ASYNC SEQUENCES AND ALGORITHMS

BUILDING USABLE ASYNC PRIMITIVES

Created by Kirk Shoop / @kirkshoop

USE ALL AVAILABLE RESOURCES

INDEPENDENT AND EAGER

Photo Credit: Olgierd Pstrykotwórca cc

USE MINIMAL RESOURCES

DEPENDENT AND LAZY

Photo Credit: Rob Grambau cc

ASYNCHRONOUS AND CONCURRENT

Photo Credit: Carlos Caicedo cc

STREAMS ARE NOT ALWAYS BYTES

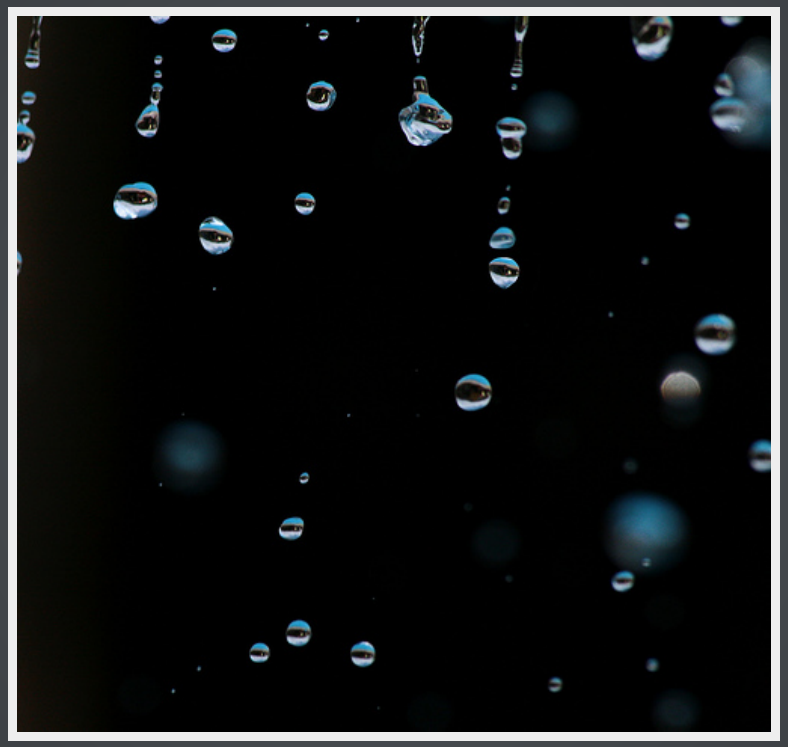
Photo Credit: jeffk cc

ALL ABOUT THE ALGORITHMS

Movie Credit: Office Space

TIME-INDEXED SEQUENCES OF <T>

Photo Credit: Rik Hermans cc

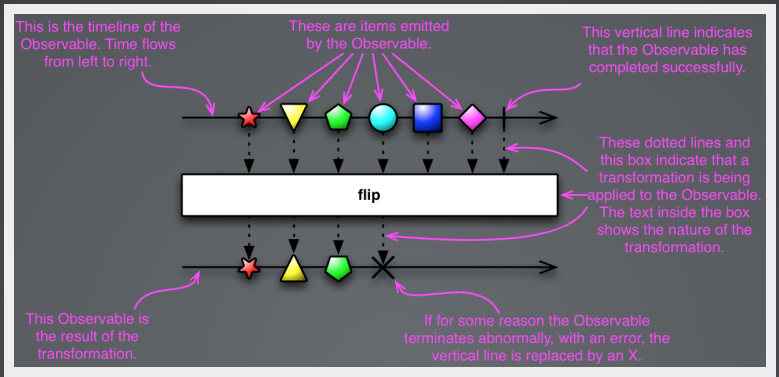


EXAMPLES

- Periodic Timer
- Mouse Clicks
- Mouse Moves
- Network Packets
- File Reads/Writes
- . . .

