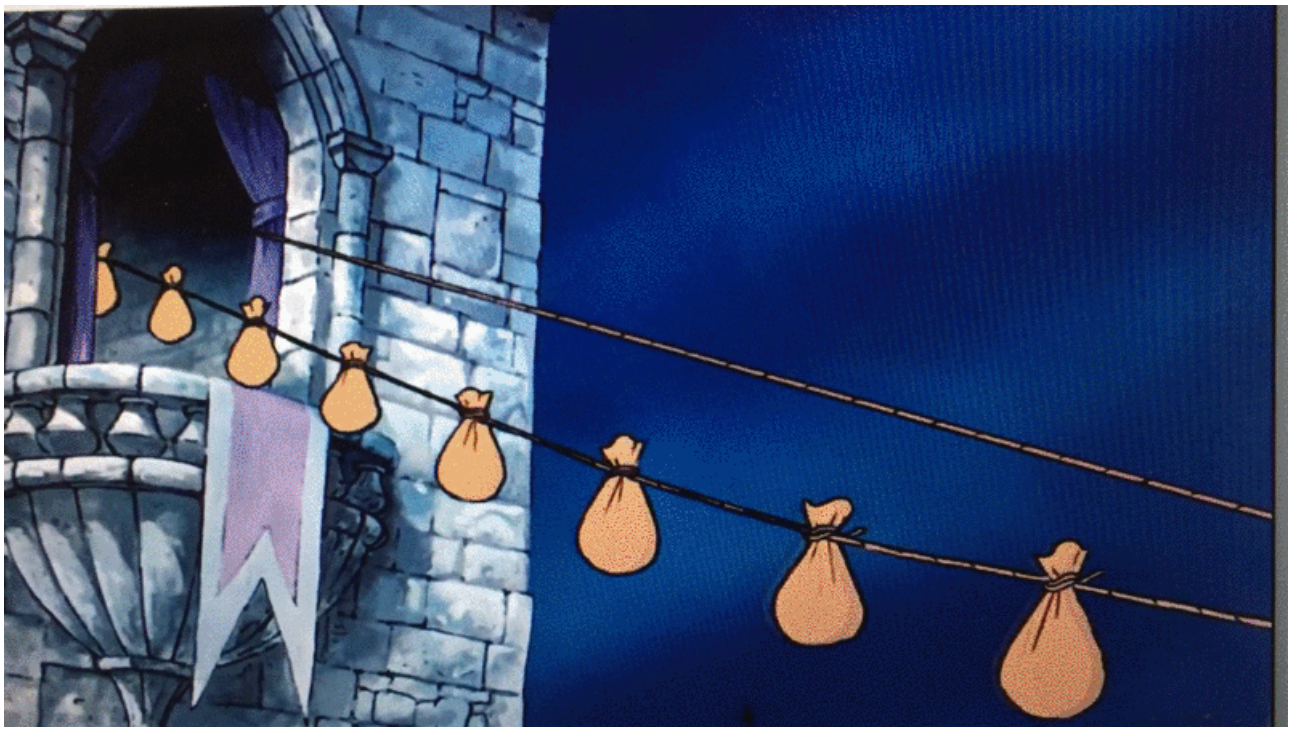
promise per Tweet

```
auto onnext = [&c](int){++c;};
for (int i = 0; i < 10000000; i++) {
    std::promise<unit> ready;
    ready.set_value(unit());
    auto isready = ready.get_future();
    if (isready.wait_for(0ms) == timeout) {
        isready.wait();
    }
    onnext(0);
}
```

- 10,000,000 on_next calls
- 3018ms elapsed
- 3,313,450 ops/sec

handling Tweets

subscription to all Tweets



handling Tweets

subscription to all Tweets

```
auto o = rx::make_subscriber<int>(
    [&c](int){++c;},
    [](std::exception_ptr){abort();});

for (int i = 0; i < 10000000; i++) {
    o.on_next(i);
}
o.on_completed();
```

- 10,000,000 on_next calls
- 17ms elapsed
- 588,235,000 ops/sec

handling Tweets

raw callbacks

```
void parseline(const string& line, auto& handlers) {
    try {
        auto text = tweettext(json::parse(line));
        auto words = splitwords(text);

        // publish tweets - multicast
        for(auto& f : handlers) {
            f(text, words);
        }
    } catch (const exception& ex){
        cerr << ex.what() << endl;
    }
}
```

handling Tweets

raw promises

w/o coroutines

```
each(objProc.begin(), objProc.end(),
 [&](It<string> c, It<string> e) {
   // split chunks and group into tweets
   string chunk = partial + *c;
   partial.clear();
   string line;
   for (auto& fragment : split(chunk, "\r\n")){
     if (!isEndOfTweet(fragment)) {
        partial = line = fragment;
        continue;
     }
     partial.clear();
     parseline(line, handlers);
   }
 });
```

coroutines

```
for await (auto& c : objProc) {

 // split chunks and group into tweets
 string chunk = partial + *c;
 partial.clear();
 string line;
 for (auto& fragment : split(chunk, "\r\n")){
   if (!isEndOfTweet(fragment)) {
      partial = line = fragment;
      continue;
   }
   partial.clear();
   parseline(line, handlers);
 }
}
```

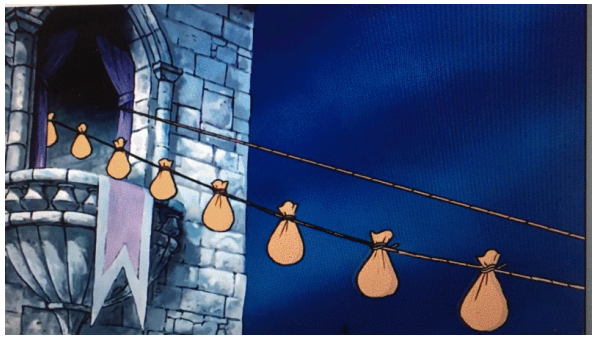
handling Tweets

subscription

```
auto tweets = defer([=](){
    auto url = oauth2SignUrl("https://stream.twitter...");
    return http.create(http_request{url, method, {}, {}}) |
      map([](http_response r){
        return r.body.chunks;
      }) |
      merge(tweetthread);
  }) |
  twitter_stream_reconnection(tweetthread) |
  parsetweets(poolthread, tweetthread) |
  publish() | ref_count();
```

handling Tweets COMPLETED

- mentioned callback composition
- showed promise and subscription costs
- showed examples of each



next >> values distributed in time

values distributed in time

my son says: "space vs time is explained by playing cards"

values distributed in time

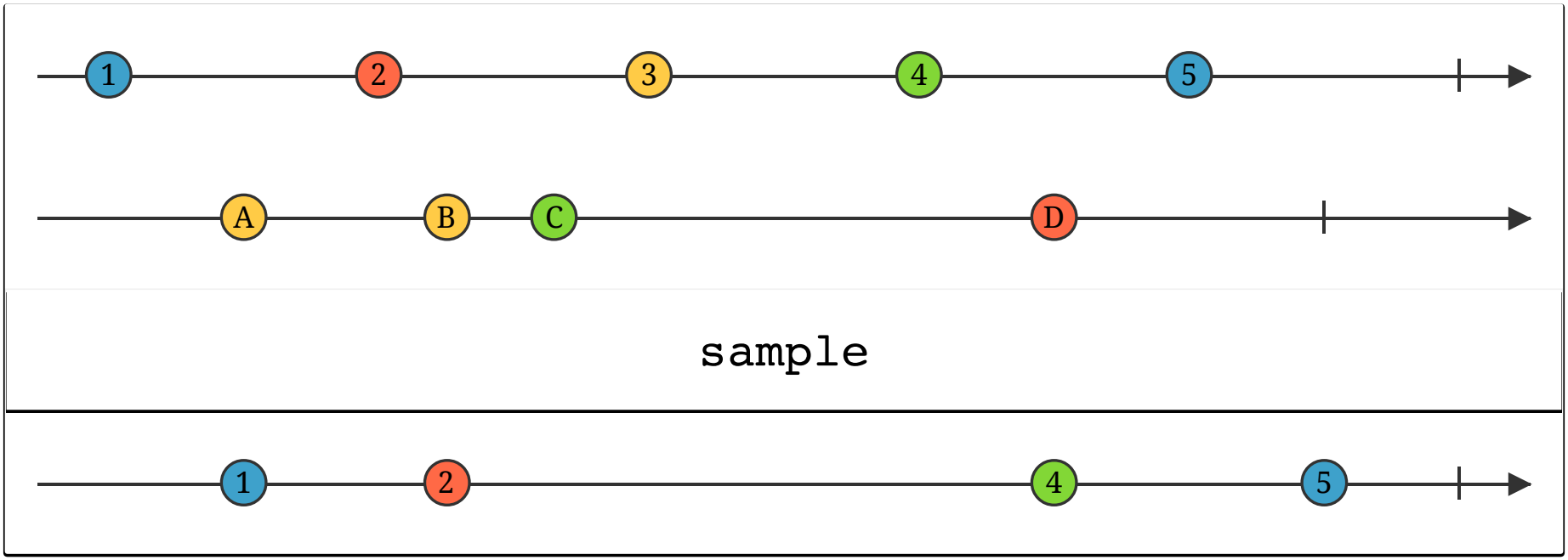
each player has 0 or more cards

values distributed in time

the dealer distributes cards in time

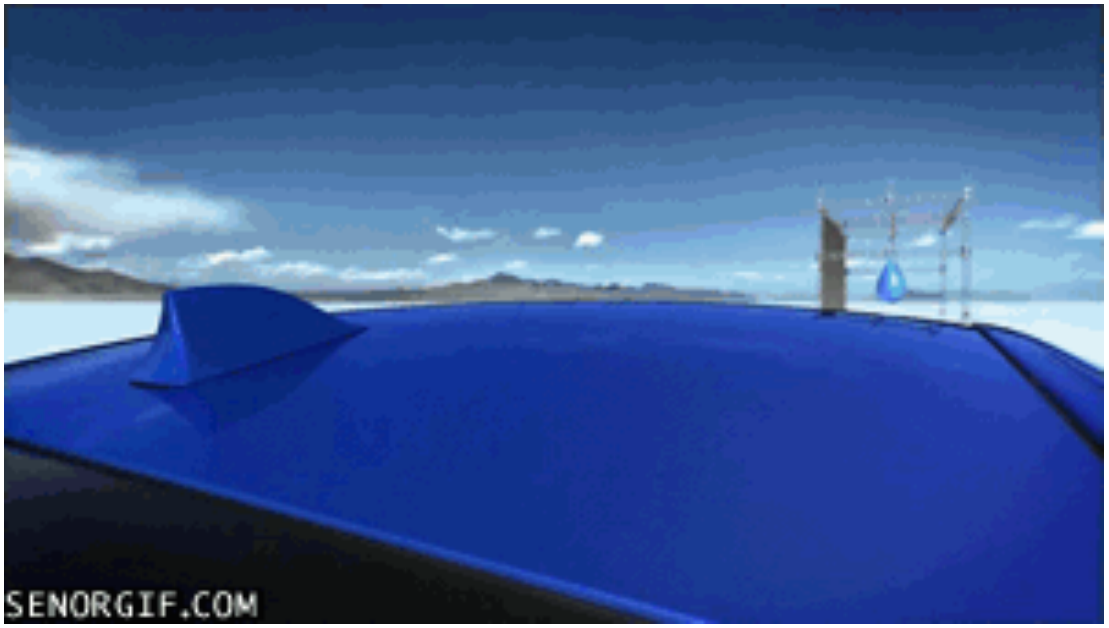
values distributed in time

marble diagrams are used to describe values distributed in time.



flow of a subscription

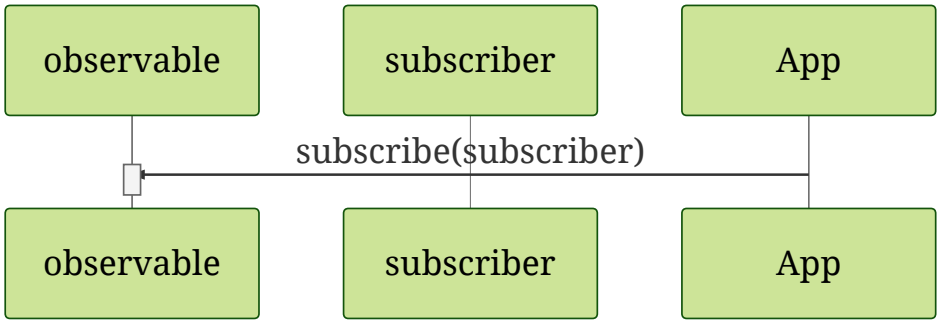
how not to get wet!



```
observable | subscribe(subscriber);
```

