Adobe

Gold level sponsor

Bloomberg

Engineering

THANK YOU

for playing an instrumental role in
making the NYC++ Meetup possible

THANK YOU

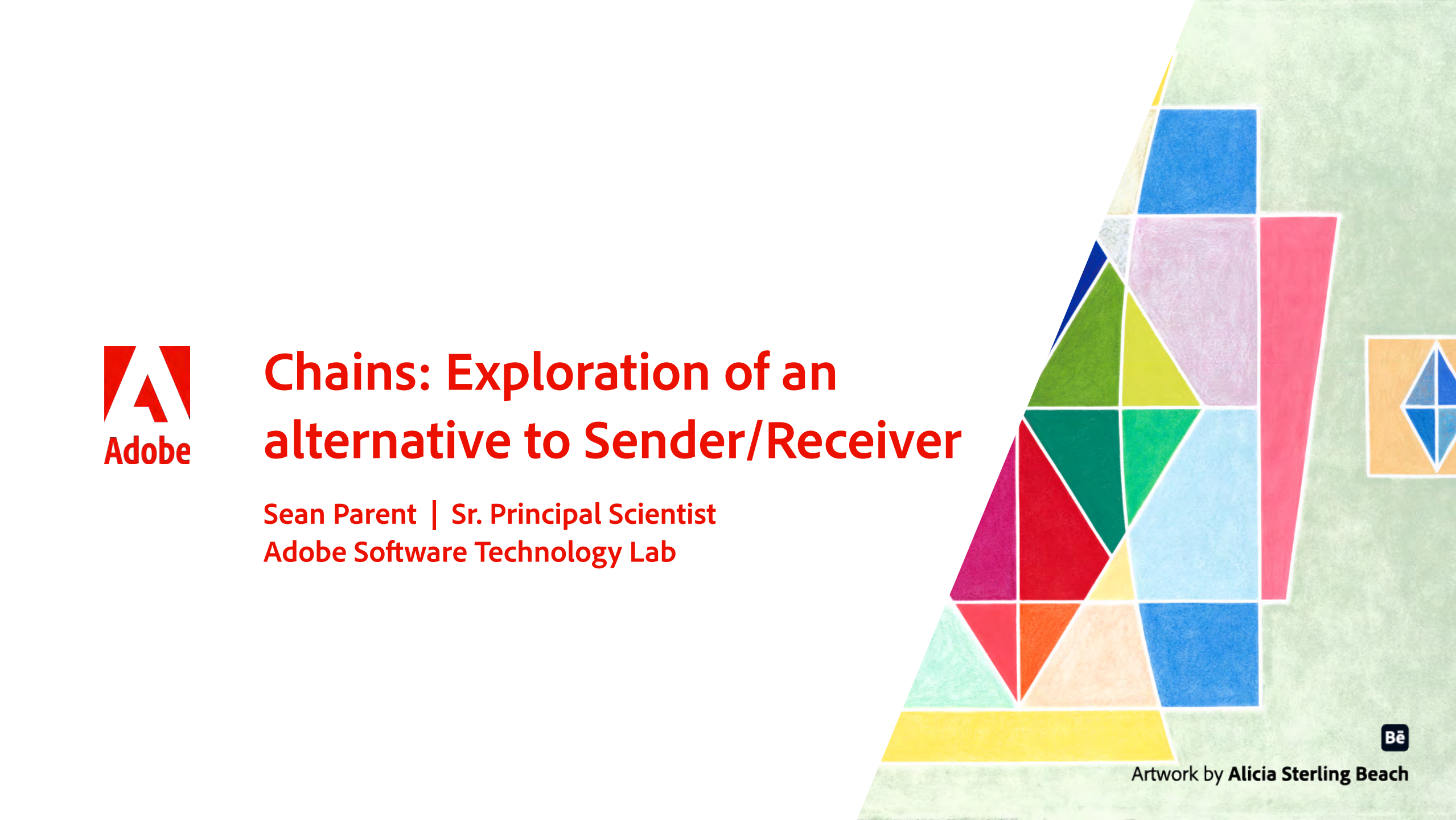
to our sponsors…

Bronze

🙏 **Monochrome Search**

🙏 **Undo**

for supporting the NYC++ Meetup



Chains: Exploration of an alternative to Sender/Receiver

Sean Parent | Sr. Principal Scientist
Adobe Software Technology Lab

Artwork by **Alicia Sterling Beach**

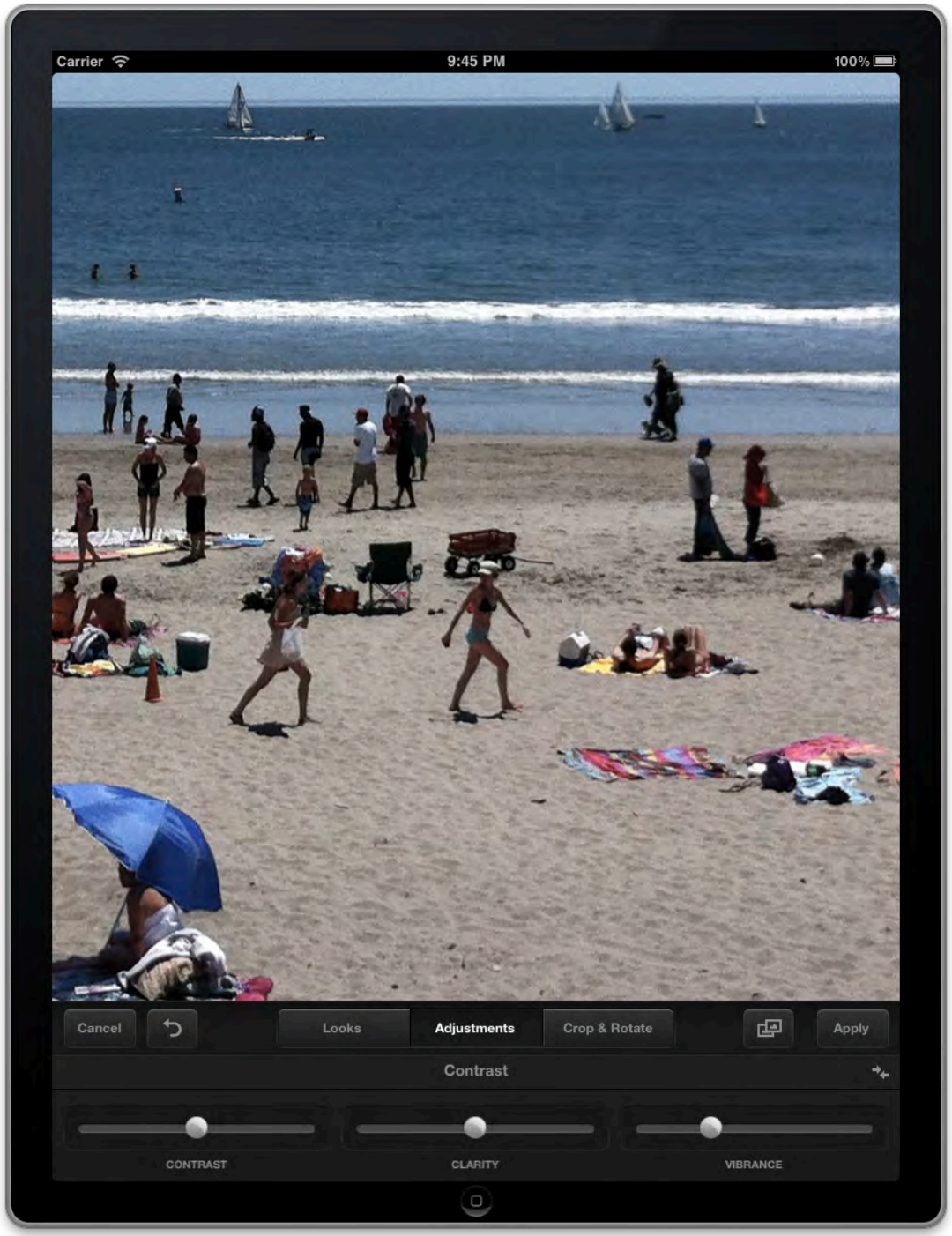
History

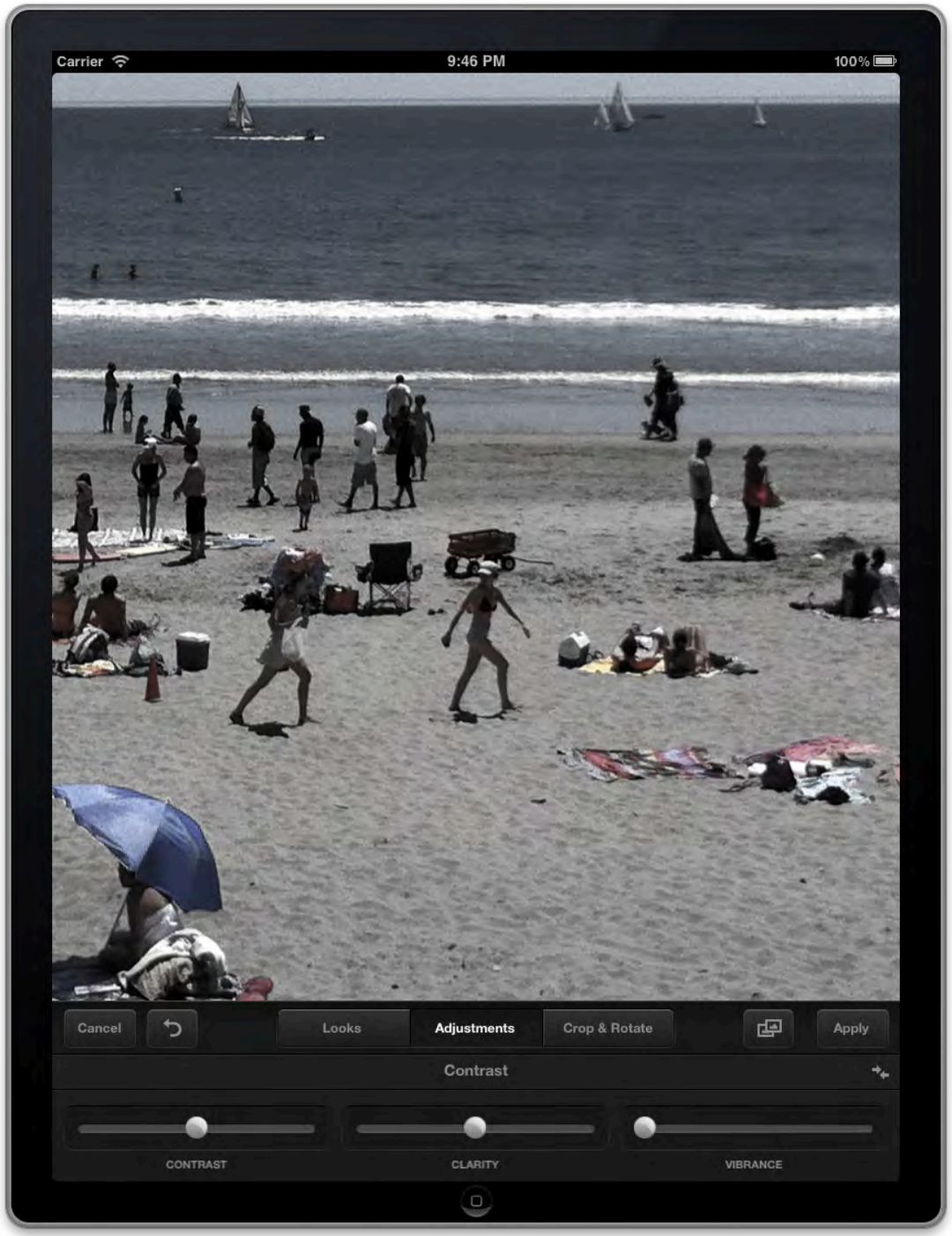