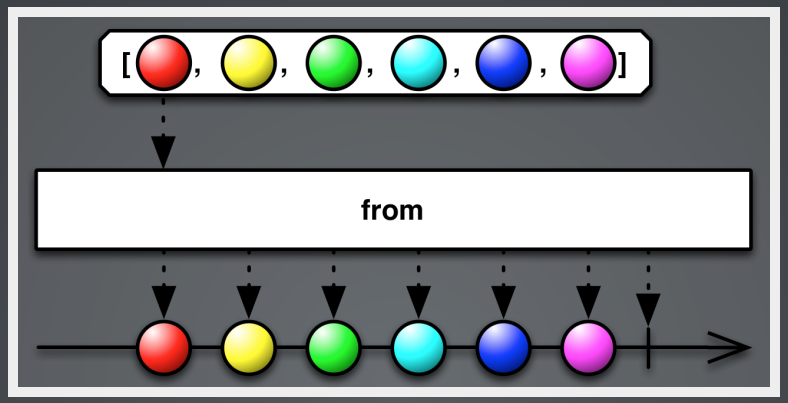
DESCRIBING SEQUENCES IN TIME

Credit: RxJava Wiki



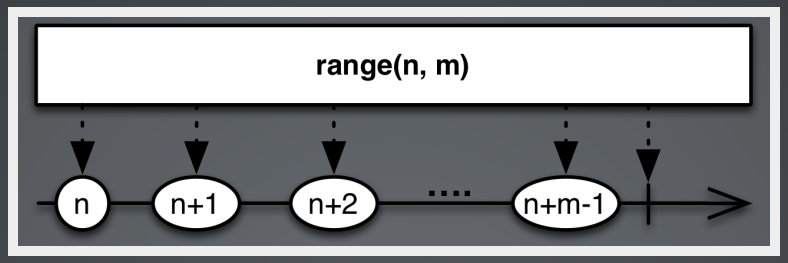
VALUE SEQUENCE DIAGRAM

Credit: RxJava Wiki



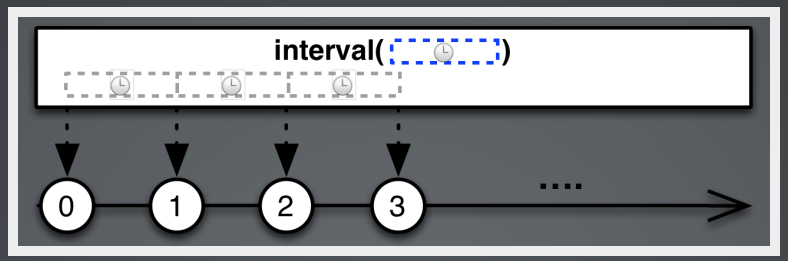
VALUE GENERATOR DIAGRAM

Credit: RxJava Wiki



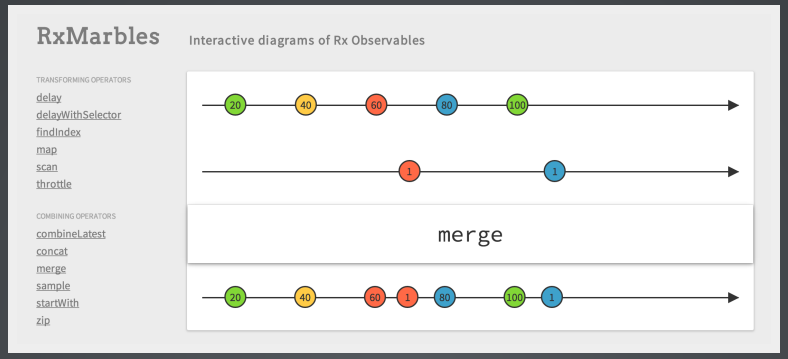
TIME INTERVAL DIAGRAM

Credit: RxJava Wiki



INTERACTIVE DIAGRAMS

Many interactive diagrams can be found at RxMarbles



CIRCLE DEMO

Idea credit: David Sankel's cppnow2014 presentation

Gui: OpenFrameworks for C++11

Code: OfxRx & RxCpp

COMBINE TIME AND MOVEMENT

```
auto orbit_points = orbitPointsFromTimeInPeriod(
    timeInPeriodFromMilliseconds(
        updates.
        milliseconds()));

auto location_points = mouse.
    moves().
    map(pointFromMouse);

location_points.
    combine_latest(std::plus<>(), orbit_points).
    subscribe(
        [=](ofPoint c){
            // update the point that the draw() call will use
            center = c;
        });
```