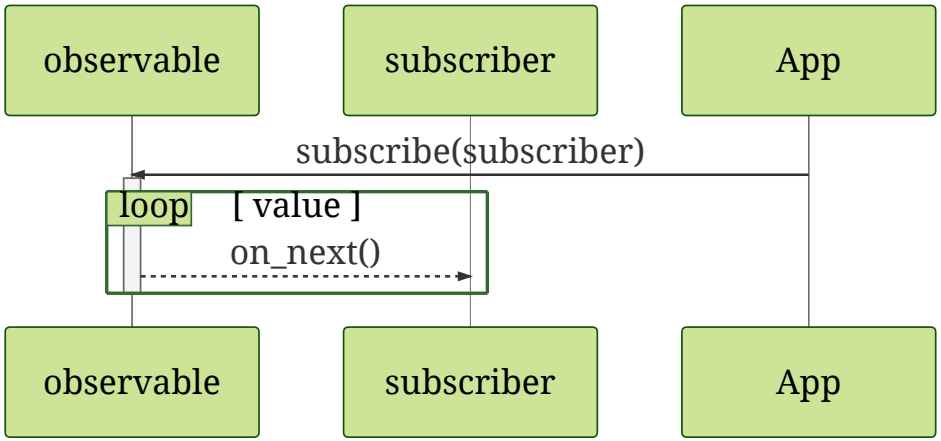
- demonstrate the contract of a subscription

```
observable | subscribe(subscriber);
```



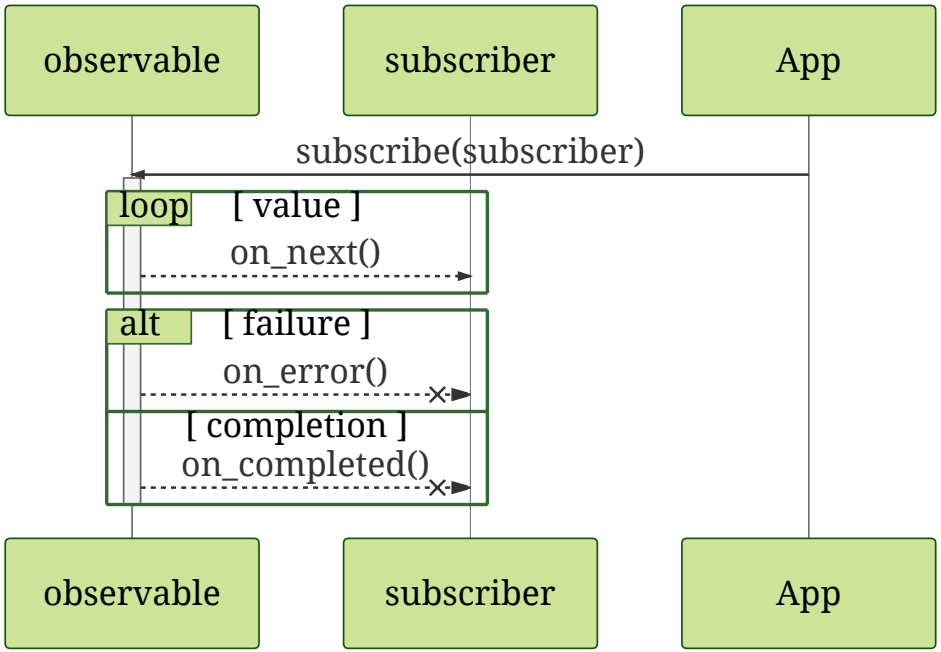
- demonstrate the contract of a subscription
- observables defer work

```
observable | subscribe(subscriber);
```



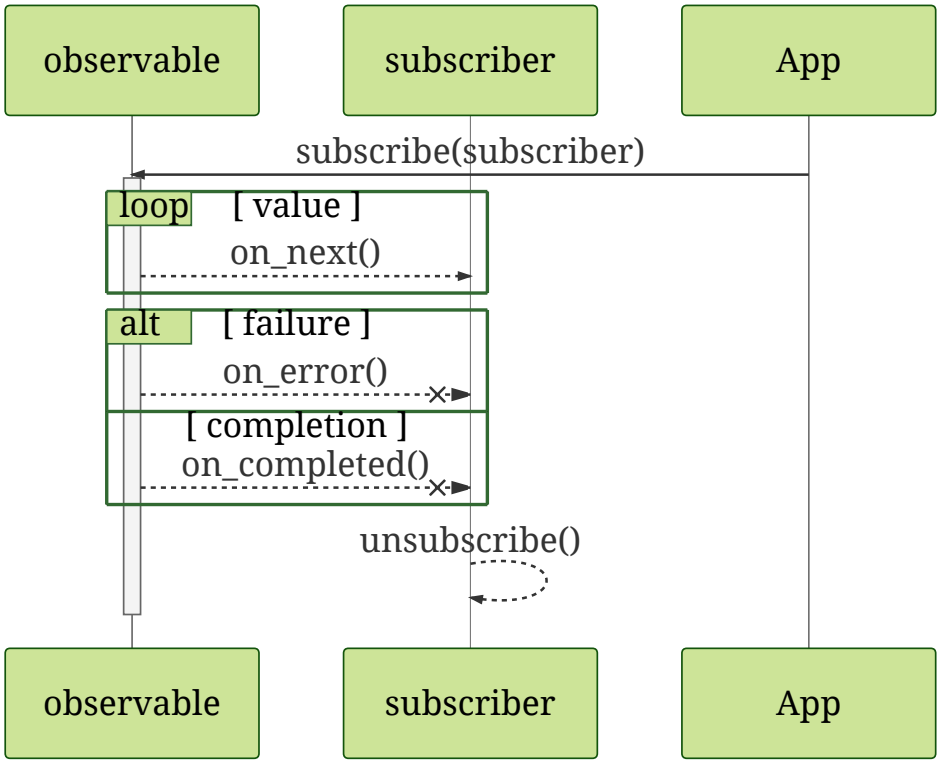
- demonstrate the contract of a subscription
- observables defer work
- calls to the subscriber will never overlap in time

```
observable | subscribe(subscriber);
```



- demonstrate the contract of a subscription
- observables defer work
- calls to the subscriber will never overlap in time
- on_error is the last call that a subscriber will receive
- on_completed is the last call that a subscriber will receive

```
observable | subscribe(subscriber);
```



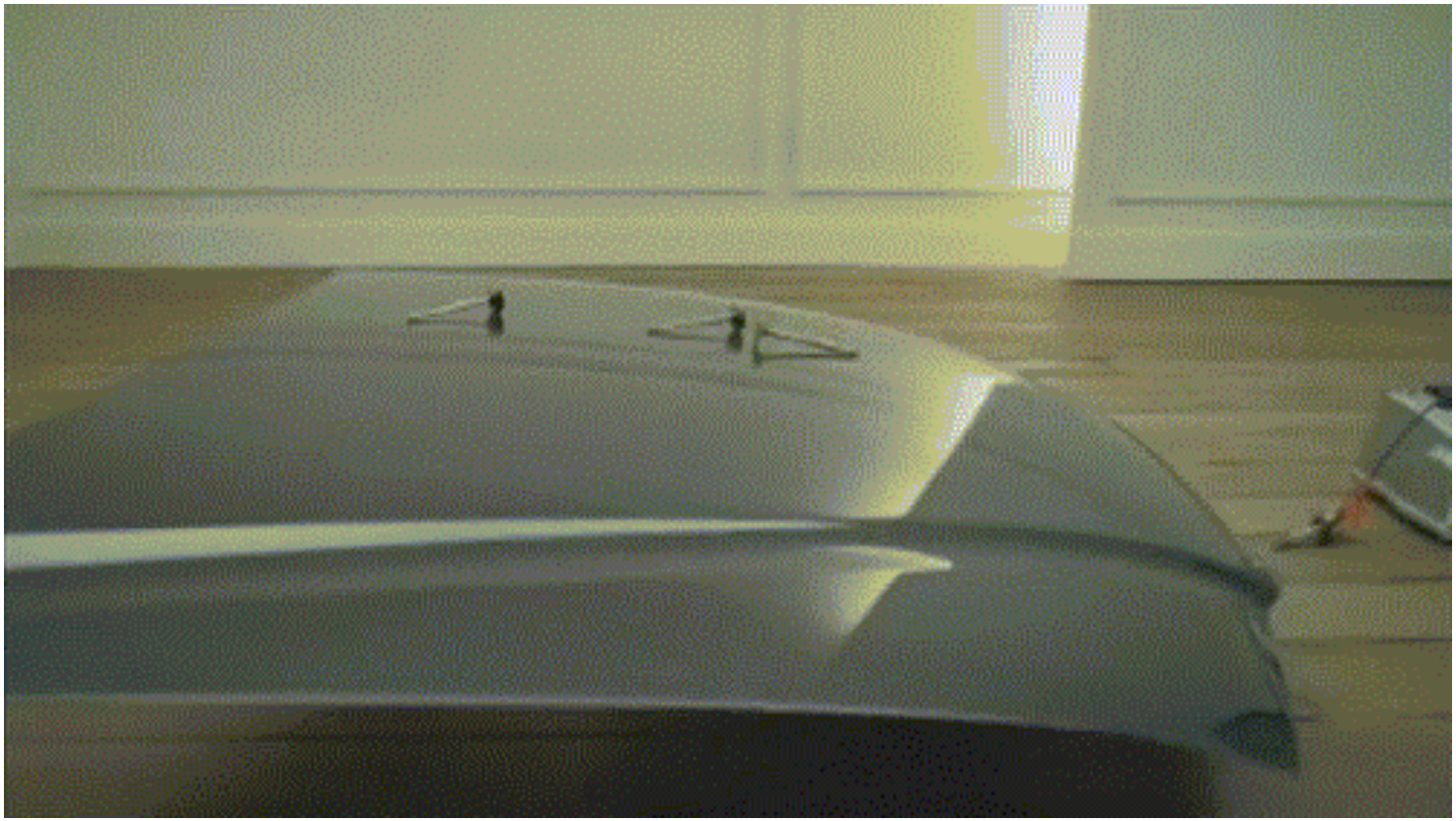
- demonstrate the contract of a subscription
- observables defer work
- calls to the subscriber will never overlap in time
- on_error is the last call that a subscriber will receive
- on_completed is the last call that a subscriber will receive
- unsubscribe is the destructor for the subscription lifetime

values distributed in time COMPLETED

- demonstrated values distributed in time
- described flow of a subscription
- subcriptions are useful to..
  - defer work
  - provide intermediate results
  - combine values from multiple sources

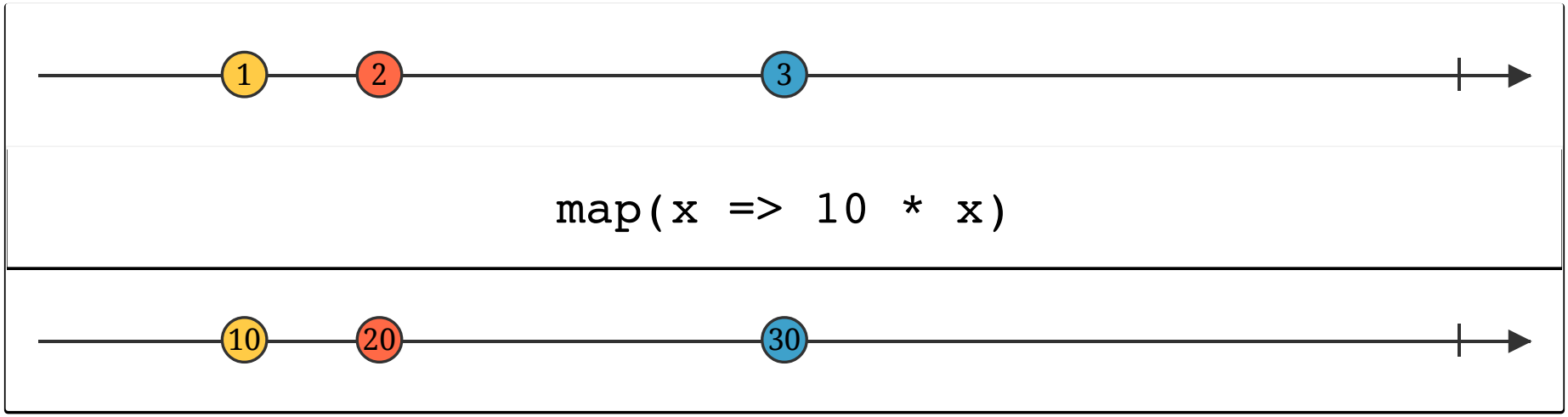
next >> write an algorithm

write an algorithm



write an algorithm - transform

`transform` calls a function with each value that arrives and passes out the result of the function.



map is a common alias for `transform`.

write an algorithm - transform

```
auto map = [](auto selector){
```

write an algorithm - transform

```
auto map = [](auto selector){

  return [=](auto in){
    return create([=](auto out){
```

- an operator is a function that takes an observable and returns an observable

write an algorithm - transform

```
auto map = [](auto selector){

  return [=](auto in){
    return create([=](auto out){

      return in
        | subscribe(
          out.get_subscription(),
          [](auto v){
            out.on_next(selector(v));
          },
```

- an operator is a function that takes an observable and returns an observable
- values from `in` are transformed by `selector` and the result passed to `out`

