Introductory RxJava

First Step to Reactive Concept in Java World

Intae Kim

greatkit@gmail.com

Agenda

- What is RxJava?
- Prerequisite to Understand RxJava
- RxJava Concepts
- RxJava
- Express with RxJava
- Is it essential?

AD

X

O'REILLY

안드로이드 활용 사례 포함

RxJava를 활용한 리액티브 프로그래밍

RxJava의 개념과 사용법, 실무 활용까지

프로그래밍인사이트

토마스 누르키비치, 벤 크리스텐센 지음
김인태 옮김

What is RxJava?

ReactiveX is **a library**
for composing **asynchronous**
and **event-based** programming
by using **observable sequences**

http://reactivex.io/intro.html

Warm up first

Prerequisite to understand RxJava

- Asynchrony
- Java 8 lambda
- Higher-order function

Asynchrony:
Java thread at a glance

```
// in this slide, whenever you see a log method like:
log("Hello, world!")

// You will see the following pattern:
SS.sss [threadName] Hello, world!
```

Really simple thread example

```java
public class FalseThreadRunner {
    public static void main(String ... args) {
        Runnable runnable = () -> {
            log("Hello, thread world!");
            sleep(1000L);
            log("Thread done!");
        };

        log("Hello, main() world!");
        new Thread(runnable).start();
        log("main() end!!!");

    }

    // static void sleep(), static void log() ...
}
```

```
22.839 [main] Hello, main() world!
22.860 [main] main() end!!!
22.860 [Th-0] Hello, thread world!
23.862 [Th-0] Thread done!

Process finished with exit code 0
```

Similar code on JUnit test

```java
Runnable runnable = () -> {
    log("Hello, thread world");
    sleep(1000L);
    log("Thread done");
};

@Test public void threadRunner() {
    log("Hello, main() world");
    new Thread(runnable).start();
    log("End");
}
```

```
26.014 [main] Hello, main() world
26.016 [main] Done
26.016 [Th-0] Hello, thread world
27.??? [Th-0] Thread done

Process finished with exit code 0
```

Another example: make a new file

```
public class FalseThreadTest implements PrimevalLogger {
   Runnable fileTouch = () -> {
        try {
            Thread.sleep(100L);
            File f = new File("/tmp/rxjava.txt");
            f.createNewFile();
        } catch (Exception e) {}
    };

    @Test public void createFile() throws Exception {
        log("create file");
        new Thread(fileTouch).start();
        log("End");
    }
}
```

File NOT created!!!

How to guarantee your thread run properly?

#1: naive way

```
Runnable runnable = () -> {
    sleep(1000L);
    log("Hello, Thread world");
};

@Test public void threadRunner() {
    log("Hello, main() world");
    new Thread(runnable).start();
    log(“End");
    sleep(2000L);
    log(“Done");
}
```

```
14.970 [main] Hello, main() world
14.978 [main] End
15.981 [Th-0] Hello, Thread world
16.979 [main] Done

Process finished with exit code 0
```

#2: better way

```java
Runnable runnable = () -> {
    sleep(1000L);
    log("Hello, Thread world");
};

@Test public void threadRunner() {
    log("Hello, main() world");
    Thread t = new Thread(runnable)
    t.start();
    log(“End");
    t.join();
    log(“Done");
}
```

```
23.030 [main] Hello, main() world
23.034 [main] End
24.038 [Th-0] Hello, Thread world
24.039 [main] Done

Process finished with exit code 0
```

Nice. But, it’s “blocking”.

thread controlling methods

- `setDaemon(boolean)`
- `join()`

#3: Best way

Notify. Do NOT block.

(But it’s too complex)

synchronized

Two task methods:

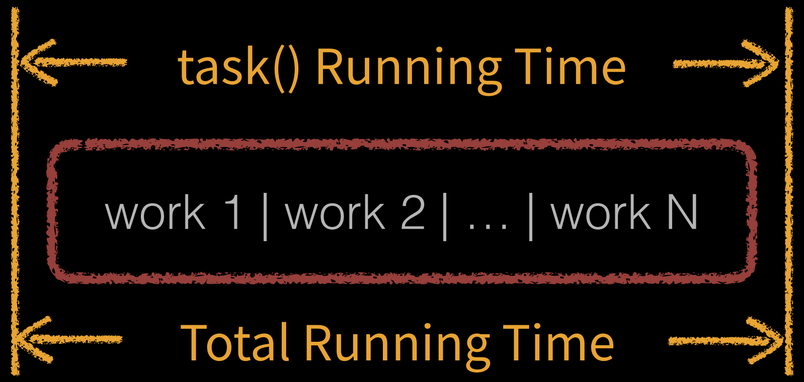
```
void task() {
    // Work 1
    // Work 2
    // ...
    // Work N
}
```

```
synchronized void task() {
    // Work 1
    // Work 2
    // ...
    // Work N
}
```