ORBIT FROM TIME

```
rxcpp::observable<float>
ofxCircle::timeInPeriodFromMilliseconds(
    rxcpp::observable<unsigned long long> timeInMilliseconds){
    return timeInMilliseconds.
        map(
            [this](unsigned long long tick){
                // map the tick into the range 0.0-1.0
                return ofMap(tick % int(orbit_period * 1000),
                    0, int(orbit_period * 1000), 0.0, 1.0);
            });
}

rxcpp::observable<ofPoint>
ofxCircle::orbitPointsFromTimeInPeriod(
    rxcpp::observable<float> timeInPeriod){
    return timeInPeriod.
        map(
            [this](float t){
                // map the time value to a point on a circle
                return ofPoint(orbit_radius * std::cos(t * 2 * 3.14),
                    orbit_radius * std::sin(t * 2 * 3.14));
            });
}
```

RX - REACTIVE EXTENSIONS

ORIGIN

- Rx originated as Rx.Net
- Rx.Net followed LINQ.

LINQ

- Set of algorithms for `IEnumerable`
- Equivalent to range efforts for C++
- Names from SQL syntax

```
// C#

List<string> fruits =
    new List<string> { "apple", "passionfruit", "banana", "mango",
                    "orange", "blueberry", "grape", "strawberry" };

IEnumerable<string> query = fruits.Where(fruit => fruit.Length < 6);

IEnumerable<int> squares =
    Enumerable.Range(1, 10).Select(x => x * x);
```

RX IS LINQ INVERTED

LINQ (Pull)

```
// C#

// GetEnumerator starts an independent in-order traversal of the source
IEnumerator<T> IEnumerable<T>::GetEnumerator();
    void IEnumerator<T>::IDisposable::Dispose(); // cancelation
    bool IEnumerator<T>::MoveNext(); // false for complete
    T    IEnumerator<T>::Current; // throws for error
```

Rx (Push)

```
// C#

// Subscribe inverts GetEnumerator and MoveNext
IDisposable IObservable<T>::Subscribe(Observer<T>);
    void IDisposable::Dispose(); // cancelation
    Observer<T>::OnComplete(); // inverts MoveNext return value
    Observer<T>::OnNext(T); // inverts Current
    Observer<T>::OnError(Exception); // inverts throw
```

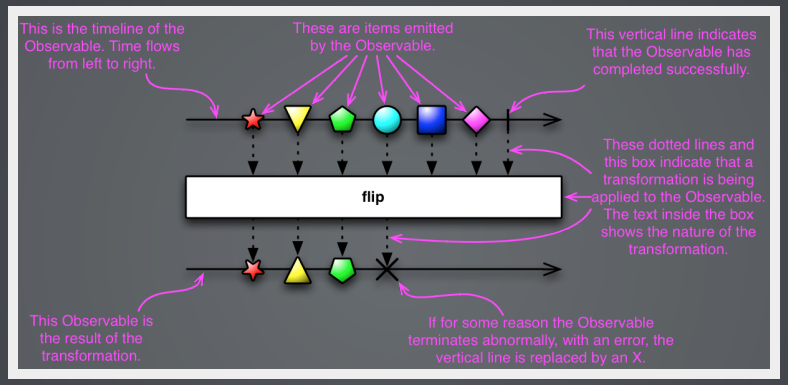
TIME

```
// C#

DateTimeOffset Scheduler::Now; // current time
IDisposable Scheduler::Schedule(DateTimeOffset dueTime, Action action);
    void IDisposable::Dispose(); // cancelation
    void Action::Action();
```

RX CONTRACT

- OnNext, OnError and OnComplete calls are always serialized
- Dispose, OnError and OnComplete terminate
- OnError and OnComplete call Dispose