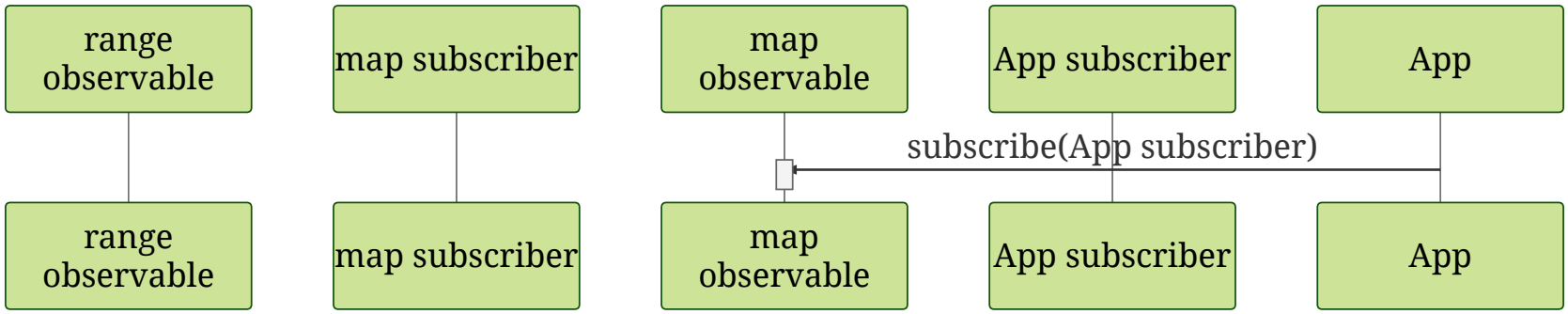
write an algorithm - transform

```
auto map = [](auto selector){

  return [=](auto in){
    return create([=](auto out){

      return in
        | subscribe(
          out.get_subscription(),
          [](auto v){
            out.on_next(selector(v));
          },

          [](exception_ptr ep){out.on_error(ep);},
          []()                {out.on_completed();}

        ); }); }; };
```

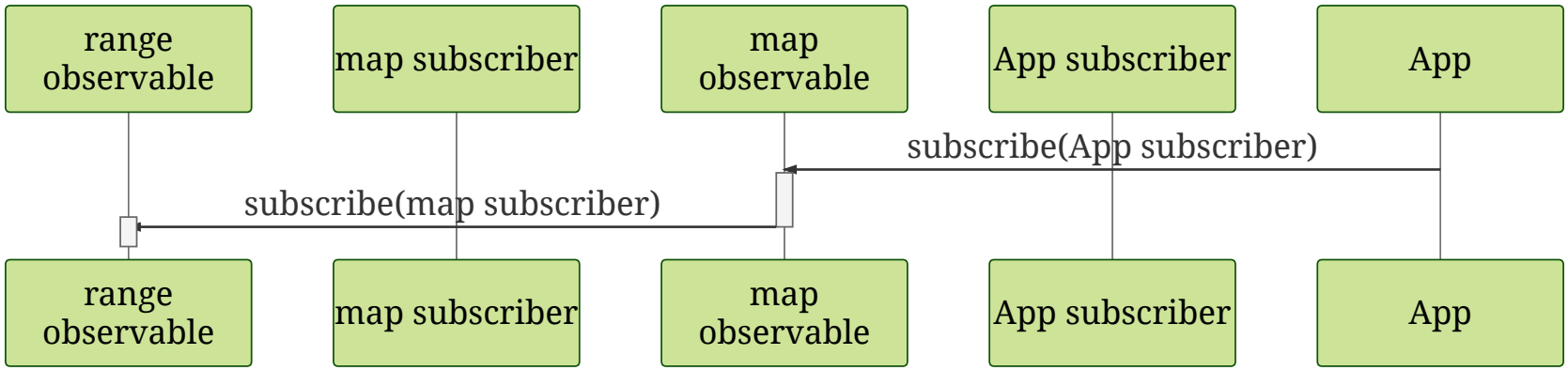
- an operator is a function that takes an observable and returns an observable
- values from `in` are transformed by `selector` and the result passed to `out`
- `on_error` and `on_completed` are passed to `out` unchanged

```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```

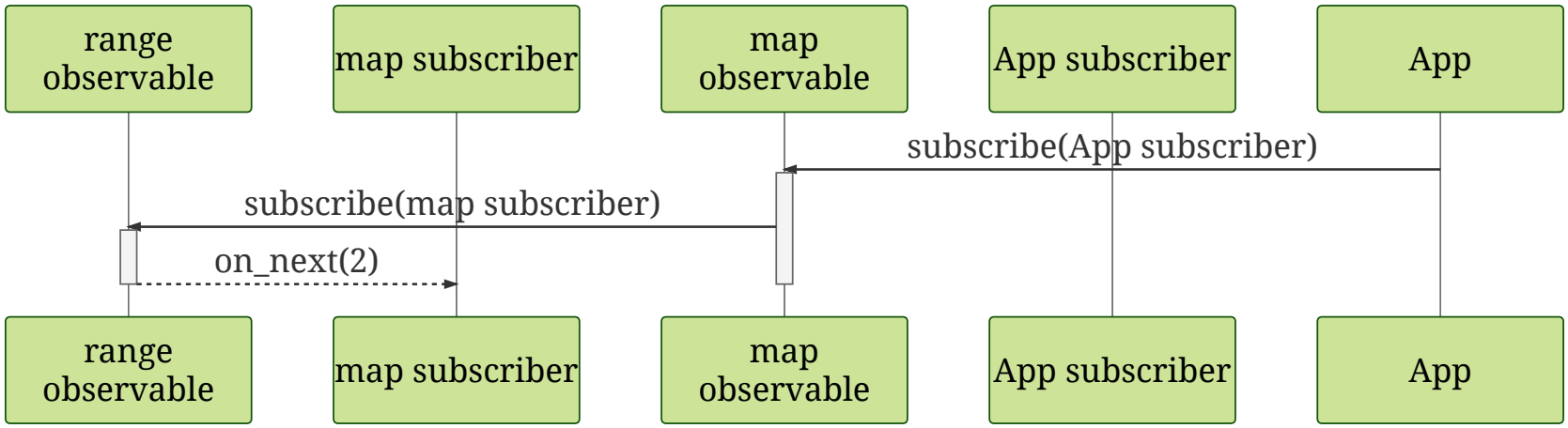
```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```



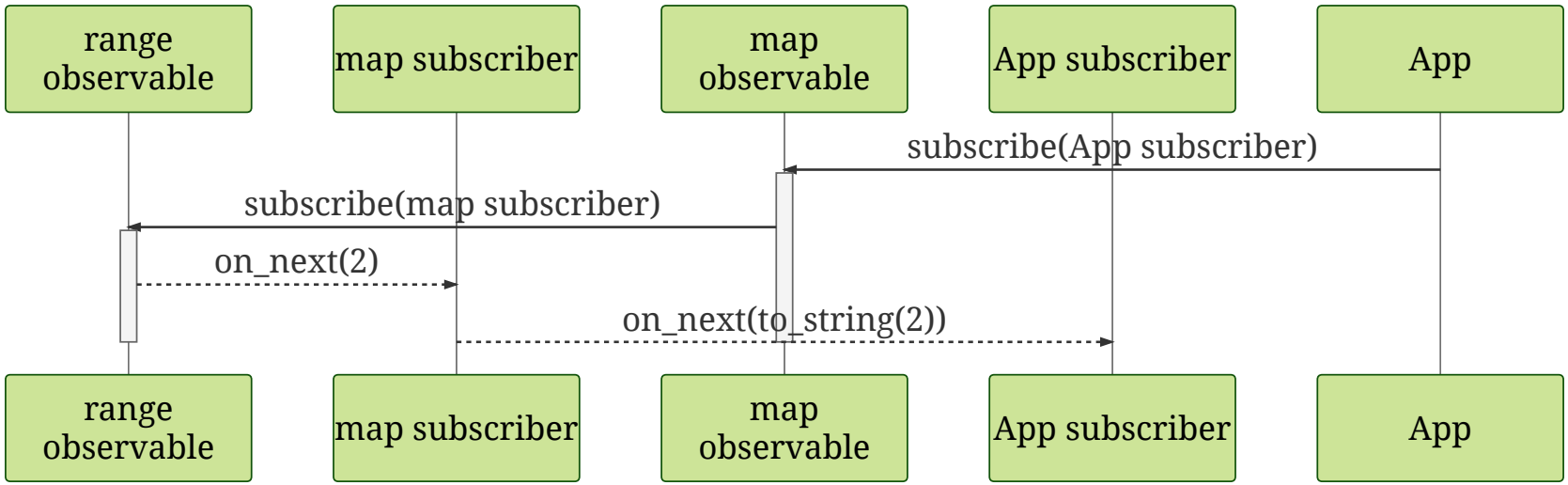
```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```



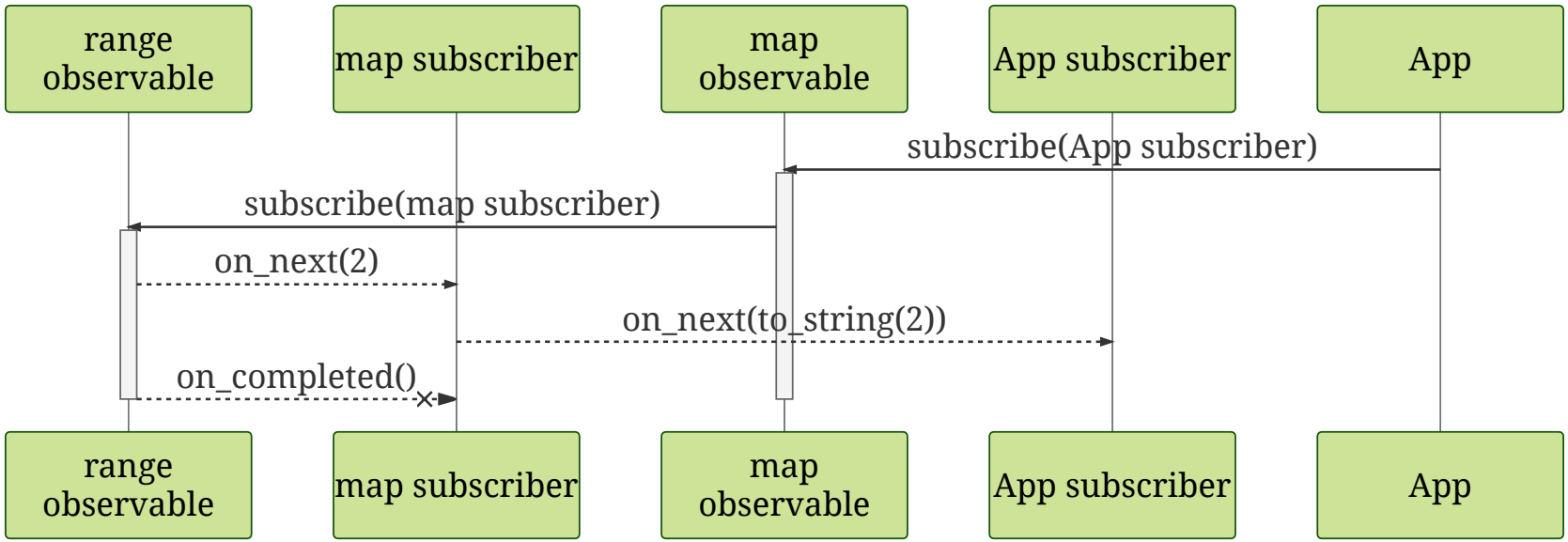
```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```