- In 2015, I was involved in bringing Lightroom to the browser
- This was under asm.js - a single-threaded model
- Lightroom is a multi-threaded application
- The code had to be transformed so it could run single-threaded
  - and scale to run efficiently on many cores

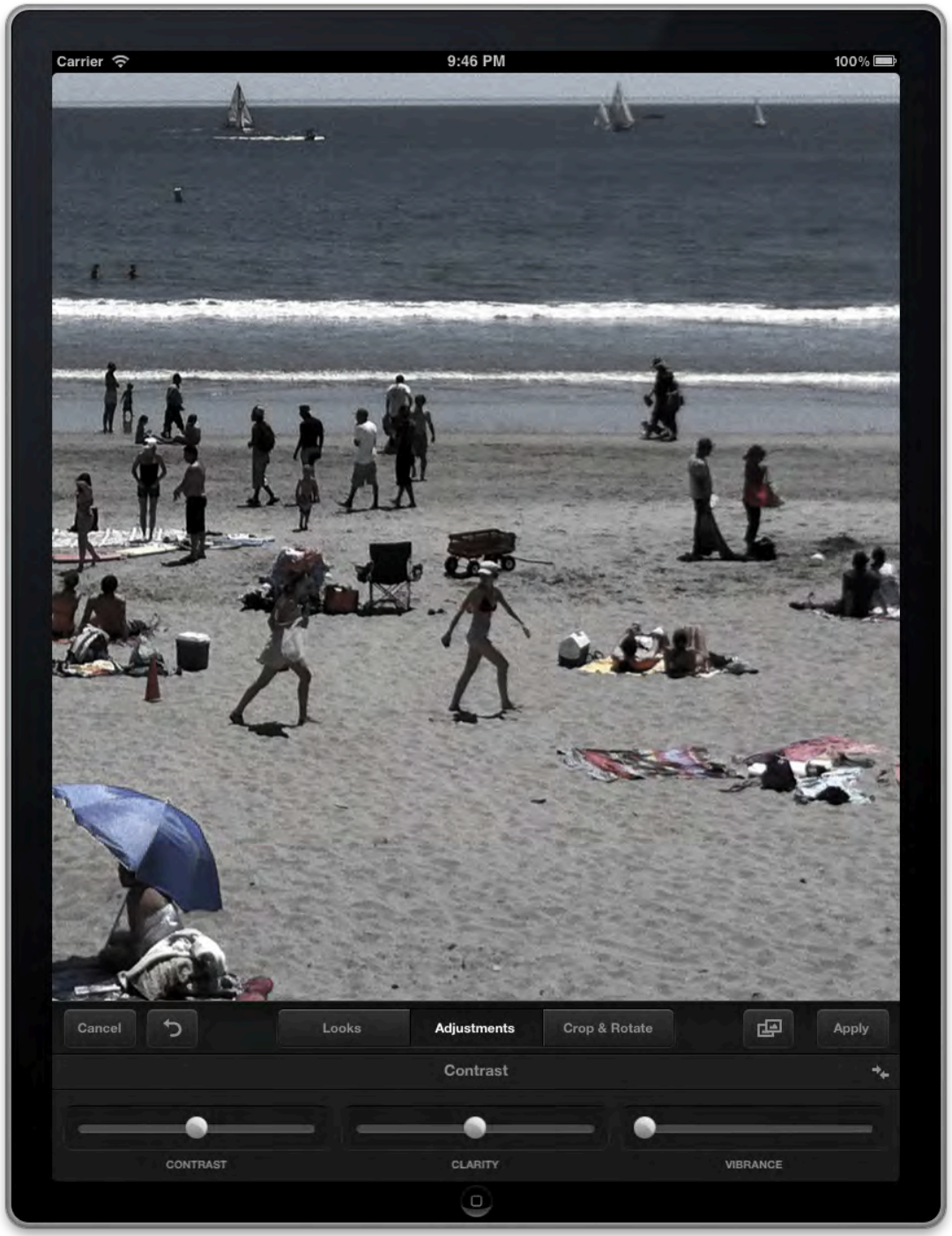
History - Boost futures with continuation

```
auto f = make_ready_future(42);

auto f0 = f.then([](auto a){ return a.get() + 2; });
auto f1 = move(f).then([](auto a){ return a.get() + 3; });

print("{}\n", f0.get());
print("{}\n", f1.get());
```