Credit: RxJava Wiki

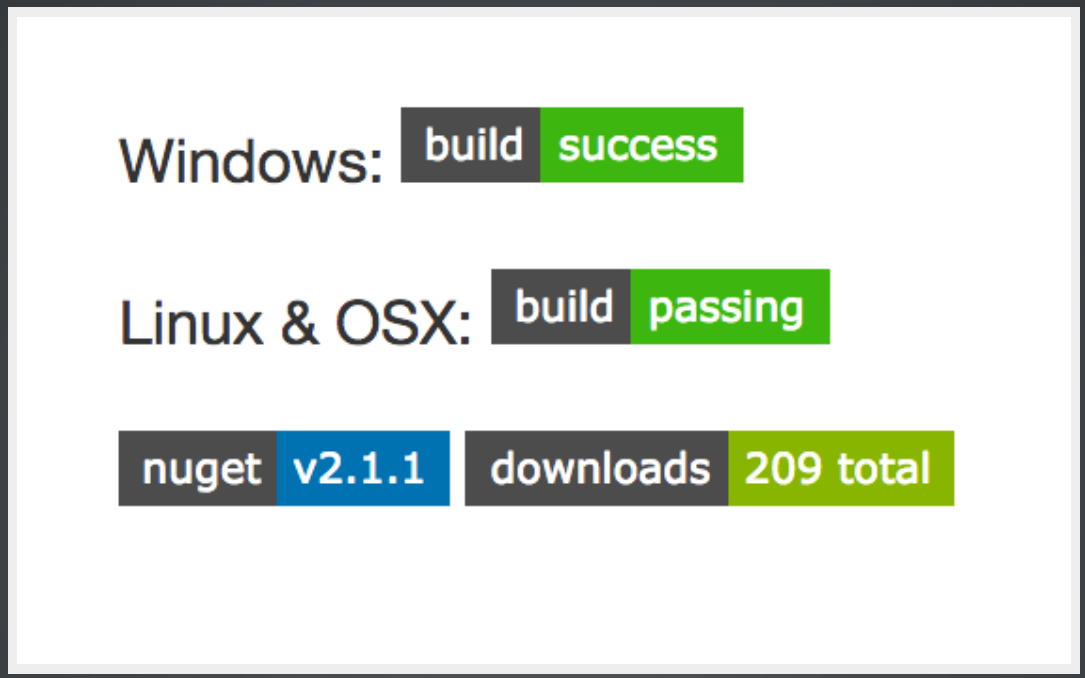
RX AND FRP

Time in Rx is Discrete, not Continuous

RX LEARNING RESOURCES

- LearnRx (RxJS) by JHusain from Netflix
- IntroToRx (Rx.NET) by Lee Campbell
- IntroRx (RxJS) by André Staltz

RXCPP


Windows:
build
success
Linux & OSX:
build
passing
nuget
v2.1.1
downloads
209 total

ORIGIN

- Started as a prototype written by Aaron Lahman
- v2 changes the tradoffs selected by the prototype
- v2 has changes introduced in RxJava

V2 DESIGN GOALS

- header-only
- compile-time polymorphism
- opt-in runtime polymorphism
- opt-in concurrency
- ref classes wrapping all pointers
- compiles with latest gcc, clang and VC compilers
- C++ library is the only dependency

LIFETIME

SUBSCRIPTION

```
// lifetime
bool subscription::is_subscribed();
void subscription::unsubscribe();

// nested lifetimes
weak_subscription composite_subscription::add(Subscription /*void()*/);
void composite_subscription::remove(weak_subscription);
```

FACTORY

OBSERVABLE<>

```
static observable<T> observable<>::create<T>(
    OnSubscribe /*void(subscriber<T> out)*/);

// sources

static observable<T0> observable<>::from(T0, TN...);
static observable<T> observable<>::iterate(Collection<T>);

static observable<T> observable<>::range(
    T first, T last,
    difference_type step);

static observable<long> observable<>::interval(
    rxsc::scheduler::clock_type::time_point initial,
    rxsc::scheduler::clock_type::duration period);

static observable<T> observable<>::never<T>();
static observable<T> observable<>::empty<T>();
static observable<T> observable<>::error<T>(Exception);

// . . .
```

INSTANCE

OBSERVABLE<T>

```
composite_subscription observable<T>::subscribe(
    composite_subscription lifetime,
    OnNext /*void(T)*/,
    OnError /*void(std::exception_ptr)*/,
    OnCompleted /*void()*/);

// operators

observable<T> observable<T>::filter(Predicate /*bool(T)*/);
observable<U> observable<T>::map(Transform /*U(T)*/);

observable<V> observable<T>::flat_map(
    Extract /*observable<U>(T)*/,
    Transform /*V(T, U)*/);

observable<U> observable<T0>::combine_latest(
    Transform /*U(T0, TN...)*/,
    observable<TN>...);

observable<T> observable<T>::merge(observable<T>...);
observable<T> observable<T>::concat(observable<T>...);

// . . .
```