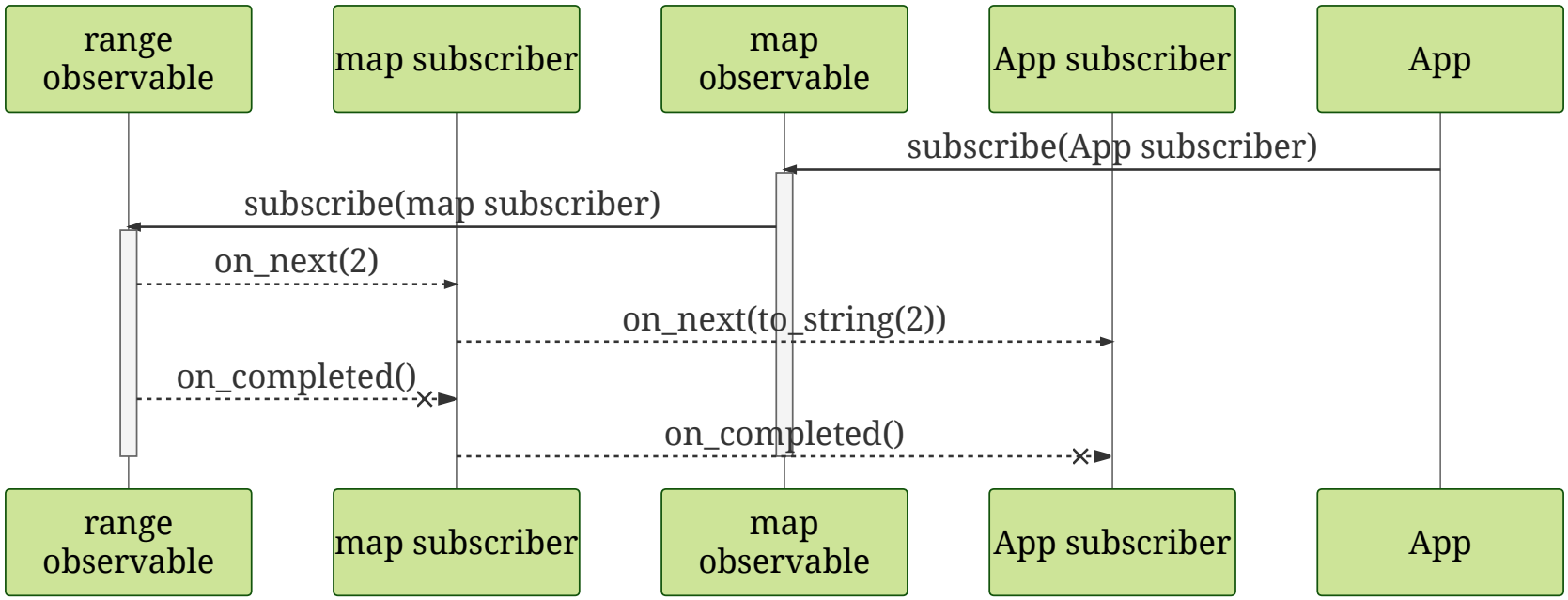
```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```



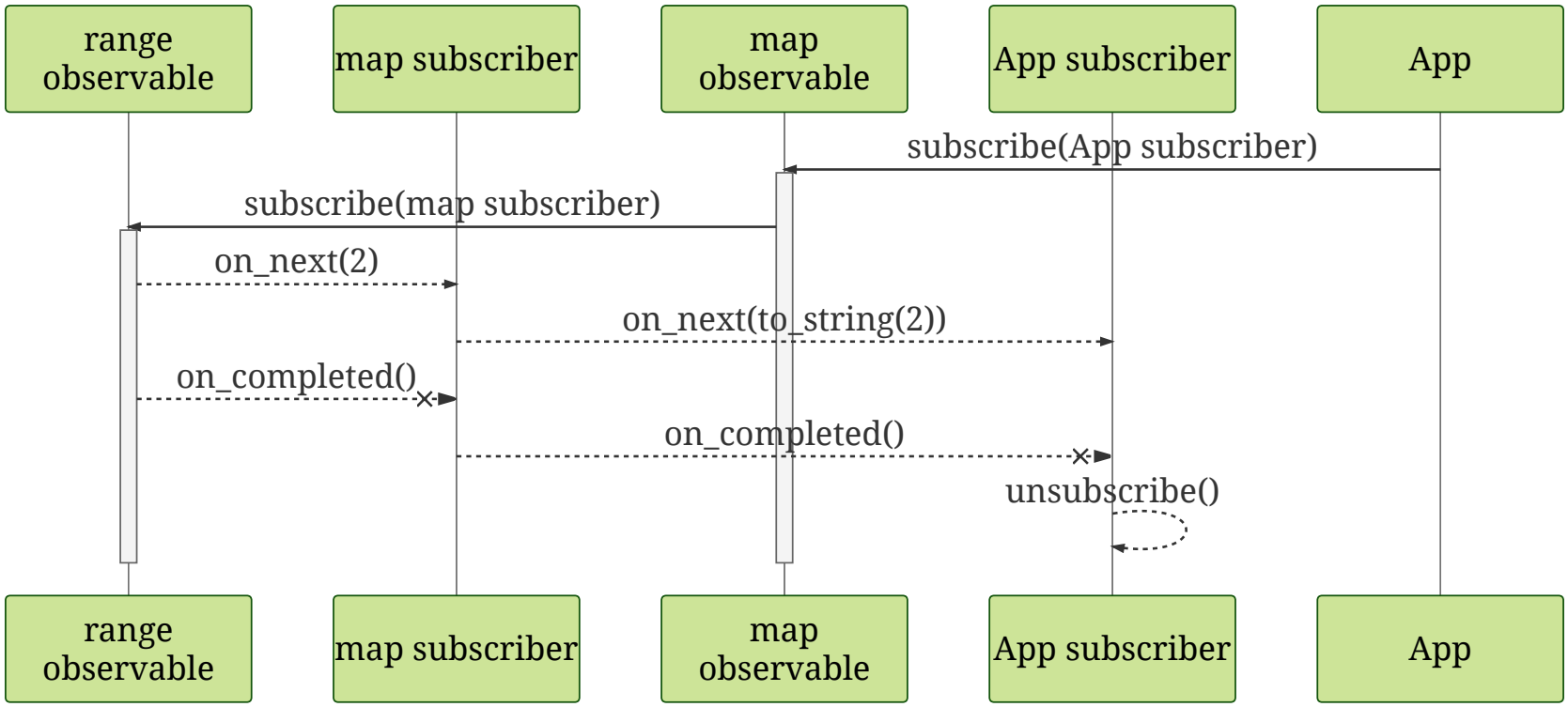
```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```



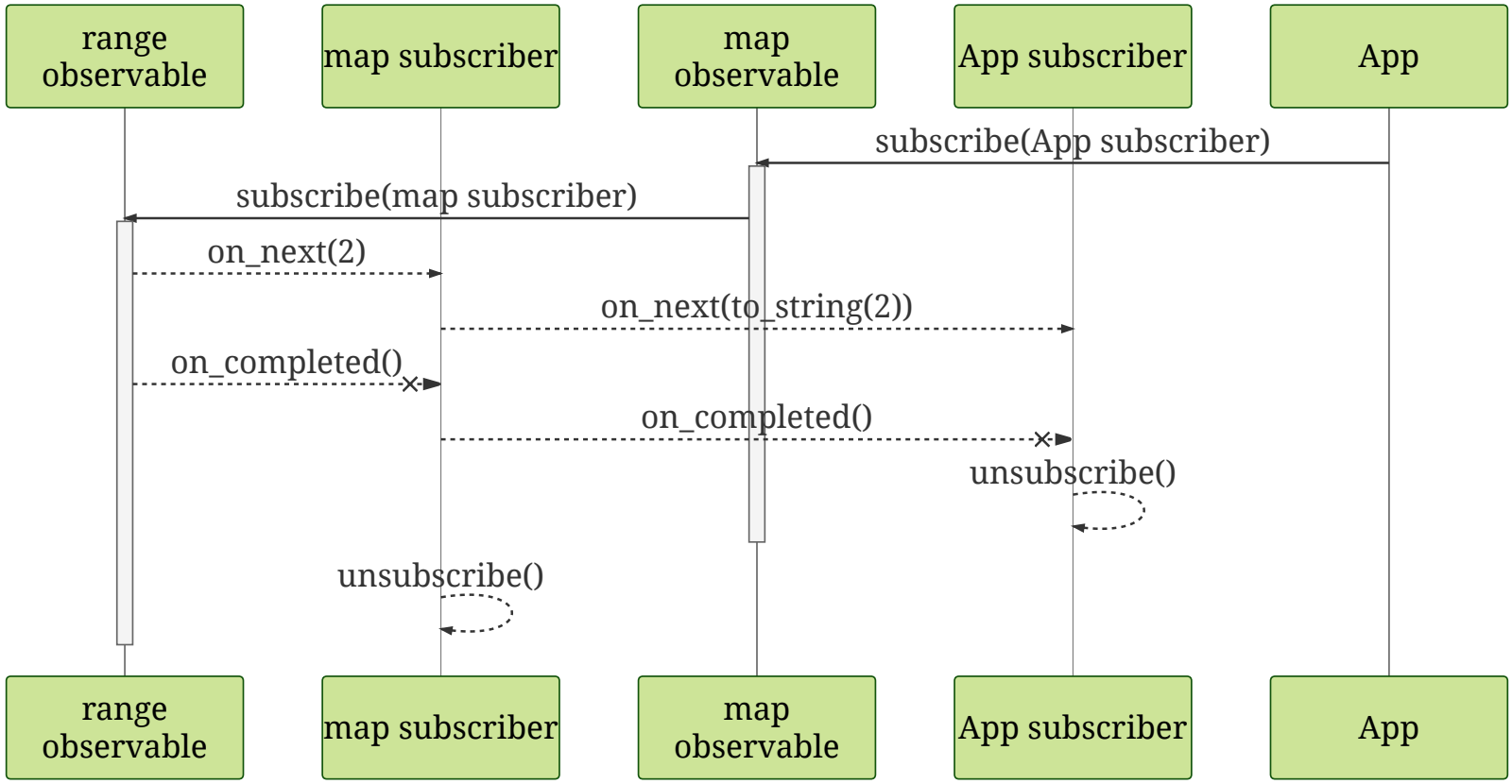
```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```



```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```

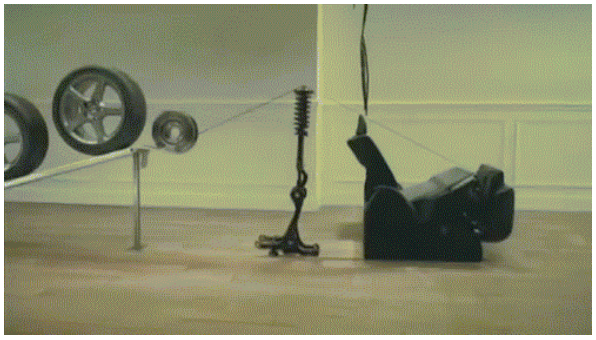


```
range(2, 2) | map([](long l){return to_string(l);}) | subscribe();
```



write an algorithm COMPLETED

- created transform algorithm
- used transform algorithm to change values from long to string
- showed subscription flow through algorithm



next >> virtuous procrastination

virtuous procrastination

how to get the json from stream.twitter.com

how to get the json from stream.twitter.com

```
auto requesttwitterstream = defer([=](){
  auto url = oauth2SignUrl("https://stream.twitter...");
```

- `defer` is used to call `oauth2SignUrl` each time the request is repeated

how to get the json from stream.twitter.com

```
auto requesttwitterstream = defer([=](){
  auto url = oauth2SignUrl("https://stream.twitter...");

  return http.create(http_request{url, method, {}, {}}) |
```

- `defer` is used to call `oauth2SignUrl` each time the request is repeated
- `create` an observable that will start a request when `subscribe` is called

how to get the json from stream.twitter.com

```
auto requesttwitterstream = defer([=](){
  auto url = oauth2SignUrl("https://stream.twitter...");

  return http.create(http_request{url, method, {}, {}}) |

    map([](http_response r){
      return r.body.chunks;
    }) |
```

- `defer` is used to call `oauth2SignUrl` each time the request is repeated
- `create` an observable that will start a request when `subscribe` is called
- `chunks` is an `observable<string>` that emits parts of the body as they arrive

how to get the json from stream.twitter.com

```
auto requesttwitterstream = defer([=](){
  auto url = oauth2SignUrl("https://stream.twitter...");

  return http.create(http_request{url, method, {}, {}}) |

    map([](http_response r){
      return r.body.chunks;
    }) |

    merge(tweetthread);
}) |
twitter_stream_reconnection(tweetthread);
```

- `defer` is used to call `oauth2SignUrl` each time the request is repeated
- `create` an observable that will start a request when `subscribe` is called
- `chunks` is an `observable<string>` that emits parts of the body as they arrive
- emit all tweets on a dedicated thread
- `twitter_stream_reconnection` implements the twitter reconnect protocol for errors

how to handle twitter retry protocol

how to handle twitter retry protocol

```
auto twitter_stream_reconnection = [](auto tweetthread){
    return [=](observable<string> chunks){
        return chunks |
```

- an operator is a function that takes an observable and returns an observable

how to handle twitter retry protocol

```
auto twitter_stream_reconnection = [](auto tweetthread){
    return [=](observable<string> chunks){
        return chunks |

            timeout(90s, tweetthread) |
```

- an operator is a function that takes an observable and returns an observable
- first rule is to reconnect if nothing has arrived for 90 seconds

how to handle twitter retry protocol

```
auto twitter_stream_reconnection = [](auto tweetthread){
    return [=](observable<string> chunks){
        return chunks |

            timeout(90s, tweetthread) |

            on_error_resume_next([=](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, tweetthread);
            }) |
```

- an operator is a function that takes an observable and returns an observable
- first rule is to reconnect if nothing has arrived for 90 seconds
- `twitterRetryAfterHttp` returns an observable that completes after a time (based on the rules)
- `timeout_error` should reconnect now
- unhandled errors are re-thrown

how to handle twitter retry protocol

```
auto twitter_stream_reconnection = [](auto tweetthread){
    return [=](observable<string> chunks){
        return chunks |

            timeout(90s, tweetthread) |

            on_error_resume_next([=](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, tweetthread);
            }) |

            repeat();
}; };
```

- an operator is a function that takes an observable and returns an observable
- first rule is to reconnect if nothing has arrived for 90 seconds
- `twitterRetryAfterHttp` returns an observable that completes after a time (based on the rules)
- `timeout_error` should reconnect now
- unhandled errors are re-thrown
- when the stream completes, repeat the request

virtuous procrastination COMPLETED

defer work

so that work can be..