Ordinary execution:

```
void task(int num) {
    log("begin task " + num);
    sleep(1000L); // Work 1, 2,... N
    log("end task " + num);
}

...

log("START");
task(1);
log("END");
log("DONE");
```

```
49.470 [main] START
49.475 [main] begin task 1
50.477 [main] end task 1
50.477 [main] END
50.477 [main] DONE
```

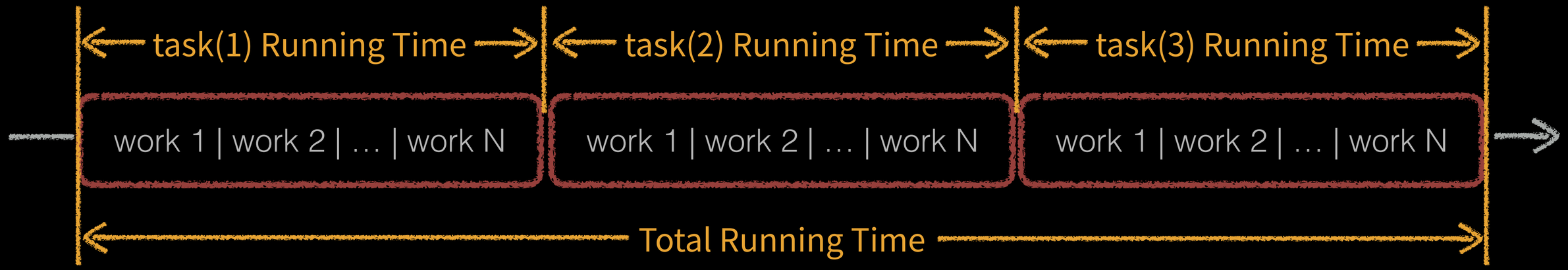

task() Running Time
work 1 | work 2 | … | work N
Total Running Time

Execute N(=3) times:

```
void task(int num) {
    log("begin task " + num);
    sleep(1000L);  // Work 1, 2,... N
    log("end task " + num);
}

...

log("START");
task(1);
task(2);
task(3);
log("END");
log("DONE");
```

```
10.327 [main] START

10.333 [main] begin task 1
11.339 [main] end task 1

11.340 [main] begin task 2
12.346 [main] end task 2

12.346 [main] begin task 3
13.350 [main] end task 3

13.351 [main] END
13.351 [main] DONE
```

Total Running Time ≈ Σ task() Running Time



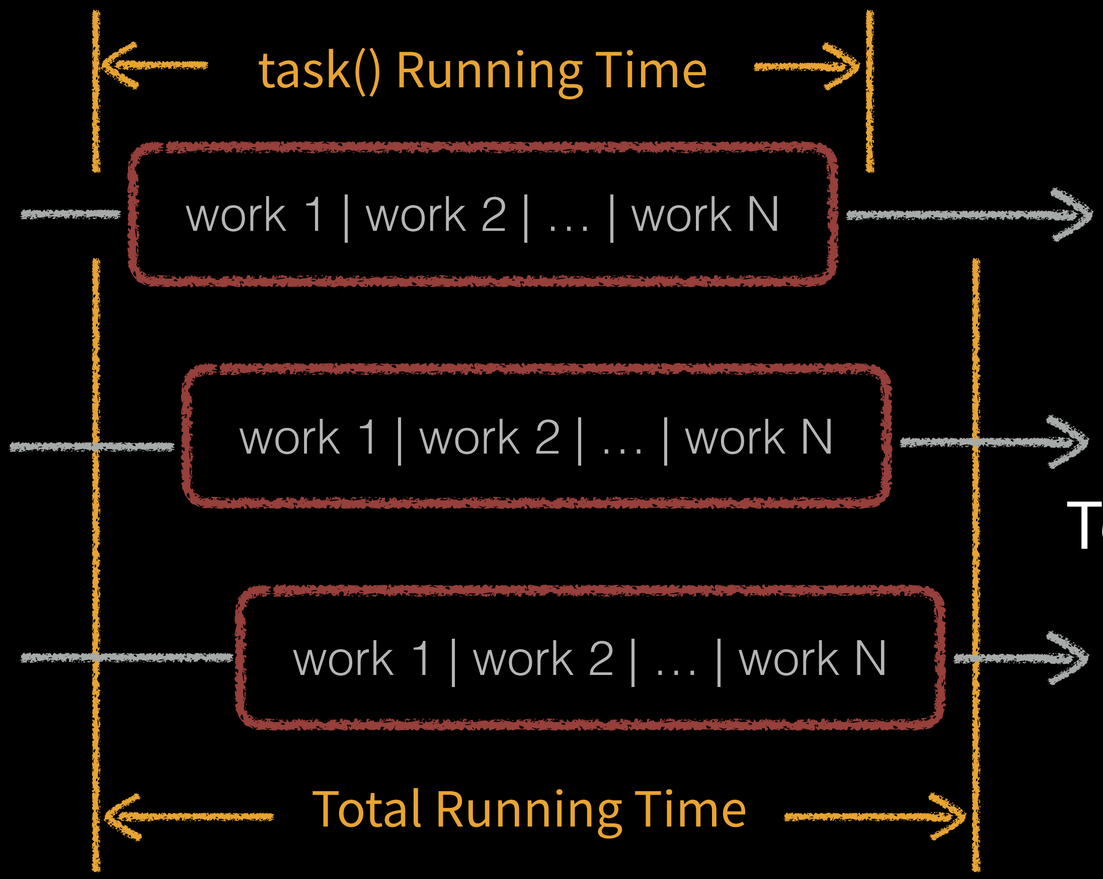
Now, tasks on thread:

```
class Task implements Runnable {
    private final int id;

    Task(int id) {
        this.id = id;
    }

    public void run() {
        task(id);
    }
}
```