THREAD-SAFETY

COORDINATION

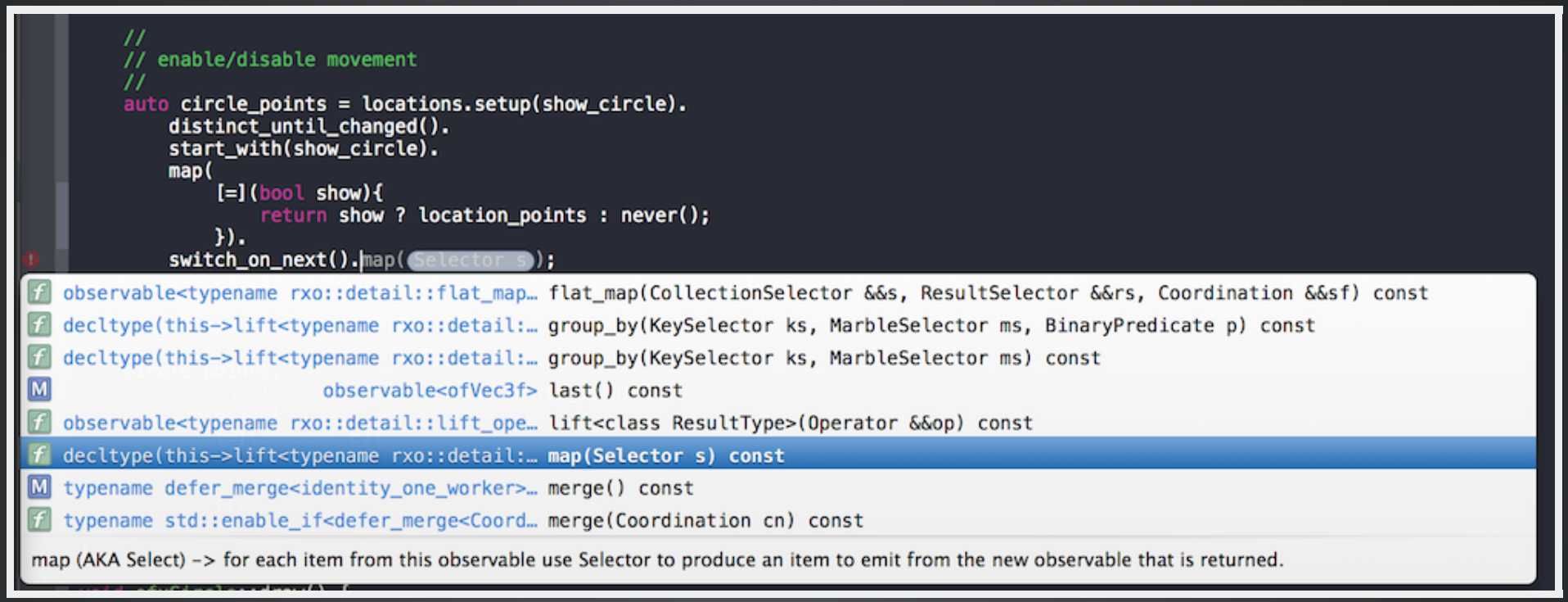
```
// Default - not thread safe
// noop for when all observables are using the same thread
auto noop_immediate = identity_immediate();
auto noop_trampoline = identity_current_thread();

// Opt-in - thread safe

// Uses a mutex to serialize calls from multiple threads
auto serialize_with_pool = serialize_event_loop();
auto serialize_with_new = serialize_new_thread();

// Uses a queue to shift all calls to a new thread
auto observe_on_pool = observe_on_event_loop();
auto observe_on_new = observe_on_new_thread();
```

SUGGESTIONS



TEST DEMO

Code: OfxRx & RxCpp

TIMEINPERIODFROMMILLISECONDS TEST

```
auto sc = rxsc::make_test();
auto w = sc.create_worker();
const rxsc::test::messages<unsigned long long> m_on;
const rxsc::test::messages<float> p_on;

auto xs = sc.make_hot_observable({
    m_on.next(300, 250), m_on.next(400, 500),
    m_on.next(500, 750), m_on.next(600, 1000),
    m_on.completed(700)
});

orbit_offset = 0;
orbit_period = 1.0;

auto res = w.start(
   [&]() {
       return timeInPeriodFromMilliseconds(xs);
   });

auto required = rxu::to_vector({
    p_on.next(300, 0.25), p_on.next(400, 0.5),
    p_on.next(500, 0.75), p_on.next(600, 0.0),
    p_on.completed(700)
});
auto actual = res.get_observer().messages();
```