- repeated
- retried
- shared

next >> opt-in thread-safety

opt-in thread-safety

how to write a twitter app

how to write a twitter app

```
auto tweets = twitterrequest(tweetthread, http) |
  parsetweets(poolthread, tweetthread) |
  publish() | ref_count(); // share
```

- request and parse tweets
- share parsed tweets

how to write a twitter app

```
auto tweets = twitterrequest(tweetthread, http) |
  parsetweets(poolthread, tweetthread) |
  publish() | ref_count(); // share

auto models = iterate(actions /*, currentthread*/) |
  merge(mainthread)|
  scan(Model{}, [=](Model& m, auto f){
    auto r = f(m);
    return r;
  }) |
```

- request and parse tweets
- share parsed tweets
- actions - process tweets into model updates
- run actions on mainthread

how to write a twitter app

```
auto tweets = twitterrequest(tweetthread, http) |
  parsetweets(poolthread, tweetthread) |
  publish() | ref_count(); // share

auto models = iterate(actions /*, currentthread*/) |
  merge(mainthread)|
  scan(Model{}, [=](Model& m, auto f){
    auto r = f(m);
    return r;
  }) |

  sample_with_time(200ms, mainthread) |
  publish() | ref_count(); // share
```

- request and parse tweets
- share parsed tweets
- actions - process tweets into model updates
- run actions on mainthread
- update to the latest model every 200ms and share

how to write a twitter app

```
auto tweets = twitterrequest(tweetthread, http) |
  parsetweets(poolthread, tweetthread) |
  publish() | ref_count(); // share

auto models = iterate(actions /*, currentthread*/) |
  merge(mainthread)|
  scan(Model{}, [=](Model& m, auto f){
    auto r = f(m);
    return r;
  }) |

  sample_with_time(200ms, mainthread) |
  publish() | ref_count(); // share

iterate(renderers /*, currentthread*/) |
  merge(/*currentthread*/) |
  subscribe<Model>();
```

- request and parse tweets
- share parsed tweets
- actions - process tweets into model updates
- run actions on mainthread
- update to the latest model every 200ms and share
- renderers - process the latest model onto the screen
- subscribe starts the app

how to batch calls to sentiment web service

how to batch calls to sentiment web service

```
auto sentimentaction = tweets |
  buffer_with_time(500ms, tweetthread) |
```

- buffer tweets into a vector and emit the vector every 500ms

how to batch calls to sentiment web service

```
auto sentimentaction = tweets |
  buffer_with_time(500ms, tweetthread) |

  filter([](vector<Tweet> v){ return !v.empty(); }) |
  map([=](const vector<Tweet>& buffy) {
```

- buffer tweets into a vector and emit the vector every 500ms
- ignore empty vectors

how to batch calls to sentiment web service

```
auto sentimentaction = tweets |
  buffer_with_time(500ms, tweetthread) |

  filter([](vector<Tweet> v){ return !v.empty(); }) |
  map([=](const vector<Tweet>& buffy) {

    vector<string> text = buffy |
      view::transform(tweettext);
```

- buffer tweets into a vector and emit the vector every 500ms
- ignore empty vectors
- **range-v3** is used to extract a vector of strings from the json

how to batch calls to sentiment web service

```
auto sentimentaction = tweets |
  buffer_with_time(500ms, tweetthread) |

  filter([](vector<Tweet> v){ return !v.empty(); }) |
  map([=](const vector<Tweet>& buffy) {

    vector<string> text = buffy |
      view::transform(tweettext);

    return sentimentrequest(poolthread, http, text) |
      map([=](const string& body){
```

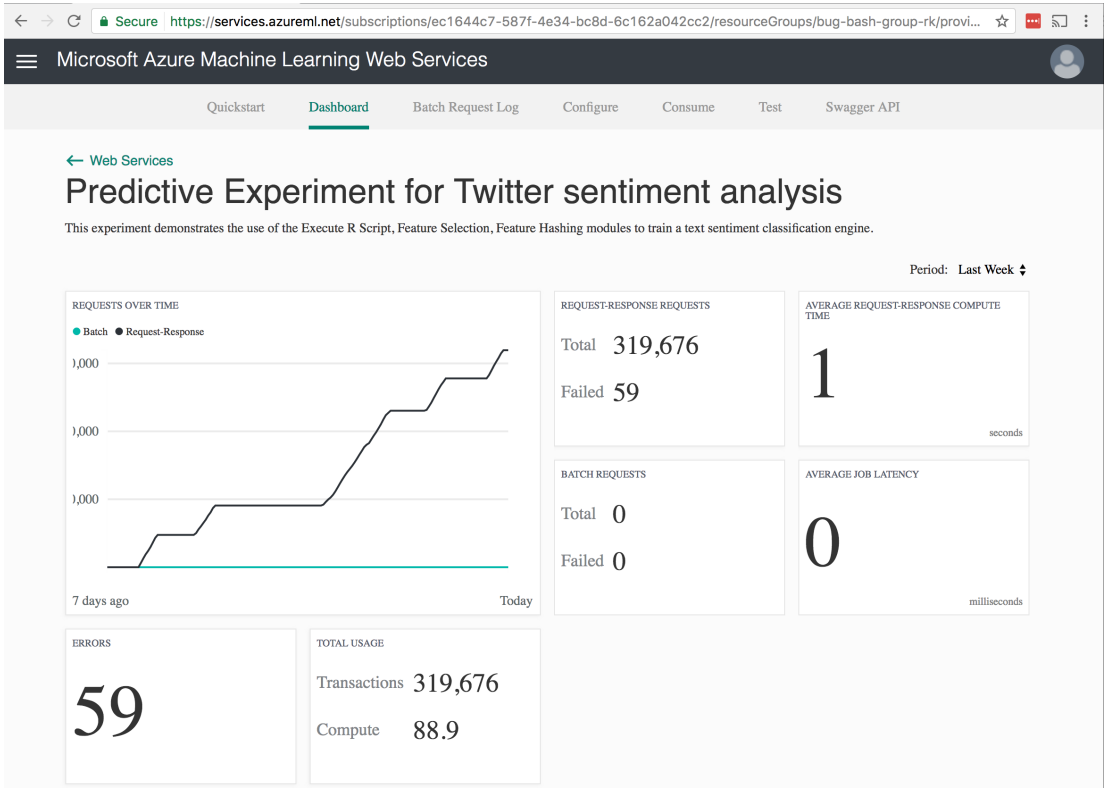
- buffer tweets into a vector and emit the vector every 500ms
- ignore empty vectors
- **range-v3** is used to extract a vector of strings from the json
- send the vector to get a vector of the sentiment of each

how to batch calls to sentiment web service

```
auto sentimentaction = tweets |
  buffer_with_time(500ms, tweetthread) |

  filter([](vector<Tweet> v){ return !v.empty(); }) |
  map([=](const vector<Tweet>& buffy) {

    vector<string> text = buffy |
      view::transform(tweettext);

    return sentimentrequest(poolthread, http, text) |
      map([=](const string& body){

        auto sentiments = json::parse(body);
        auto combined = view::zip(sentiments, buffy);
        // . . .
      }); });
```

- buffer tweets into a vector and emit the vector every 500ms
- ignore empty vectors
- **range-v3** is used to extract a vector of strings from the json
- send the vector to get a vector of the sentiment of each
- parse the sentiment vector from the json
- **range-v3** zips the tweet and sentiment vectors to match the tweet with the sentiment

Azure Machine Learning



Azure Machine Learning

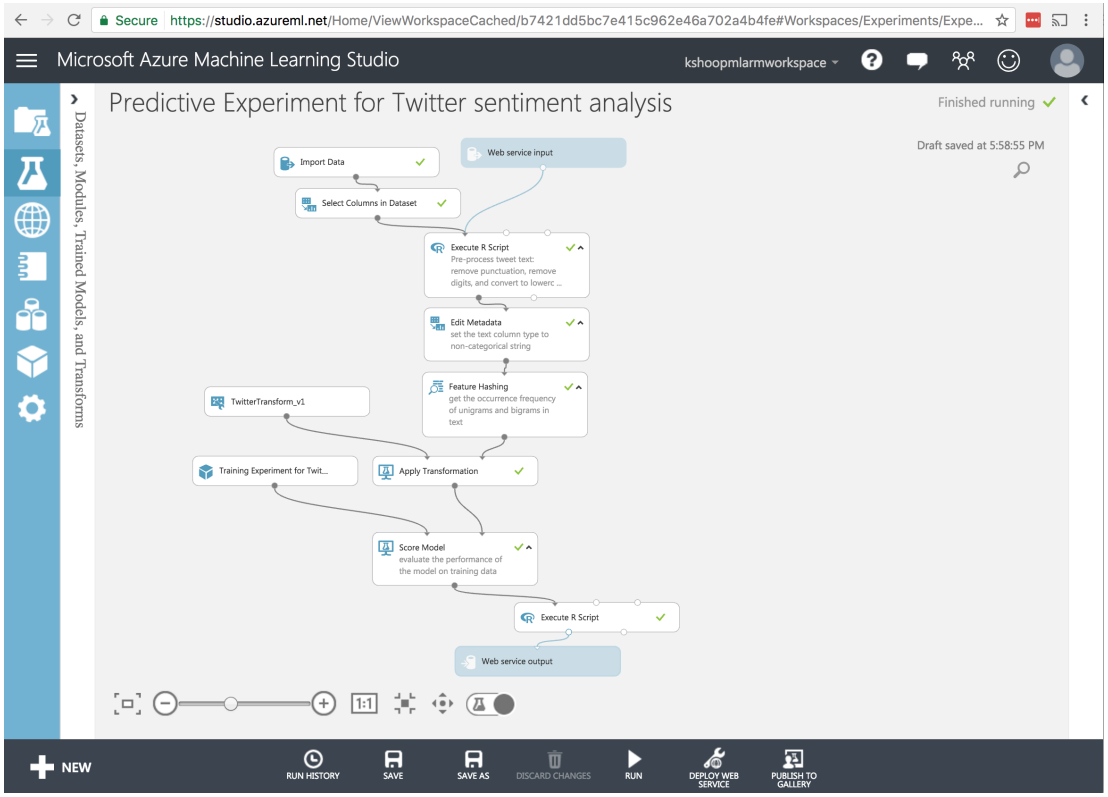
Request

```
POST /subscriptions/<subscription-id>/services/<service-id>/execute HTTP/1.1
Host: ussouthcentral.services.azureml.net
Connection: keep-alive
Content-Length: 148
Authorization: Bearer <api-key>

{"Inputs":{"input1":[{"tweet_text":"..."}]},"GlobalParameters":{}}
```

Response

```
HTTP/1.1 200 OK
Content-Length: 77
Content-Type: application/json; format=swagger; charset=utf-8

{"Results":{"output1":[{"Sentiment":"positive","Score":"0.87877231836319"}]}}
```

Azure Machine Learning

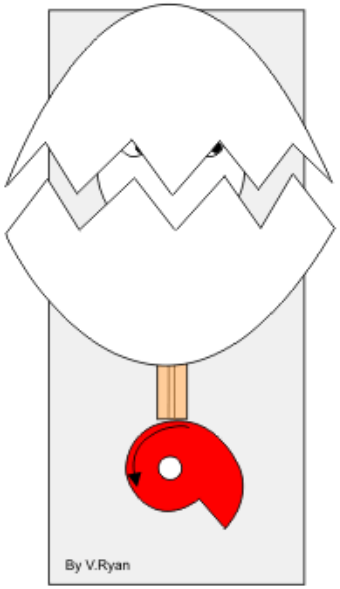


opt-in thread-safety COMPLETED

- described non-thread-safe scheduler default
- specified thread-safe schedulers to coordinate **multiple** streams
- specified thread-safe schedulers to coordinate **time** with streams

next >> adapt asyc sources

adapt async sources



adapt async sources - http requests using libcurl

adapt async sources - http requests using libcurl

- `curl_multi_perform` supports multiplexing requests on a thread

adapt async sources - http requests using libcurl

- `curl_multi_perform` supports multiplexing requests on a thread
- all the calls to curl must be made from that thread

adapt async sources - http requests using libcurl

- `curl_multi_perform` supports multiplexing requests on a thread
- all the calls to curl must be made from that thread
- completion and results must be delivered to the matching request

adapt async sources - http requests using libcurl

```
auto worker = create<CURLMsg*>([](subscriber<CURLMsg*> out){
    while(out.is_subscribed()) {
      curl_multi_perform(curlm, /*. . .*/);
      for(;;) {
        CURLMsg *message = nullptr;
        message = curl_multi_info_read(curlm, /*. . .*/);
        out.on_next(message);
        if (!!message /*. . .*/) { continue; }
        break;
      }
      int handlecount = 0;
      curl_multi_wait(curlm, nullptr, 0, 500, &handlecount);
    }
    out.on_completed();
  }) |
  subscribe_on(httpthread) |
  publish() | connect_forever(); // share
```

adapt async sources - http requests using libcurl

```
auto worker = create<CURLMsg*>([](subscriber<CURLMsg*> out){
    while(out.is_subscribed()) {
      curl_multi_perform(curlm, /*. . .*/);
      for(;;) {
        CURLMsg *message = nullptr;
        message = curl_multi_info_read(curlm, /*. . .*/);
        out.on_next(message);
        if (!!message /*. . .*/) { continue; }
        break;
      }
      int handlecount = 0;
      curl_multi_wait(curlm, nullptr, 0, 500, &handlecount);
    }
    out.on_completed();
  }) |
  subscribe_on(httpthread) |
  publish() | connect_forever(); // share
```

adapt async sources - http requests using libcurl

to create an http request, use `worker` observable to run the curl api calls on the httpthread.