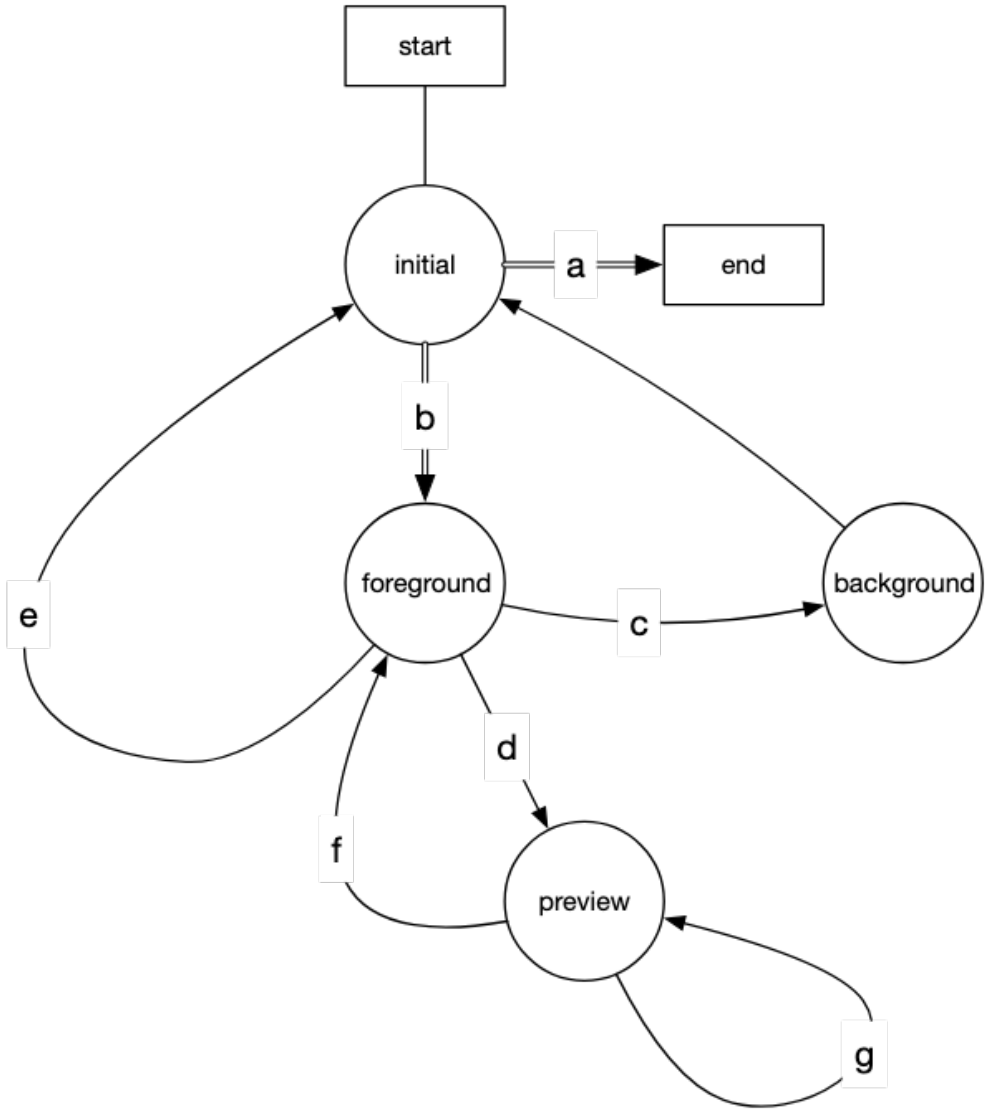
The Problem



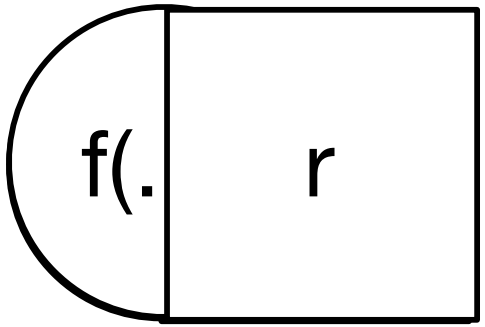




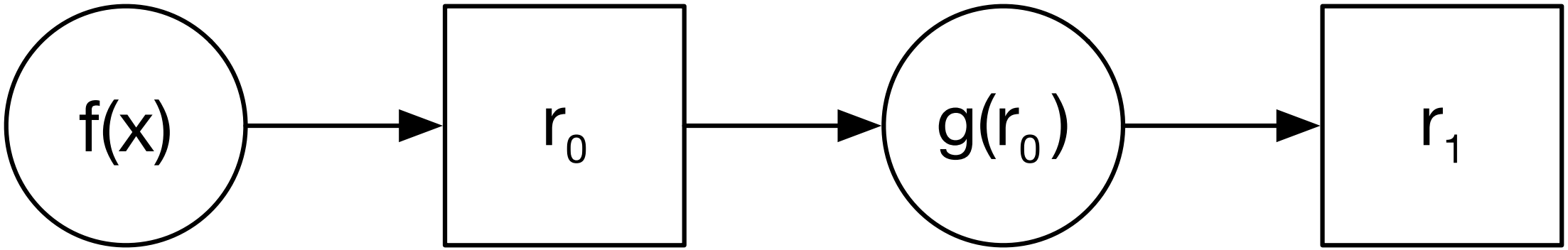
Rendering State Machine Structure



Futures



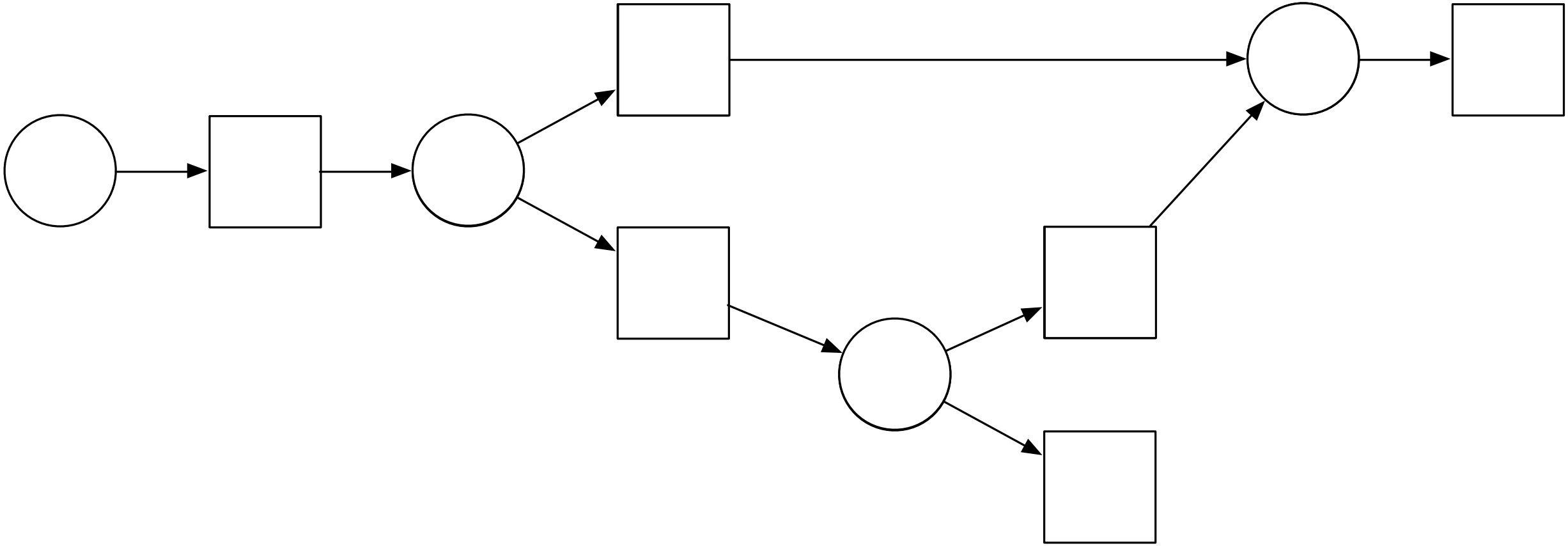
Futures: Continuations




```
g(f(x));
```

```
auto future = async(f, x) | g;
```

Cancelation



Efficient Cancelation

- A dependency graph is a bipartite DAG of operations and their results
- Operations may be scheduled for execution, active (executing), or completed
- If a result is no longer needed (future is destructed), any **uniquely** contributing operations are *canceled*
  - Active operations receive a stop request
  - Scheduled operations will not start, and their resources (which may include results of other operations) are released