ORBITPOINTSFROMTIMEINPERIOD TEST

```
const rxsc::test::messages<float> p_on;
const rxsc::test::messages<ofPoint> pt_on;

auto roundedPt = [](ofPoint pt){
    return ofPoint(std::round(pt.x), std::round(pt.y));
};

auto xs = sc.make_hot_observable({
    p_on.next(300, 0.25), p_on.next(400, 0.5),
    p_on.next(500, 0.75), p_on.next(600, 0.0),
    p_on.completed(700)
});

orbit_radius = 50;

auto res = w.start(
    [&]() {
        return orbitPointsFromTimeInPeriod(xs).map(roundedPt);
    });

auto required = rxu::to_vector({
    pt_on.next(300, ofPoint(0,  50)), pt_on.next(400, ofPoint(-50, 0)),
    pt_on.next(500, ofPoint(0, -50)), pt_on.next(600, ofPoint( 50, 0)),
    pt_on.completed(700)
});
auto actual = res.get_observer().messages();
```

UNIT TESTS

Library Credit: philsquared/Catch

CATCH LIBRARY TEST CASE

```
SCENARIO("take 2 - fails", "[take][fails][operators]"){ GIVEN("a source"){
        auto sc = rxsc::make_test();
        auto w = sc.create_worker();
        const rxsc::test::messages on;

        auto xs = sc.make_hot_observable({
            on.next(150, 1), on.next(210, 2), on.next(220, 3),
            on.next(230, 4), on.next(240, 5), on.completed(250)
        });

        WHEN("2 values are taken"){
            auto res = w.start(
                [xs]() {
                    return xs.skip(2).as_dynamic();
                });

            THEN("the output only contains items sent while subscribed"){
                auto required = rxu::to_vector({
                    on.next(210, 2), on.next(220, 3), on.completed(220)
                });
                auto actual = res.get_observer().messages();
                REQUIRE(required == actual);
            }
// . . .
}}}
```