subscribe to http request

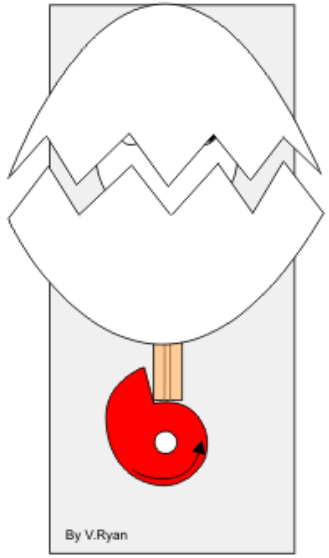
```
worker
  | take(1)
  | tap([] (CURLMsg*){
    auto curl = curl_easy_init();
    curl_easy_setopt(curl,
      CURLOPT_URL, url.c_str());
    // . . .
    curl_multi_add_handle(curlm, curl);
    //
  })
  | subscribe();
```

unsubscribe http request (cancel)

```
worker
  | take(1)
  | tap([] (CURLMsg*){
    //
    //
    //
    //
    curl_multi_remove_handle(curlm, curl);
    curl_easy_cleanup(curl);
  })
  | subscribe();
```

adapt async sources COMPLETED

- adapted libcurl to rxcpp
- built **polling loop** to adapt libcurl to rxcpp
- **callback**, **future**, **completion port**, etc..
  patterns can also be adapted to rxcpp



DROP CAM

next >> algorithmic trancendence

algorithmic trancendence

sample of the algorithms available

http://reactive-extensions.github.io/RxCpp/namespacerxcpp_1_1operators.html

- **buffer** - reactivex.io
- combine_latest - rxmarbles.com
- concat - rxmarbles.com
- concat_map
- **debounce** - rxmarbles.com
- **delay** - rxmarbles.com
- distinct - rxmarbles.com
- **distinct_until_changed** - rxmarbles.com
- element_at - rxmarbles.com
- **filter** - rxmarbles.com
- **finally**
- flat_map - reactivex.io
- **group_by** - reactivex.io
- **ignore_elements** - reactivex.io
- **map** - rxmarbles.com
- **merge** - rxmarbles.com
- observe_on - reactivex.io
- **on_error_resume_next** - reactivex.io
- pairwise
- **publish** - reactivex.io
- reduce - rxmarbles.com
- **repeat**
- replay - reactivex.io
- **retry** - reactivex.io

algorithms used in the twitter app

sample of the algorithms available

http://reactive-extensions.github.io/RxCpp/namespacerxcpp_1_1operators.html

- **sample** - rxmarbles.com
- **scan** - rxmarbles.com
- sequence_equal - reactivex.io
- skip - rxmarbles.com
- skip_last - rxmarbles.com
- skip_until - rxmarbles.com
- **start_with** - rxmarbles.com
- **subscribe_on** - reactivex.io
- switch_if_empty - reactivex.io
- **switch_on_next** - reactivex.io
- take - rxmarbles.com
- take_last - rxmarbles.com
- **take_until** - rxmarbles.com
- take_while - reactivex.io
- time_interval - reactivex.io
- **timeout** - reactivex.io
- timestamp - reactivex.io
- **window** - reactivex.io
- **with_latest_from** - rxmarbles.com
- zip - rxmarbles.com

algorithms used in the twitter app

learn algorithms once, use them in any Language

- Java: https://github.com/ReactiveX/RxJava
- JavaScript: https://github.com/Reactive-Extensions/RxJS, https://github.com/ReactiveX/RxJS
- C#: https://github.com/Reactive-Extensions/Rx.NET
- C#(Unity): https://github.com/neuecc/UniRx
- Scala: https://github.com/ReactiveX/RxScala
- Clojure: https://github.com/ReactiveX/RxClojure
- C++: https://github.com/Reactive-Extensions/RxCpp
- Lua: https://github.com/bjornbytes/RxLua
- Ruby: https://github.com/Reactive-Extensions/Rx.rb
- Python: https://github.com/ReactiveX/RxPY
- Go: https://github.com/ReactiveX/RxGo
- Groovy: https://github.com/ReactiveX/RxGroovy
- JRuby: https://github.com/ReactiveX/RxJRuby
- Kotlin: https://github.com/ReactiveX/RxKotlin
- Swift: https://github.com/kzaher/RxSwift
- PHP: https://github.com/ReactiveX/RxPHP
- Elixir: https://github.com/alfert/reaxive
- Dart: https://github.com/ReactiveX/rxdart

use algorithms to solve a problem once, reuse in any Language

use algorithms to solve a problem once, reuse in any Language

parsing messages out of chunks of characters

http://stackoverflow.com/questions/31208418/split-iobservablebyte-to-characters-then-to-line

> I receive events with a `byte[]`, this array might contains part of a line, multiple lines or one line. What I want is find a way to have an `IObservable` of Line so `IObservable<String>`, where each element of the sequence will be a line.

use algorithms to solve a problem once, reuse in any Language

parsing messages out of chunks of characters

http://stackoverflow.com/questions/31208418/split-iobservablebyte-to-characters-then-to-line

> I receive events with a `byte[]`, this array might contains part of a line, multiple lines or one line. What I want is find a way to have an `IObservable` of Line so `IObservable<String>`, where each element of the sequence will be a line.

https://dev.twitter.com/streaming/overview/processing

> The body of a streaming API response consists of a series of newline-delimited messages, where “newline” is considered to be `\r\n` (in hex, `0x0D 0x0A`) and “message” is a JSON encoded data structure or a blank line.
>
> Note that Tweet content may sometimes contain linefeed `\n` characters, but will not contain carriage returns `\r`. Therefore, to make sure you get whole message payloads, break out each message on `\r\n` boundaries, as `\n` may occur in the middle of a message.

C#

from the StackOverflow answer

```
var strings = bytes.
  Select(arr =>
    (Regex.Split(
        Encoding.Default.
            GetString(arr, 0, arr.Length - 1),
        "(\r)")).
    Where(s=> s.Length != 0).
    ToObservable()).
  Concat().
  Publish().
  RefCount();

var closes = strings.
    Where(s => s.EndsWith("\r"));

var linewindows = strings.Window(closes);


var lines = linewindows.SelectMany(w =>
    w.Aggregate((l, r) => l + r));
```

C++

code to extract tweets in the twitter app

```
auto strings = chunks |
    concat_map([](const string& s){
        auto splits = split(s, "\r\n");
        return iterate(move(splits));
    }) |
    filter([](const string& s){
        return !s.empty();
    }) |
    publish() |
    ref_count();

auto closes = strings |
    filter(isEndOfTweet);

auto linewindows = strings |
    window_toggle(closes | start_with(0),
        [=](int){return closes;});

auto lines = linewindows |
    flat_map([](const observable<string>& w) {
        return w | start_with("") | sum();
    });
```

algorithmic trancendence COMPLETED

- sampled available algorithms
- showed **languages** with ReactiveX implementations
- built **message parsing** in C# and C++

next >> recap

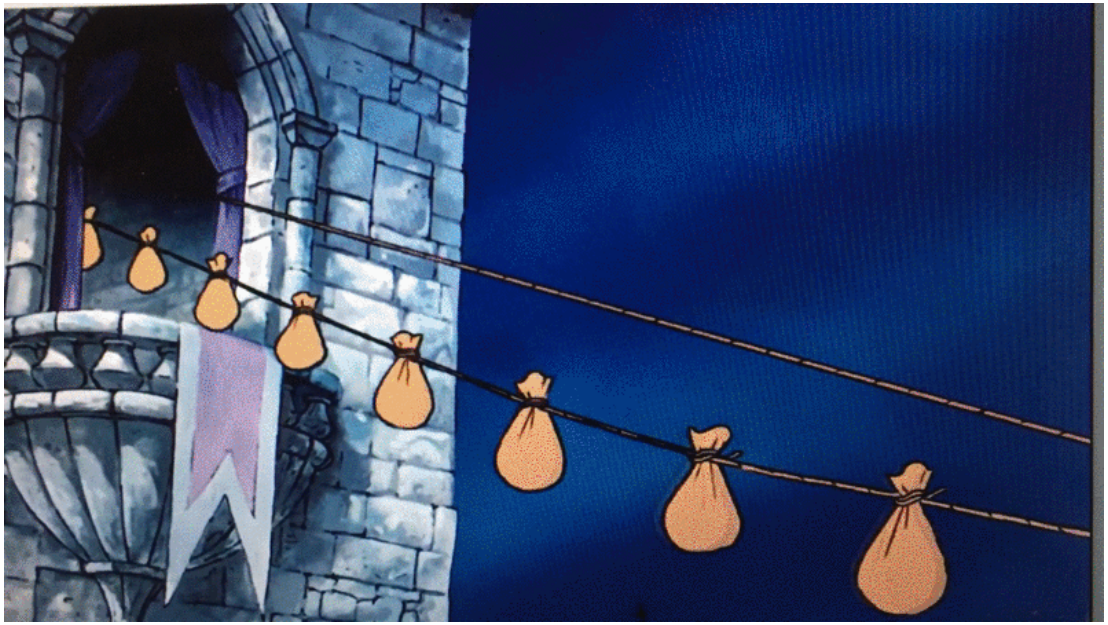
recap

recap

primitives are too primitive

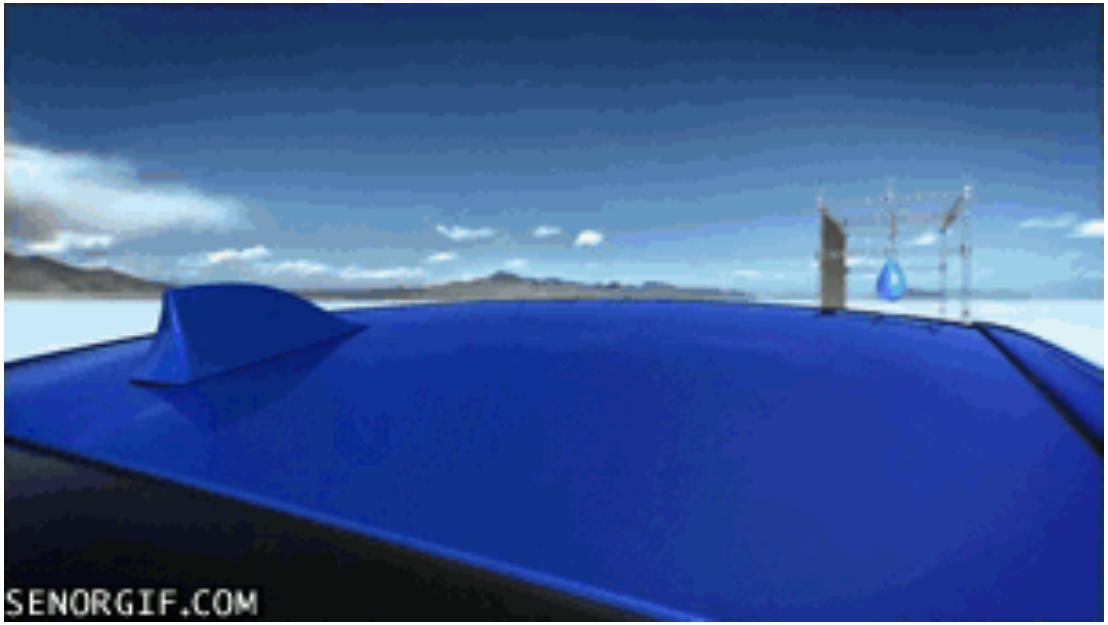
recap

handling Tweets



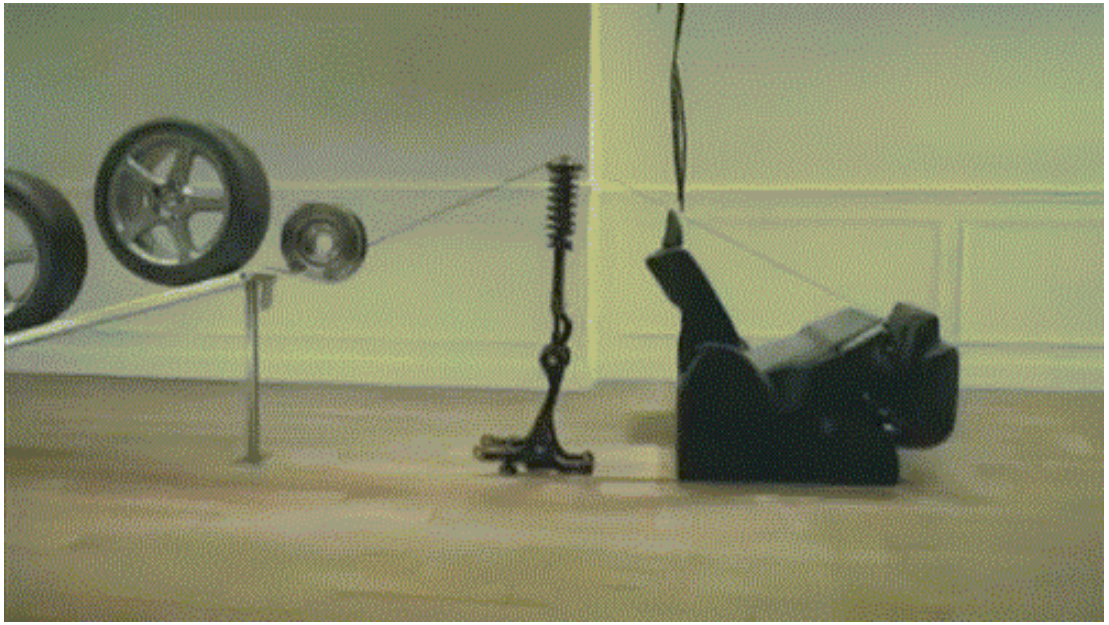
recap

values distributed in time



recap

write an algorithm



recap

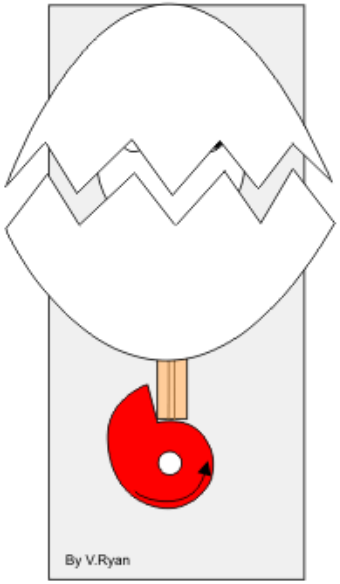
virtuous procrastination

recap

opt-in thread-safety

recap

adapt existing sources



recap

algorithmic trancendence

things I desire

- REST service library
- http service library
- bindings for ux libraries, like https://github.com/tetsurom/rxqt
- bindings for asio, like https://github.com/pudae/example
- rxcpp v3 https://github.com/kirkshoop/rxcppv3
- standardization
- native algorithm support in libraries (asio, boost, poco, etc..)

credits

Eric Mittelette shaped this presentation from start to finish. I am deeply grateful for all his time and effort.