- Cancelation is *efficient* if every active uniquely contributing operations immediately receives a stop request and no scheduled operations start after the cancelation.

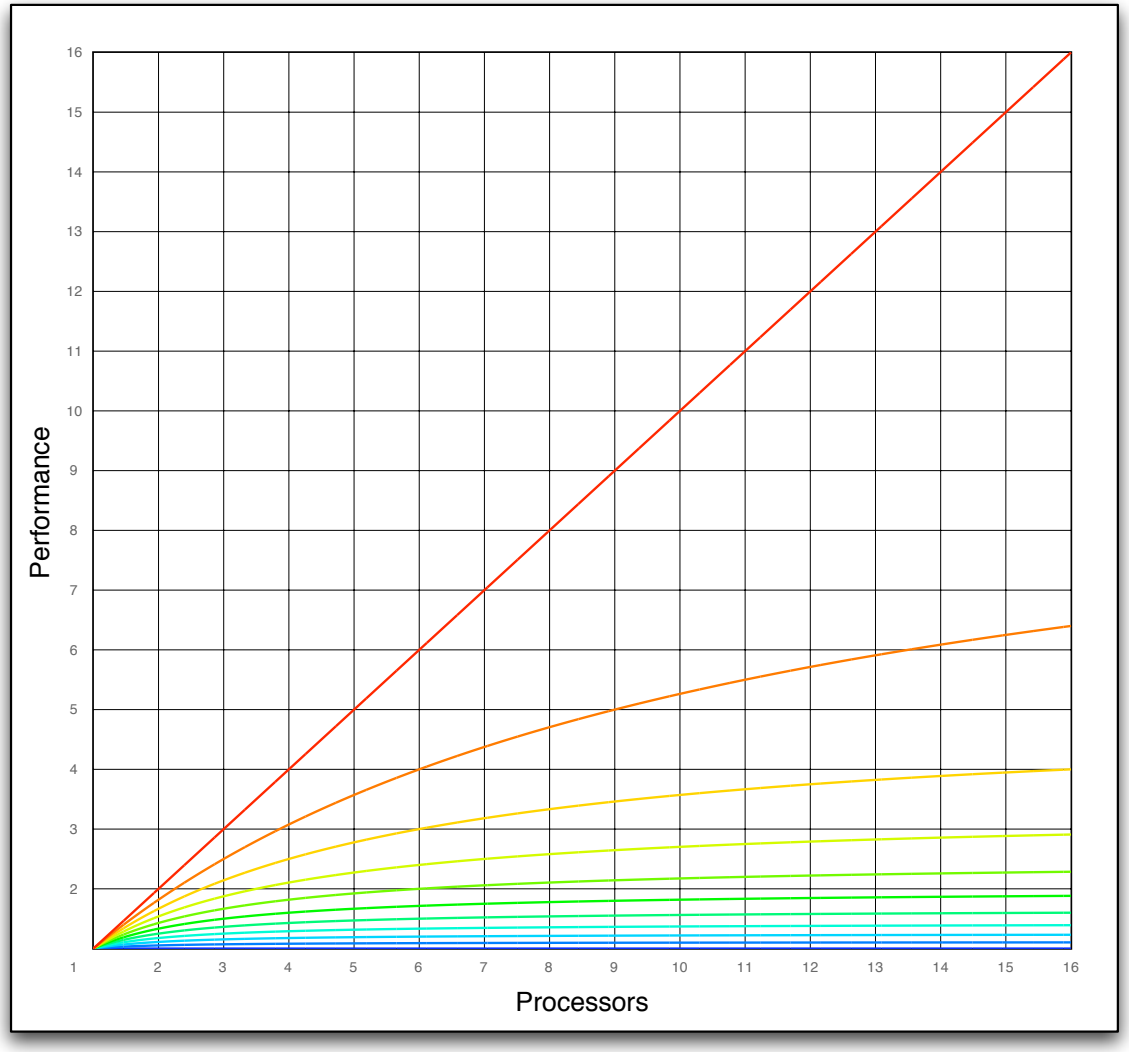
Wait Free

No Waiting

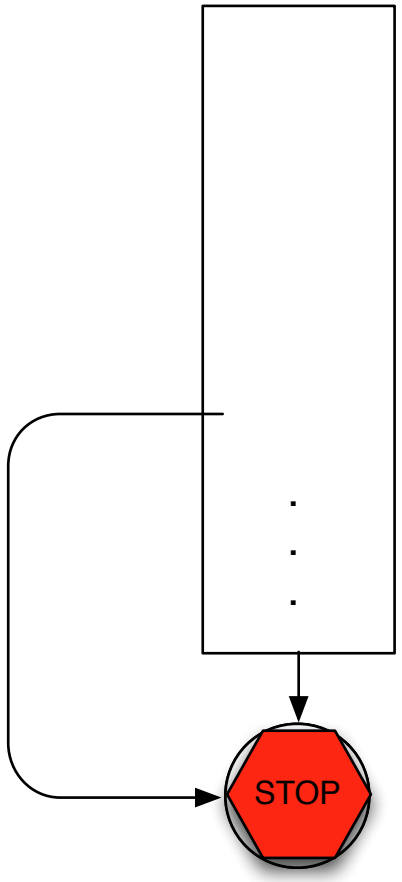
- On a single-threaded system, tasks are queued to the main run loop
- This allows work to be interleaved with responding to events
  - The amount of work done by each task needs to be small enough (a small grain size) to not block user events from coming through and being responded to
- Waiting for a task to complete (a synchronous wait or sync wait) is a deadlock in a single-threaded system
- In a multithreaded system, a sync wait is also *very bad*