plain method (= without “synchronized” keyword)

```
void task(int num) {
    log("begin task " + num);
    sleep(1000L);  // Work 1, 2,... N
    log("end task " + num);
}

...

log(“START");
Thread t1, t2, t3;
t1 = new Thread(new Task(1)); t1.start();
t2 = new Thread(new Task(2)); t2.start();
t3 = new Thread(new Task(3)); t3.start();
log("END");
t1.join(); t2.join(); t3.join();
log("DONE");
```

```
51.055 [main] START
51.067 [main] END

51.067 [Th-2] begin task 3
51.067 [Th-0] begin task 1
51.067 [Th-1] begin task 2

52.071 [Th-2] end task 3
52.071 [Th-0] end task 1
52.071 [Th-1] end task 2

52.072 [main] DONE
```

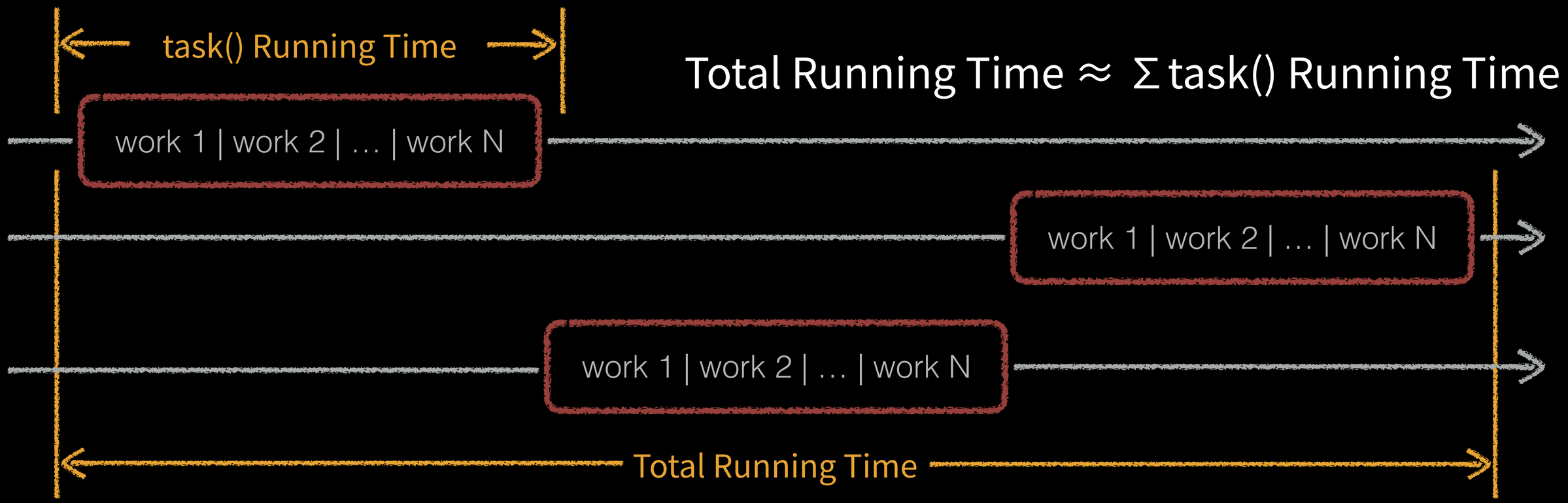


Total Running Time ≈ task() Running Time

synchronized method

```
synchronized void task(int num) {
    log("begin task " + num);
    sleep(1000L);  // Work 1, 2,... N
    log("end task " + num);
}

...

log("START");
Thread t1, t2, t3;
t1 = new Thread(new Task(1)); t1.start();
t2 = new Thread(new Task(2)); t2.start();
t3 = new Thread(new Task(3)); t3.start();
log("END");
t1.join(); t2.join(); t3.join();
log("DONE");
```

```
19.134 [main] START

19.143 [Th-0] begin task 1

19.144 [main] END

20.150 [Th-0] end task 1

20.150 [Th-2] begin task 3
21.154 [Th-2] end task 3

21.155 [Th-1] begin task 2
22.155 [Th-1] end task 2

22.156 [main] DONE
```