Niall Connaughton presented the `RxJS` twitter analisys app that inspired me to build one with `rxcpp`

Aaron Lahman made the first prototype of `rxcpp`.

Grigoriy Chudnov, **Valery Kopylov** and many other `rxcpp` contributors..

resources

- https://github.com/kirkshoop/twitter
- https://github.com/Reactive-Extensions/RxCpp
- http://reactive-extensions.github.io/RxCpp/
- https://github.com/kirkshoop/rxcppv3
- http://rxmarbles.com/
- http://reactivex.io/intro.html
- http://reactivex.io/learnrx/

Lightning talk slides

errors - forgotten, but not gone

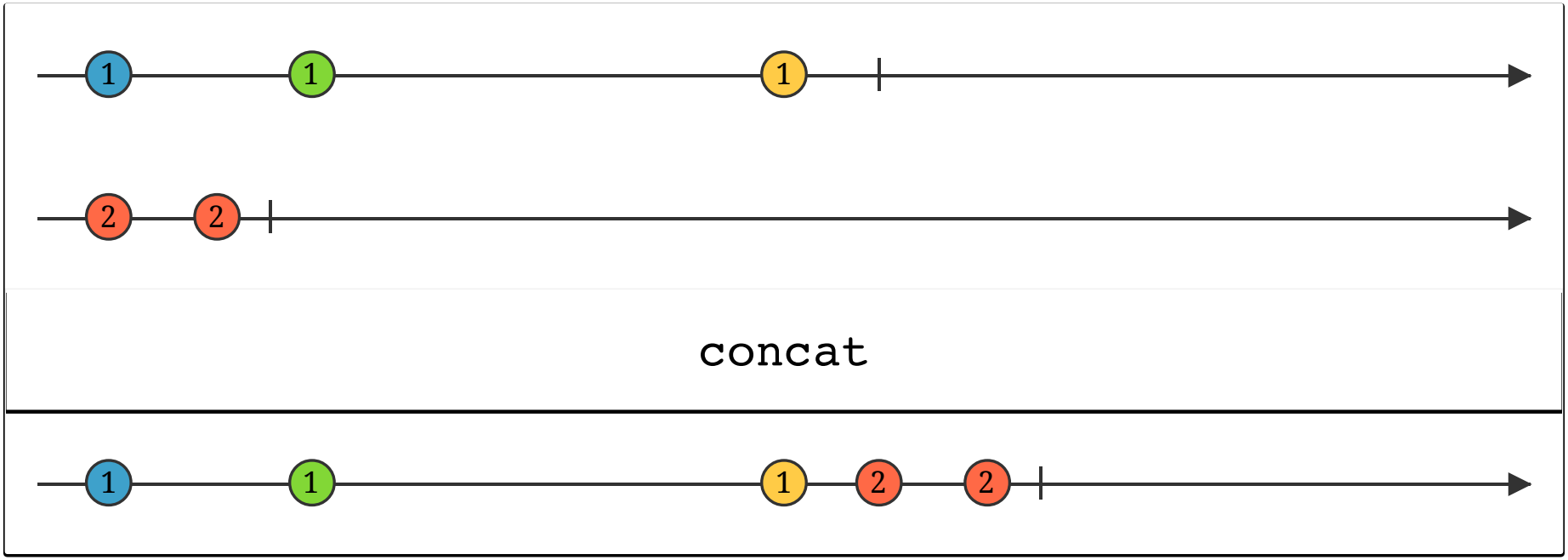
https://kirkshoop.github.io/norawthread/errors.html

Networking TS w/Algorithms

https://kirkshoop.github.io/norawthread/rxnetts.html

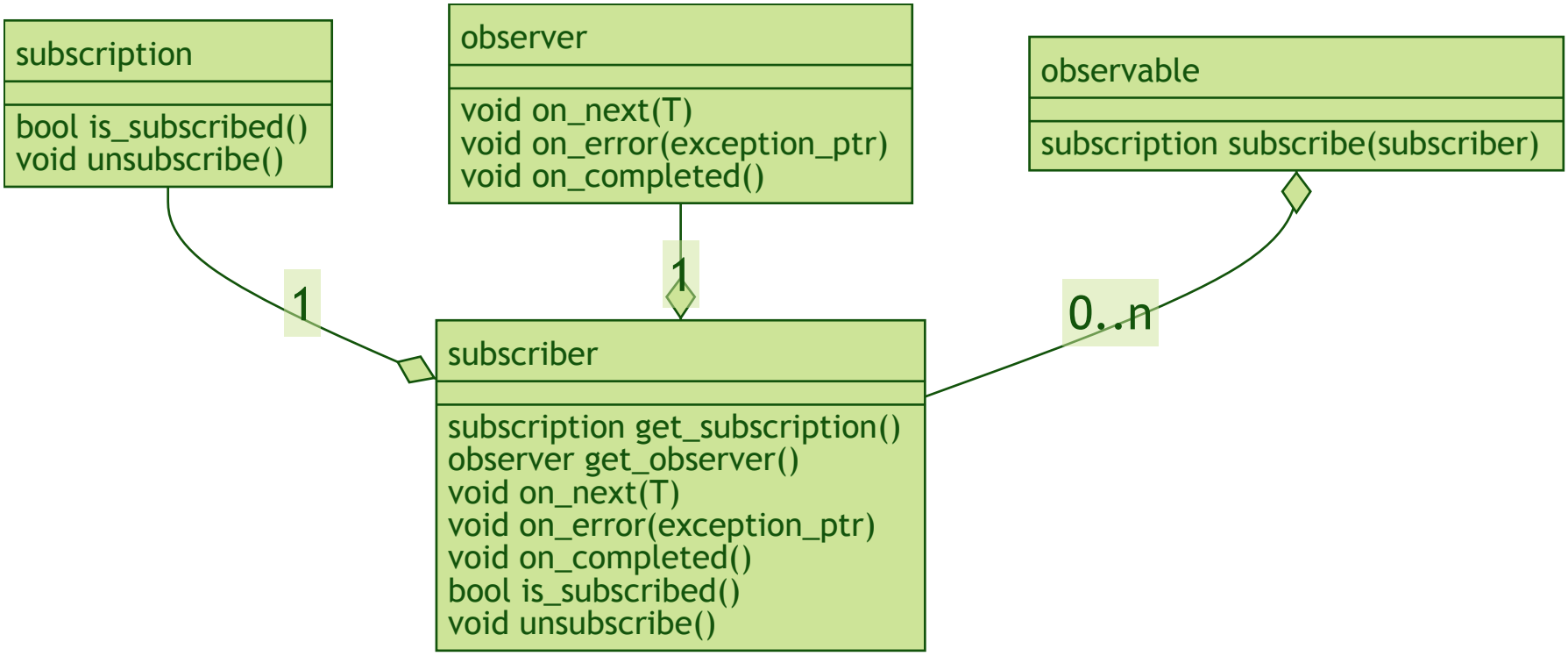
complete.

questions?

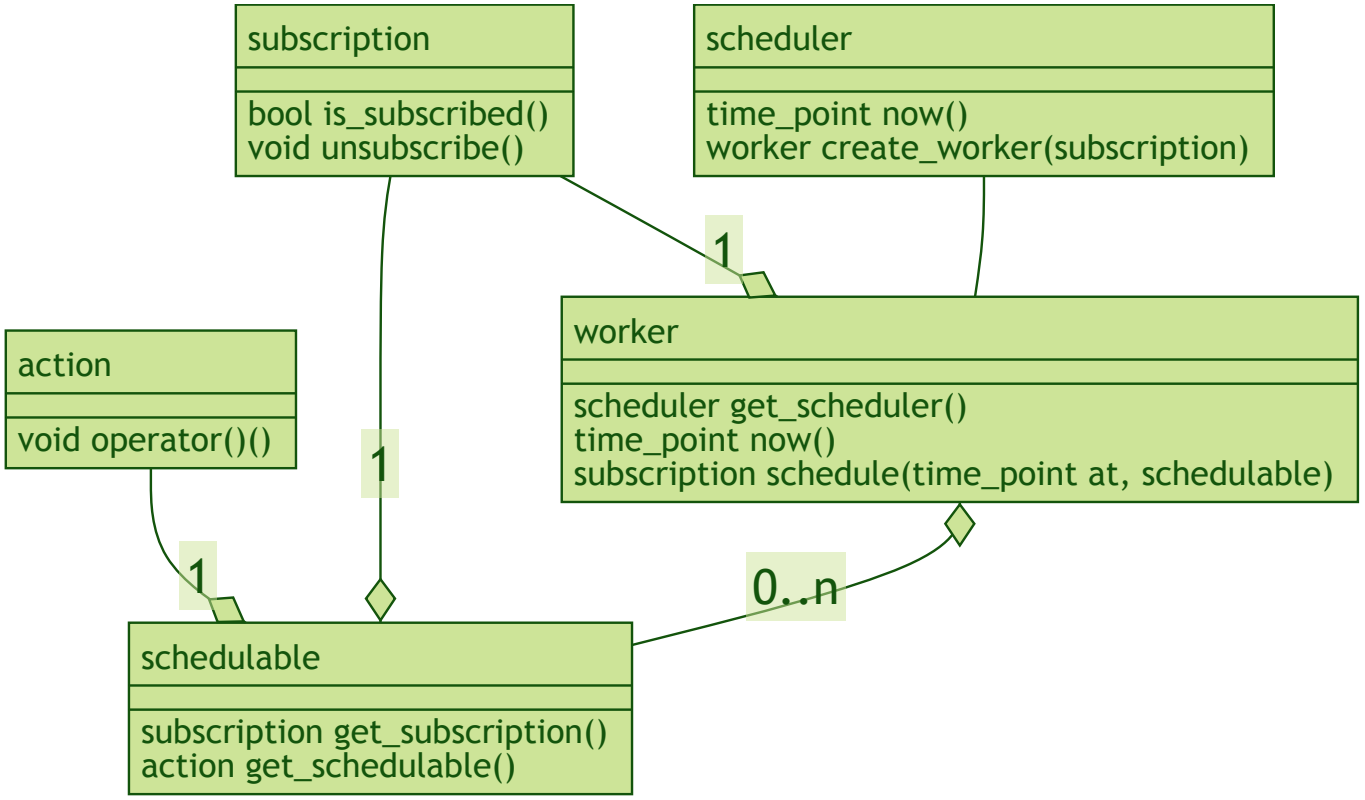


appendix

rxcpp architecture



rxcpp scheduler architecture



how to call sentiment web service

how to call sentiment web service

```
auto requestsentiment = defer([=]() {
```

how to call sentiment web service

```
auto requestsentiment = defer([=]() {

    std::map<string, string> headers;
    headers["Content-Type"] = "application/json";
    headers["Authorization"] = "Bearer " + key;
```

how to call sentiment web service

```
auto requestsentiment = defer([=]() {

    std::map<string, string> headers;
    headers["Content-Type"] = "application/json";
    headers["Authorization"] = "Bearer " + key;

    auto body = json::parse(
        R"({"Inputs":{"input1":[{"tweet_text": "Hi!"}]},"GlobalParameters":{}})"
    );
```

how to call sentiment web service

```
auto requestsentiment = defer([=]() {

    std::map<string, string> headers;
    headers["Content-Type"] = "application/json";
    headers["Authorization"] = "Bearer " + key;

    auto body = json::parse(
        R"({"Inputs":{"input1":[{"tweet_text": "Hi!"}]},"GlobalParameters":{}})"
    );

    return http.create(http_request{url, "POST", headers, body.dump()}) |
        map([](http_response r){
            return r.body.complete;
        }) |
        merge(poolthread);
});
```

delegate task execution

delegate task execution - asio `io_service`

https://github.com/pudae/example/blob/master/rx_test/rxasio/io_service

delegate task execution - asio `io_service`

https://github.com/pudae/example/blob/master/rx_test/rxasio/io_service

```
class io_service : public scheduler_interface
{
  asio::io_service& io_service_;

  class strand_worker : public worker_interface;

public:
  explicit io_service(asio::io_service& io_service);
```

delegate task execution - asio `io_service`

https://github.com/pudae/example/blob/master/rx_test/rxasio/io_service

```
class io_service : public scheduler_interface
{
  asio::io_service& io_service_;

  class strand_worker : public worker_interface;

public:
  explicit io_service(asio::io_service& io_service);

  virtual clock_type::time_point now() const {
    return clock_type::now();
  }
```

delegate task execution - asio `io_service`

https://github.com/pudae/example/blob/master/rx_test/rxasio/io_service

```
class io_service : public scheduler_interface
{
  asio::io_service& io_service_;

  class strand_worker : public worker_interface;

public:
  explicit io_service(asio::io_service& io_service);

  virtual clock_type::time_point now() const {
    return clock_type::now();
  }

  virtual worker create_worker(composite_subscription cs) const {
    return worker(move(cs), make_shared<strand_worker>(io_service_));
  }
};
```

delegate task execution - asio `io_service`

https://github.com/pudae/example/blob/master/rx_test/rxasio/io_service

```
class strand_worker : public worker_interface
{
    mutable asio::strand strand_;

public:
    explicit strand_worker(asio::io_service& io_service);

    virtual clock_type::time_point now() const {
        return clock_type::now();
    }

    virtual void schedule(const schedulable& scbl) const;

    virtual void schedule(clock_type::time_point when,
                          const schedulable& scbl) const;
};
```