Amdahl's Law

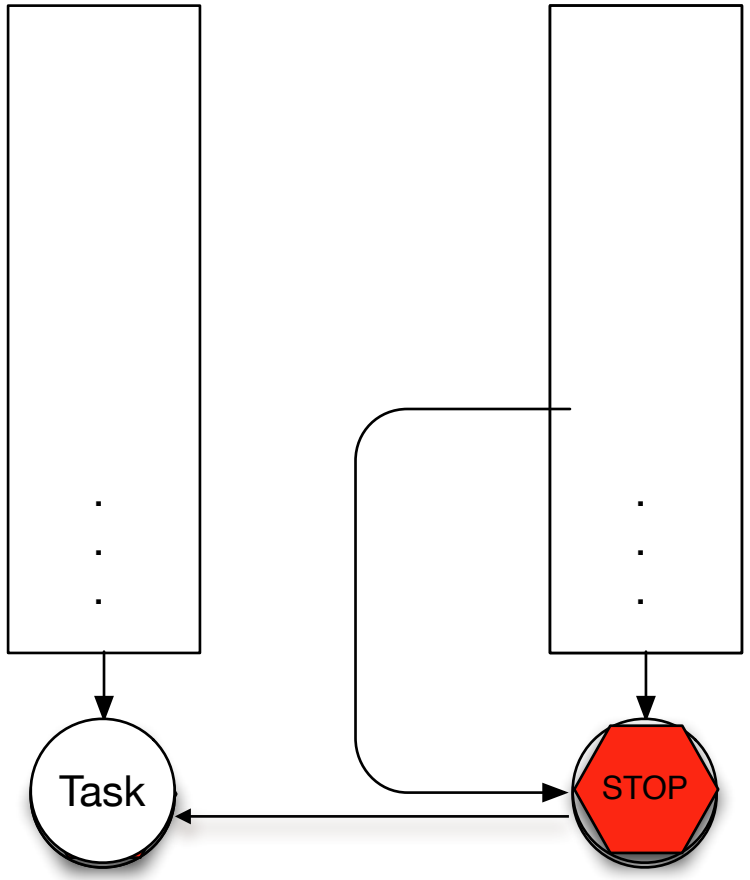
Each line represents 10% more synchronization



Sync Wait Deadlocks



Sync Wait Deadlocks



History

- I gave a talk on how I solved these problems, *Better Code: Concurrency*
- I cleaned up my code and packaged it as a little library on GitHub, stlab/libraries/concurrency
  - Written almost entirely in the hotel bar at C++Now

- Felix Petriconi found it useful and contributed most of the current code
- The library is now in broad use at Adobe and other companies

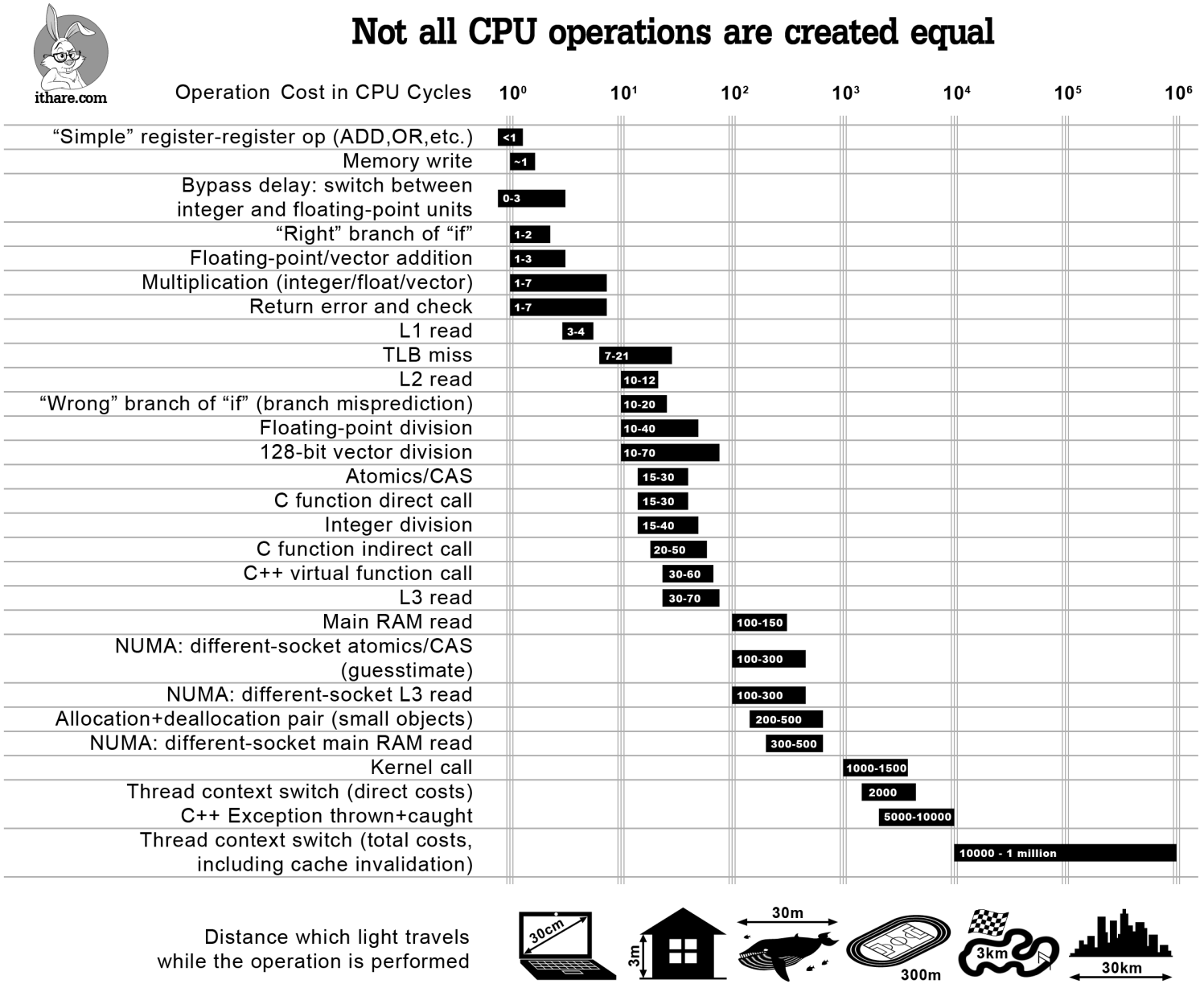
Splits with stlab futures

- stlab futures are copyable and support attaching multiple continuations and efficient cancelation

```
auto a = f | g;
auto b = f | h;
```

The Problem with the Solution

Operation Costs are Not Reflected In Code



Cost of Future Continuations

- Continuations require shared state
  - Memory allocation and deallocation with atomic operations to control lifetime
- Attaching a continuation requires synchronization
- Resolving a continuation requires synchronization
- All of this is under the hood for the simple expression, `f | g`

Don't use continuations as expensive function composition

- If you know the continuation in advance, don't write:

```
auto a = async(f) | g;
```

- But instead write:

```
auto a = async(compose(g, f));
```

Don't use continuations as expensive function composition

- Async operations need to take a callback and compose it with their work

```
auto op() -> future<T> { /*... */ return async(f); }
```

- Becomes:

```
auto op(invocable<T> auto g) -> future<decltype(g(T))> {
    /* ... */
    return async(compose(g, f));
}
```