CATCH LIBRARY OUTPUT

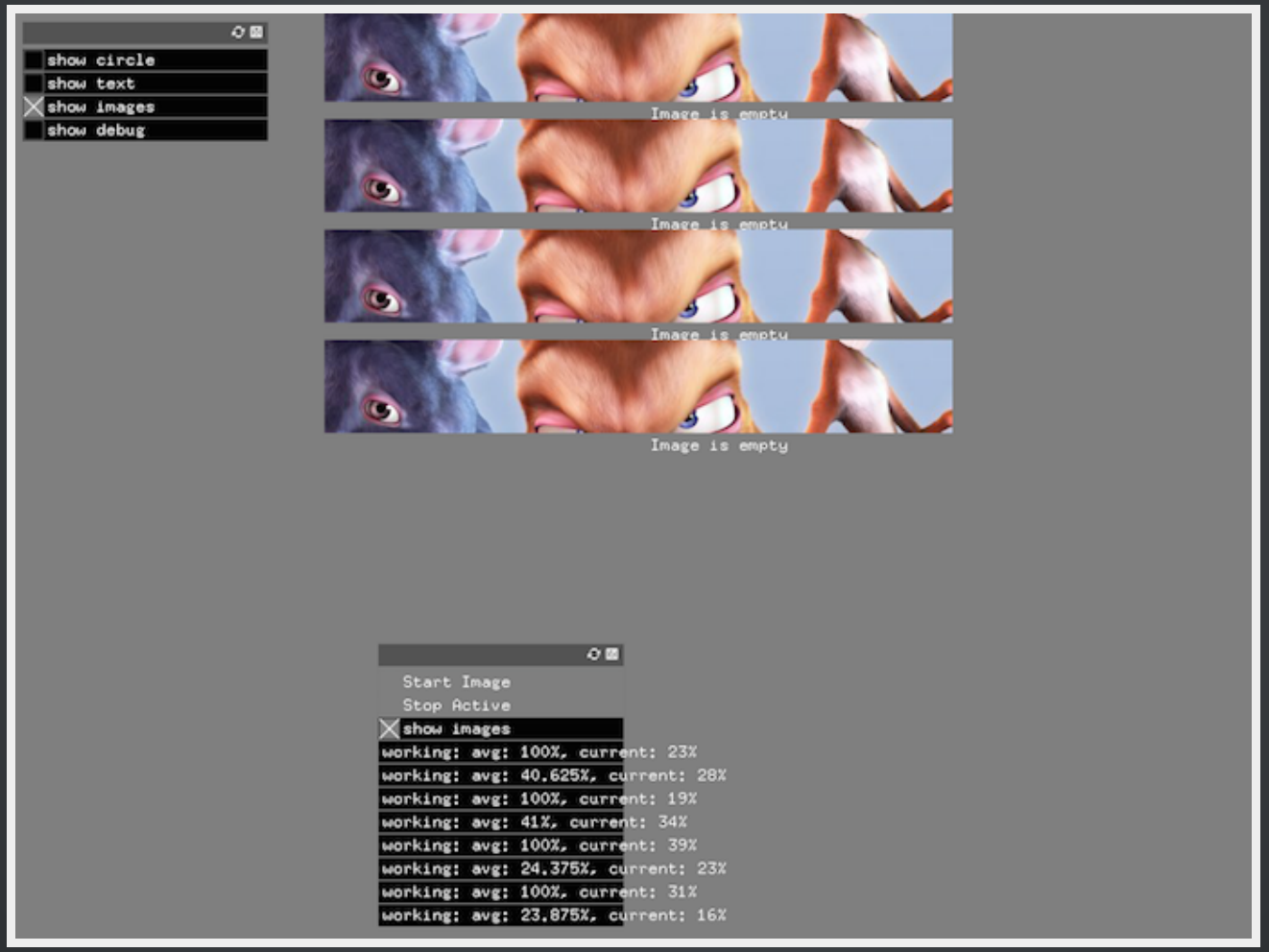
```
-------------------------------------------------------------------------------
Scenario: take 2 - fails
     Given: a source
      When: 2 values are taken
      Then: the output only contains items sent while subscribed
-------------------------------------------------------------------------------
/Users/kirk/source/rxcpp/Rx/v2/examples/tests/take.cpp:57
...............................................................................

/Users/kirk/source/rxcpp/Rx/v2/examples/tests/take.cpp:92: FAILED:
  REQUIRE( required == actual )
with expansion:
  { @210-on_next( 2), @220-on_next( 3), @220-on_completed() }
  ==
  { @230-on_next( 4), @240-on_next( 5), @250-on_completed() }

===============================================================================
```

IMAGE DEMO

Code: OfxRx & RxCpp



THREADED HTTP GET REQUESTS

```
nexts.setup(next).
    map(send_urls).
    merge().
    group_by(
        [=](const next_url& nu) {
            // round-robin requests across multiple threads
            return int(nu.first % thread_count);
        },
        [](const next_url& nu) {return nu;}).
    map(
        [=](const rxcpp::grouped_observable<int, next_url>& urls){

            auto producerthread = rxcpp::observe_on_one_worker(
                rxcpp::observe_on_new_thread().
                    create_coordinator().
                    get_scheduler());

            return http_get_image(
                producerthread,
                urls.get_key(),
                urls,
                halts);
        }).
    merge().
    subscribe();
```

QUEUING OPTIONS

```
rxcpp::observable<http_response_image>
http_get_image(
    rxcpp::observe_on_one_worker producer,
    int key,
    const rxcpp::observable<next_url>& urls,
    const rxcpp::observable<int> stops){

    return urls.
        map(
            [=](const next_url& url){
                return make_http_request(producer, key, url, stops);
            }).
#if 0
        // abort old request and start new request immediately
        switch_on_next().
#else
        // hold on to new requests until the previous have finished.
        concat().
#endif
        map(
            [=](http_response_image progress){
                return update_ui(key, progress);
            });
    }
}
```