- a worker must ensure that only one schedulable is called at a time
- this worker uses asio::strand to order the calls
- this worker uses real time for now()

delegate task execution - asio `io_service`

https://github.com/pudae/example/blob/master/rx_test/rxasio/io_service

```
virtual void schedule(const schedulable& scbl) const {
  if (!scbl.is_subscribed()) return;

  strand_.post([scbl] {
    if (scbl.is_subscribed()) {
      scbl();
    }
  });
}
```

delegate task execution - asio `io_service`

https://github.com/pudae/example/blob/master/rx_test/rxasio/io_service

```
virtual void schedule(clock_type::time_point when,
                      const schedulable& scbl) const {
  if (!scbl.is_subscribed()) return;

  auto diff_ms = duration_cast<milliseconds>(when - now());
  auto timer = make_shared<asio::deadline_timer>(strand_.get_io_service());

  timer->expires_from_now(diff_ms);

  timer->async_wait(strand_.wrap([timer, scbl](const system::error_code& ec) {
    if (!ec && scbl.is_subscribed())
    {
      scbl();
    }
  }));
}
```

delegate task execution

- delegate tasks to; thread pool, event loop, actor, channel, etc..
- implement a scheduler to delegate tasks

```
auto now() -> time_point;

auto schedule(schedulable what) -> void;

auto schedule(time_point at,
              schedulable what) -> void;
```

delegate task execution COMPLETED

described `scheduler` and `worker`

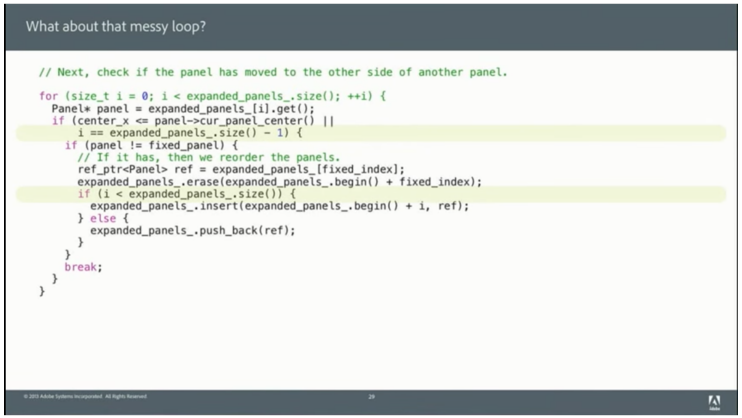
implemented `scheduler` and `worker` to delegate tasks to `asio::io_service`

next >> adapt async sources

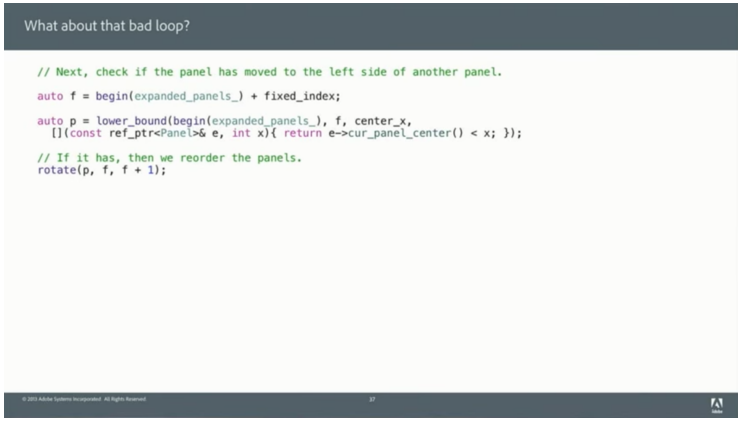
recap

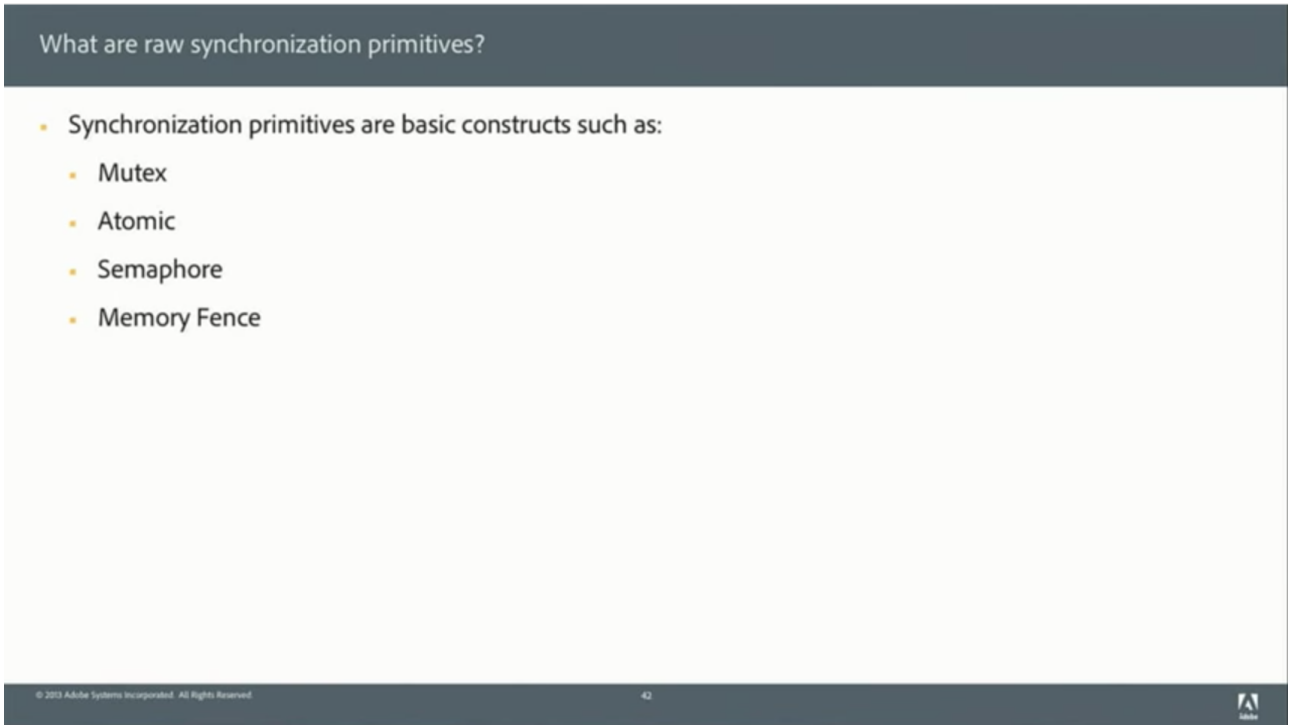
delegate task execution

Raw Loop



Algorithms




What are raw synchronization primitives?
Synchronization primitives are basic constructs such as:
Mutex
Atomic
Semaphore
Memory Fence
© 2013 Adobe Systems Incorporated. All Rights Reserved.
42

sending telemetry from client page

```
var telemetry = new Rx.Subject<TelemetryData>();
```

sending telemetry from client page

```
var telemetry = new Rx.Subject<TelemetryData>();

export function writeEntry(data: ITelemetryData) {
    telemetry.onNext(new TelemetryData(data));
}
```

sending telemetry from client page

```
var pending: { bodysize: number, entries: TelemetryData[]; } = {
    bodysize: 0,
    entries: []
};
```

sending telemetry from client page

```
var pending: { bodysize: number, entries: TelemetryData[]; } = {
    bodysize: 0,
    entries: []
};

var maxTimeTrigger = Rx.Observable.interval(60 * 1000);
```

sending telemetry from client page

```
var pending: { bodysize: number, entries: TelemetryData[]; } = {
    bodysize: 0,
    entries: []
};

var maxTimeTrigger = Rx.Observable.interval(60 * 1000);

var maxCountTrigger = telemetry.filter(e => pending.entries.length >= 50).map(e => -1);
```

sending telemetry from client page

```
var pending: { bodysize: number, entries: TelemetryData[]; } = {
    bodysize: 0,
    entries: []
};

var maxTimeTrigger = Rx.Observable.interval(60 * 1000);

var maxCountTrigger = telemetry.filter(e => pending.entries.length >= 50).map(e => -1);

var maxSizeTrigger = telemetry.filter(e => pending.bodysize >= 20000).map(e => -2);
```

sending telemetry from client page

```
var pending: { bodysize: number, entries: TelemetryData[]; } = {
    bodysize: 0,
    entries: []
};

var maxTimeTrigger = Rx.Observable.interval(60 * 1000);

var maxCountTrigger = telemetry.filter(e => pending.entries.length >= 50).map(e => -1);

var maxSizeTrigger = telemetry.filter(e => pending.bodysize >= 20000).map(e => -2);

var boundaries = Rx.Observable.merge(maxTimeTrigger, maxCountTrigger, maxSizeTrigger).share();
```

```
export var telemetryUpdate$ = telemetry.
    window(boundaries.startWith(-1), () => boundaries).
```

```
export var telemetryUpdate$ = telemetry.
    window(boundaries.startWith(-1), () => boundaries).

    flatMap(w => w.reduce((data, entry) => {
        data.bodysize += JSON.stringify(entry).length;
        data.entries.push(entry);
        return data;
    }, pending)).
    filter(data => data.entries.length !== 0).
```

```
export var telemetryUpdate$ = telemetry.
    window(boundaries.startWith(-1), () => boundaries).

    flatMap(w => w.reduce((data, entry) => {
        data.bodysize += JSON.stringify(entry).length;
        data.entries.push(entry);
        return data;
    }, pending)).
    filter(data => data.entries.length !== 0).

    flatMap(data => Rx.Observable.
        fromPromise(flushTelemetry(FlushRequest.Async)).
```

```
export var telemetryUpdate$ = telemetry.
    window(boundaries.startWith(-1), () => boundaries).

    flatMap(w => w.reduce((data, entry) => {
        data.bodysize += JSON.stringify(entry).length;
        data.entries.push(entry);
        return data;
    }, pending)).
    filter(data => data.entries.length !== 0).

    flatMap(data => Rx.Observable.
        fromPromise(flushTelemetry(FlushRequest.Async)).

    // exponential backoff on failure
        retryWhen(errors => errors.
            scan((c, e) => c + 1, 0).
            flatMap(i => Rx.Observable.timer(Math.pow(5, i) * 1000))).
```

```
export var telemetryUpdate$ = telemetry.
    window(boundaries.startWith(-1), () => boundaries).

    flatMap(w => w.reduce((data, entry) => {
        data.bodysize += JSON.stringify(entry).length;
        data.entries.push(entry);
        return data;
    }, pending)).
    filter(data => data.entries.length !== 0).

    flatMap(data => Rx.Observable.
        fromPromise(flushTelemetry(FlushRequest.Async)).

    // exponential backoff on failure
        retryWhen(errors => errors.
            scan((c, e) => c + 1, 0).
            flatMap(i => Rx.Observable.timer(Math.pow(5, i) * 1000))).

    // give up
        timeout(10 * 60 * 1000, Rx.Observable.empty())).
```

```
export var telemetryUpdate$ = telemetry.
    window(boundaries.startWith(-1), () => boundaries).

    flatMap(w => w.reduce((data, entry) => {
        data.bodysize += JSON.stringify(entry).length;
        data.entries.push(entry);
        return data;
    }, pending)).
    filter(data => data.entries.length !== 0).

    flatMap(data => Rx.Observable.
        fromPromise(flushTelemetry(FlushRequest.Async)).

    // exponential backoff on failure
        retryWhen(errors => errors.
            scan((c, e) => c + 1, 0).
            flatMap(i => Rx.Observable.timer(Math.pow(5, i) * 1000))).

    // give up
        timeout(10 * 60 * 1000, Rx.Observable.empty())).

    retry();

var subscription = telemetryUpdate$.subscribe();
```