Sender/Receivers are Function Composition

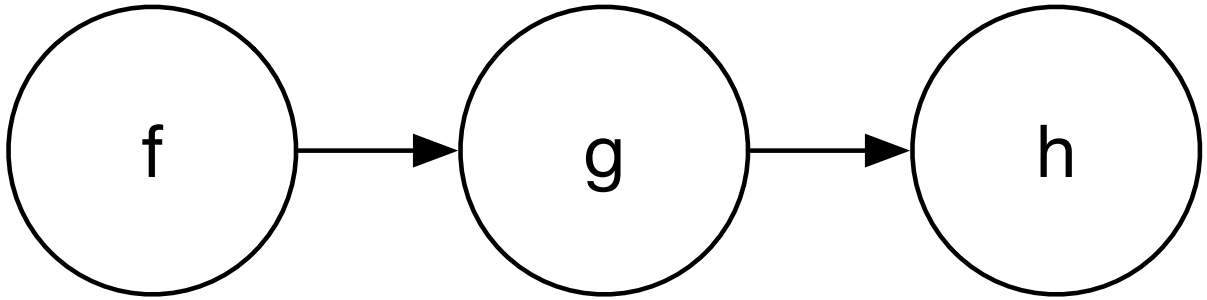
- `then(f) | then(g)`
  - `compose(g, f)`
- `transfer(s) | then(f)`
  - `bind_front(s, f)`
- `transfer(s) | then(f) | then(g)`
  - `bind_front(s, compose(g, f))`

Links

Flat Composition - Links

- `… f | g | h`
- The desired result is, `compose(h, compose(g, f))`
  - But, without the nesting - we don't want to blow the stack

Links



Flat Composition - Exceptions

- Given `f | g | h`, what if an error occurs during the application?
- If we execute `h(g(f(x)))` and f throws an exception, can `g` or `h` catch it?
- The same is true with `std::expected`
  - `f(x).and_then(g).and_then(h)`

Segments

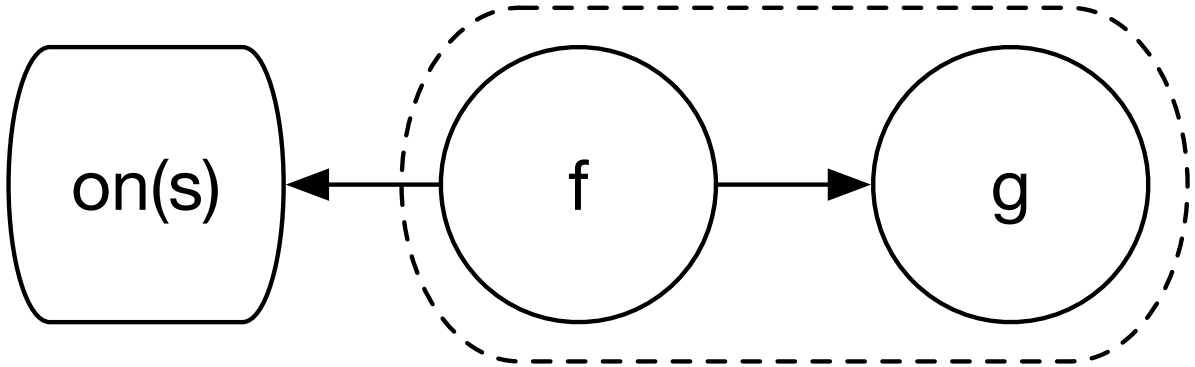
Application - Segments

- `on(s) | f | g`
  - `on()` creates a segment, a function that is the applicator function, `s`, bound to the subsequent sequence of functions
    - Additional functions may still be appended

```
bind_front(s, f | g);
```

  - `s` determines the context in which `f | g` is executed

Segments



Application - Segments

```
auto expector = [](auto f, auto... args) -> expected<...> {
    try { f(args...); }
    catch { return current_exception(); }
};

auto badd = [](int x, int y){
    if (x == 42) throw runtime_error("bad x");
    return x + y;
}

(on(expector) | badd | [](int x){ return x * 2; })(12, 5);

expected{34};
```

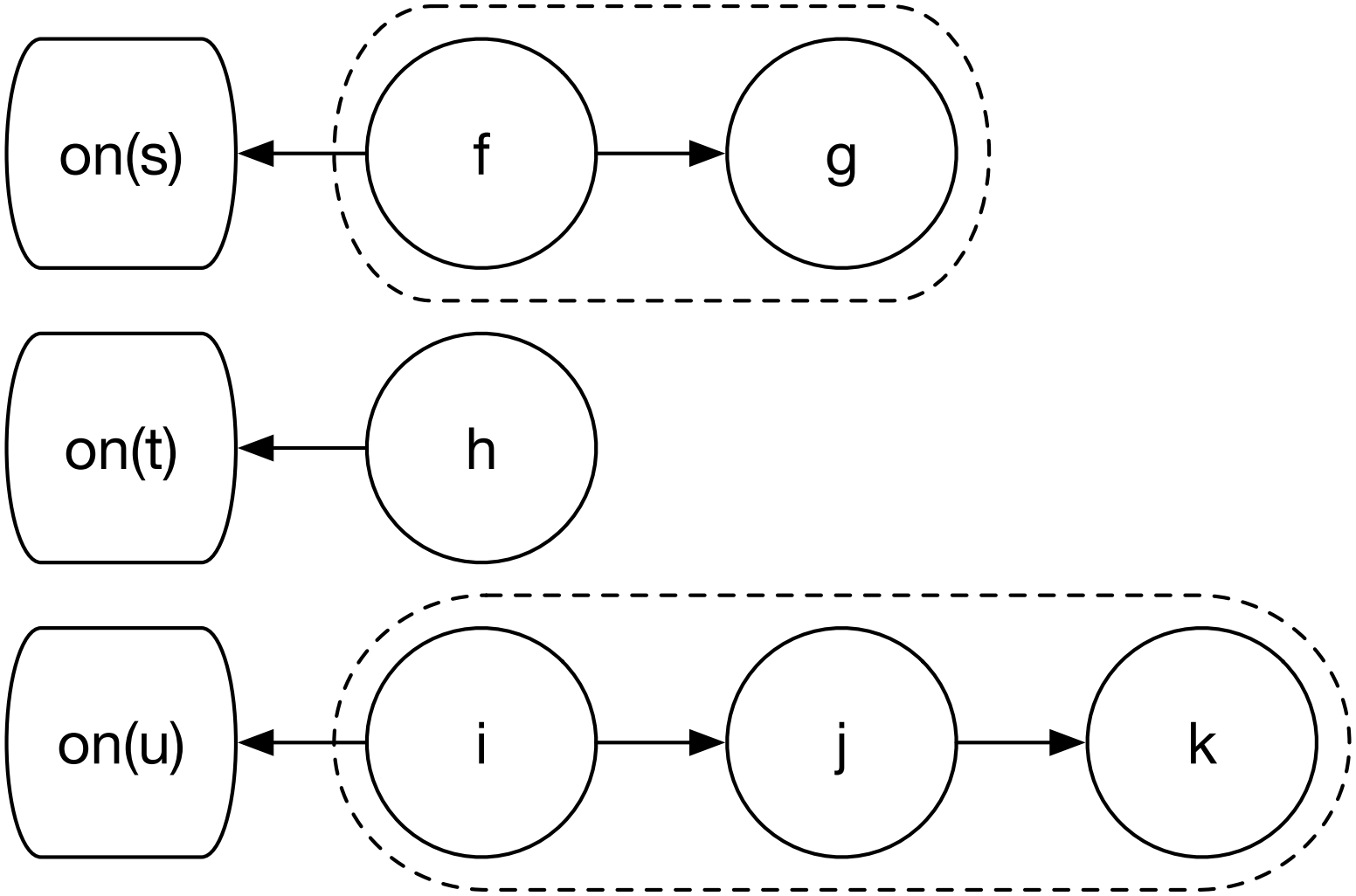
Chains

Chains

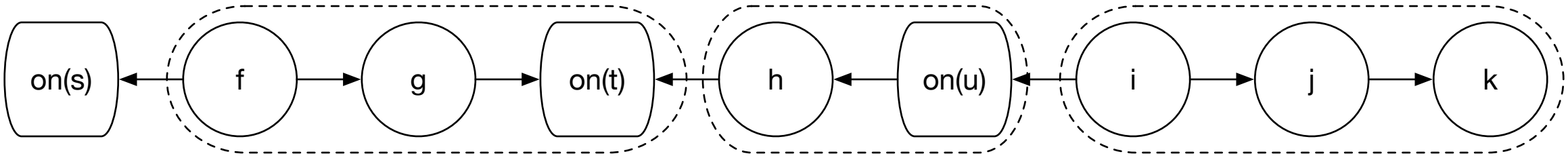
- A chain is simply a sequence of segments
- on(s) | f | g | on(t) | h
- Segments are linked - the next segment is part of the prior segment

```
[ _f = f | g | (on(t) | h), _s = s](auto... args){
    return _s(_f, args...);
}
```