CREATE CANCELLABLE REQUEST WITH RETRY

```
rxcpp::observable<http_response_image>
make_http_request(. . .){

    ++queued;
    // ofx tracing hud does not support multiple threads yet
    trace_off();

    return http.get(url.second).
        subscribe_on(producer).
        map(http_progress_image).
        merge().
        observe_on(ofxRx::observe_on_update()).
        lift<http_response_image>(
            [=](rxcpp::subscriber<http_response_image> out){
                return error_display(key, out);
            }).
        finally(
            [=](){
                if (--queued == 0) {trace_on();}
                avg[key] = (progress_labels[key].first + avg[key]) / 2;
            }).
        retry().
        take_until(stops);
}
```

APPEND CHUNKS AND DECODE IMAGE

```
rxcpp::observable<std::shared_ptr<ofPixels>>
http_image(const ofxRx::HttpProgress& hp) {
    return hp.
        body().
        scan(
            std::make_shared<ofBuffer>(),
            [](std::shared_ptr<ofBuffer> acc, ofxRx::BufferRef<char> b){
                acc->append(b.begin(), b.size());
                return acc;
            }).
        last().
        // got all the data, do heavy lifting on the background thread
        map(image_from_buffer);
}
```

SENDING URLS OVER TIME

```
rxcpp::observable<next_url>
send_urls(int) {
    static int count = 0;
    // adds the image url multiple times (20)
    // one url is added every 200 milliseconds
    return rxcpp::observable<>::
        interval(
            ofxRx::observe_on_update().now(),
            std::chrono::milliseconds(200),
            ofxRx::observe_on_update()).
        take(20).
        map(
            [=](long){
                return next_url(
                    count++,
                    "http://. . ./poster_rodents_small.jpg");
            });
}
```

COMBINE THE PROGRESS AND RESULT

```
rxcpp::observable<http_response_image>
http_progress_image(const ofxRx::HttpProgress& hp) {
    return http_progress(hp).
        combine_latest(
            http_image(hp).
                start_with(std::shared_ptr<ofPixels>()));
}
```

PRODUCE ERROR AND PERCENTAGE

```
rxcpp::observable<int>
http_progress(const ofxRx::HttpProgress& hp) {
    return hp.
        response().
        map(http_status_to_error).
        map(
            [](ofx::HTTP::ClientResponseProgressArgs rp){
                return int(rp.getProgress() * 100);
            }).
        distinct_until_changed();
}
```

END

LANGUAGES

- Rx.Net
- RxJs
- RxJava
- ReactiveCocoa, RxSwift
- RxPy
- RxGo
- ...

- Rxcpp

SUBSCRIBER<T>

```
subscriber<T> make_subscriber<T>(
composite_subscription lifetime,
OnNext /*void(T)*/,
OnError /*void(std::exception_ptr)*/,
OnCompleted /*void()*/);

// observer<T>
void subscriber<T>::on_next(T);
void subscriber<T>::on_error(std::exception_ptr);
void subscriber<T>::on_completed();

// composite_subscription
bool subscriber<T>::is_subscribed();
void subscriber<T>::unsubscribe();
weak_subscription subscriber<T>::add(
Subscription /*void()*/);
void subscriber<T>::remove(weak_subscription);
```

Coordination

```
rxsc::scheduler::clock_type::time_point identity_one_worker::now();
coordinator identity_one_worker::create_coordinator(composite_subscription);
```

coordinator<Coordinator>

```
rxsc::scheduler::clock_type::time_point coordinator<Coordinator>::now();
```

SCHEDULER AND WORKER

```
// scheduler
std::chrono::steady_clock scheduler::clock_type;

clock_type::time_point scheduler::now();

worker scheduler::create_worker(
composite_subscription lifetime /*cancel all actions*/);

// worker
clock_type::time_point worker::now();

void worker::schedule(
time_point when,
composite_subscription lifetime, // cancel Action
Action /*void()*/);

void worker::schedule_periodically(
time_point first, duration interval,
composite_subscription lifetime, // cancel Action
Action /*void()*/);
```

RECORDING AND SCHEDULING TIME

REVIEW OF ALGORITHM SETS IN C++

PROMISES

PROMISE BUFFER/QUEUE

SUBSCRIBE TO EACH VALUE

MOUSE CLICKS

- miss clicks when not subscribed
- delivering old clicks when not subscribed

BACKPRESSURE

CANCEL

LIFETIME

async does not block the calling stack

ALGORITHMS

take and skip are the same, while map and filter are different