Chains



Chains

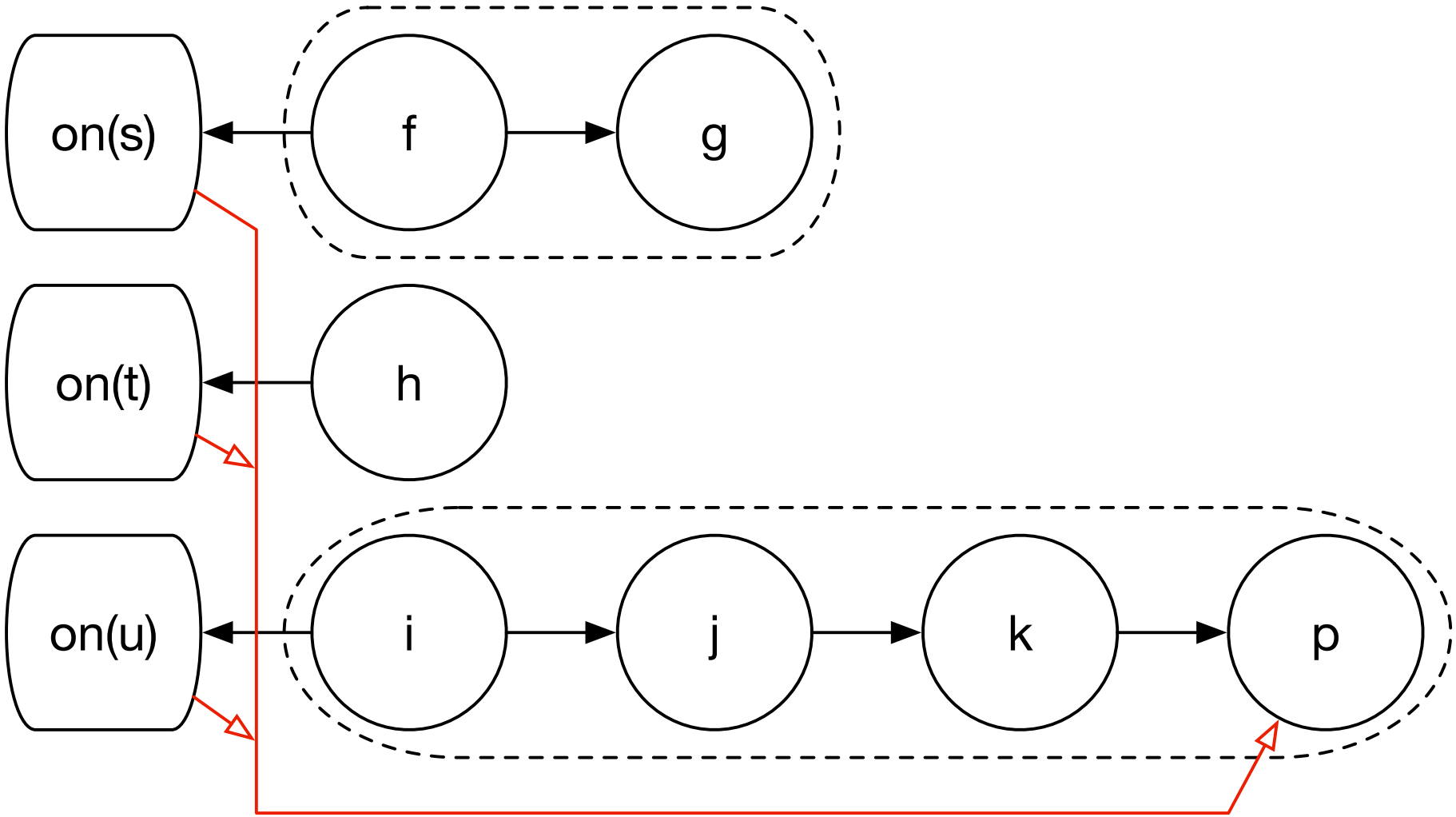


Async?

Mapping to Async

- A chain is a program that describes the sequence of execution
  - We can create segments that provide an execution context to schedule the segment
  - Functions on chains can append a promise and start the chain returning a future
    - And bind additional arguments to every applicator (i.e., a stop token and set_exception operation)

Ready for Async Execution



Mapping to Async

- Chains are a general purpose* facility to build functional descriptions (programs) in C++
- Sender/receivers are a language within which to build asynchronous descriptions**
- So far, I've implemented the equivalents of:
  - `schedule, just, just_error, transfer, then, on, sync_wait`
  - With efficient cancelation and error handling

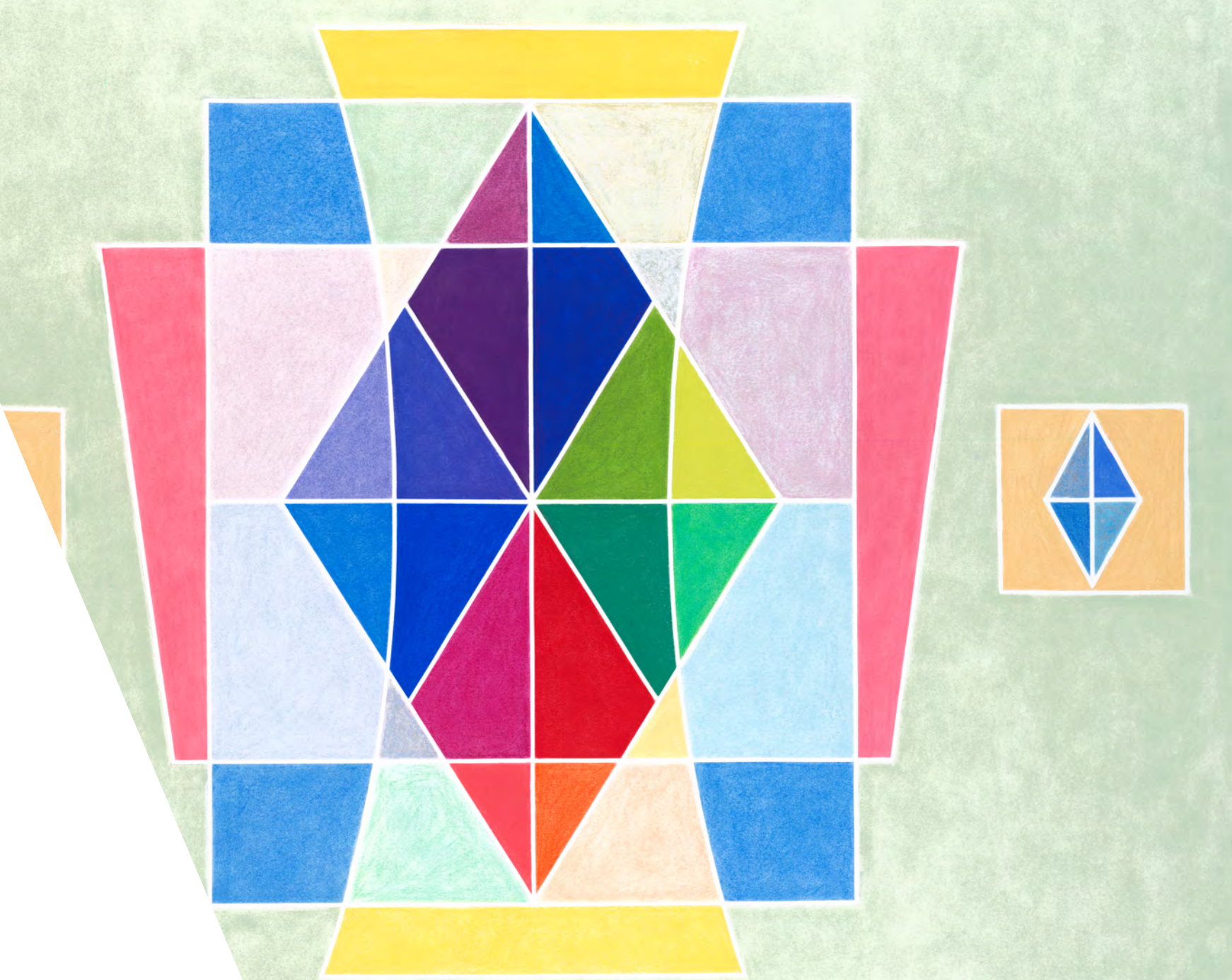
Split

- ???
- We shifted from a model of functions with detached results to one of functions *as* results.
  - What does it mean to split a function?
    - Synchronization is required
    - The shared portion should be executed once
    - Canceling should only cancel the non-shared portion
    - Must have no arguments, first invocation starts, and subsequent invocations ignored

Next Steps

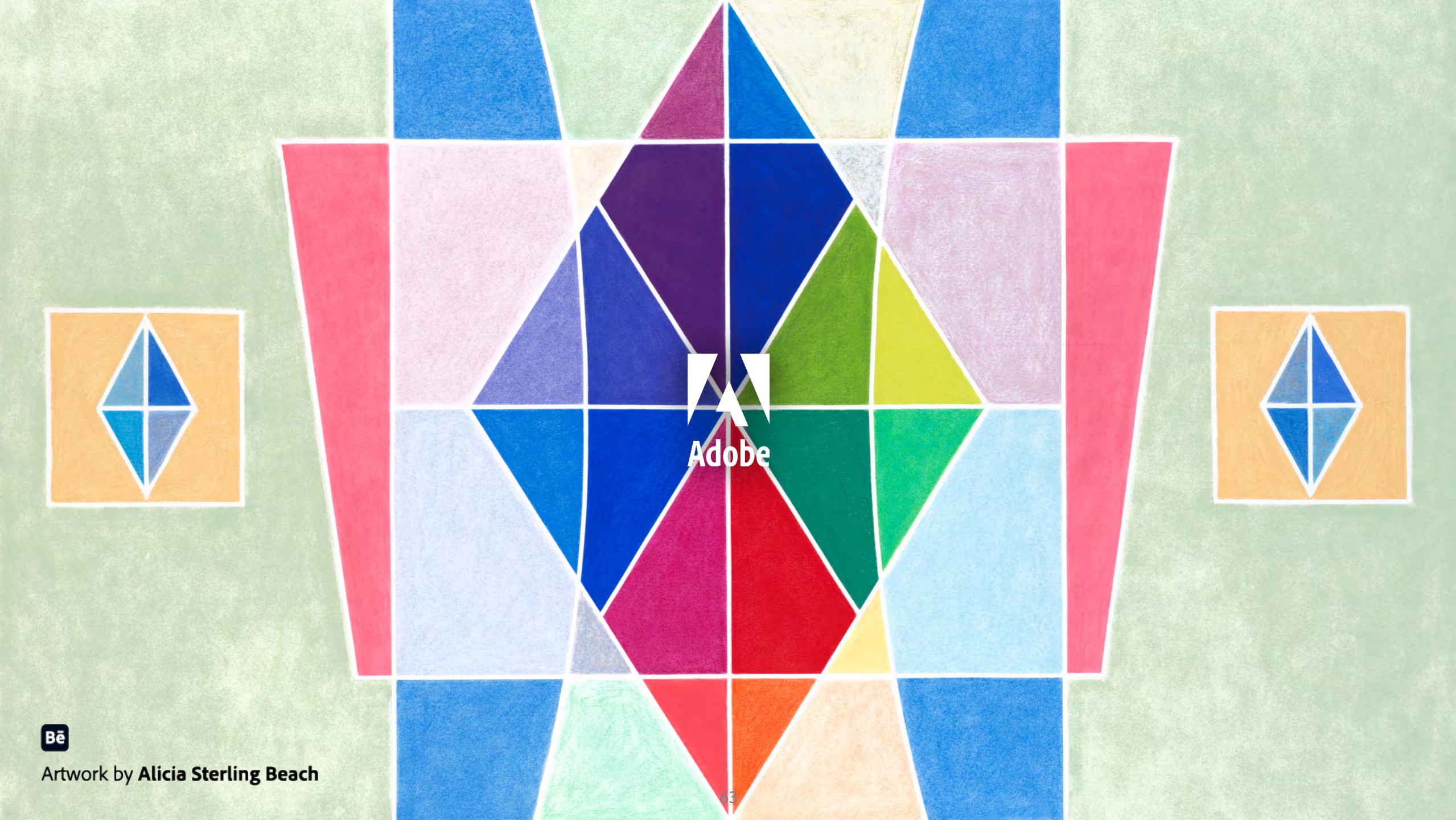
- Chains can't currently do anything that sender/receivers can't do*
  - But the model is simpler
- Chains are a prototype missing production features:
  - void result types everywhere
  - minimizing copies
  - handling perfect forwarding and move-only types everywhere
  - handling copy everywhere appropriate
  - ...

About the artist



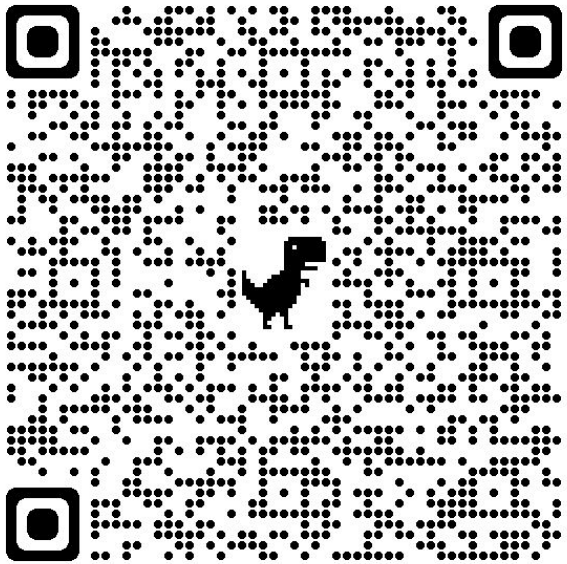
Alicia Sterling Beach

Los Angeles-based artist Alicia Sterling Beach uses watercolors, colored pencils, and soft pastels to bring beauty into the world. Growing up with the vivid colors and music of Latin America, as well as the Native cultures of the American Southwest, her artwork is informed by her history and inspired by nature, light, and classical music. Beach's work is featured on *artlifting.com*, a platform for artists impacted by housing insecurity and disabilities. In this piece, she combines symmetry and joyful colors to express balance, harmony, and spiritual attainment.



Artwork by **Alicia Sterling Beach**

Please consider supporting NYC++



Adobe

https://nycpp.dev/support