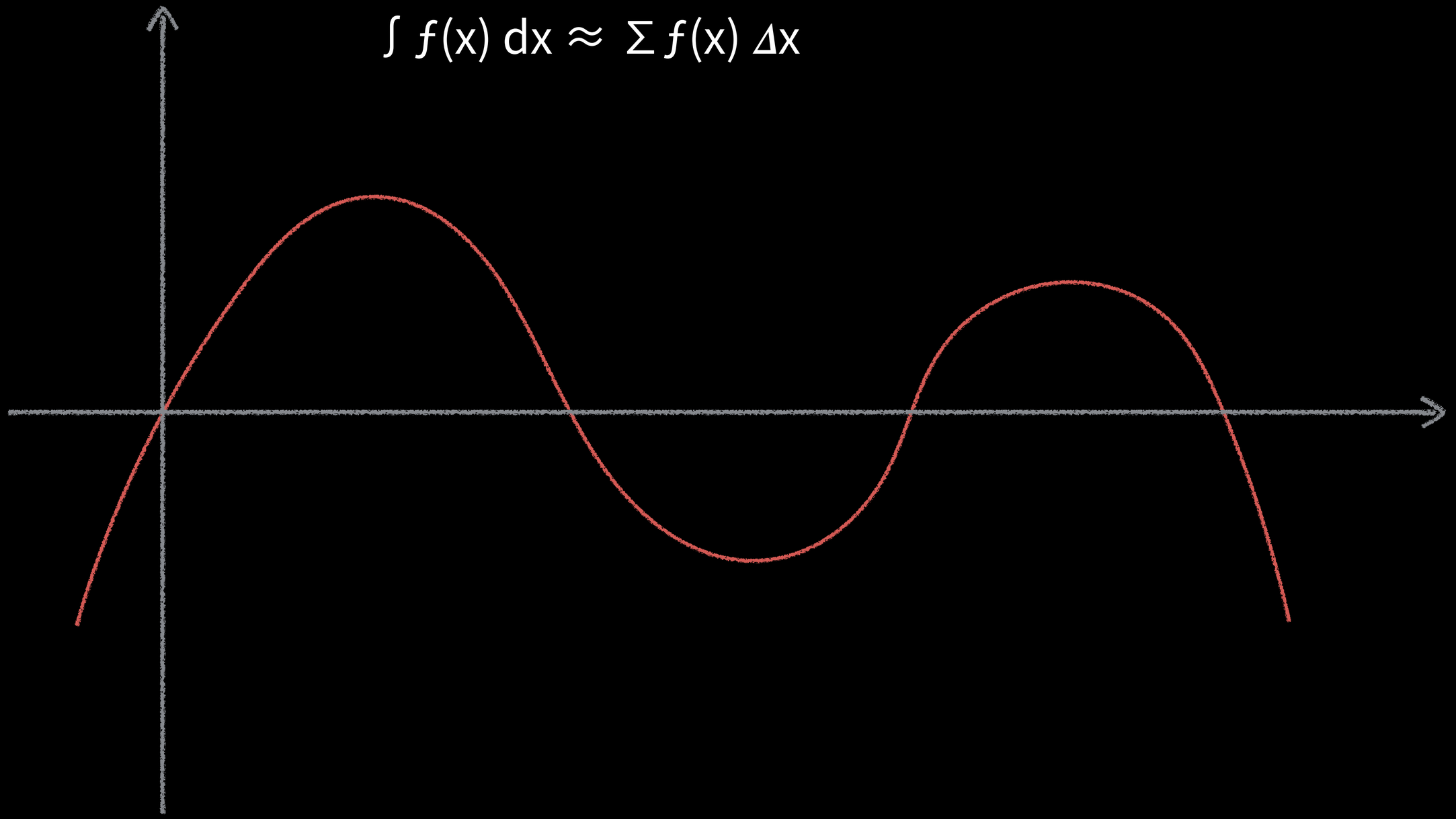

task() Running Time
Total Running Time ≈ Σ task() Running Time
work 1 | work 2 | … | work N
work 1 | work 2 | … | work N
work 1 | work 2 | … | work N
Total Running Time

Seems NO benefit…

Then, why use “synchronized”?

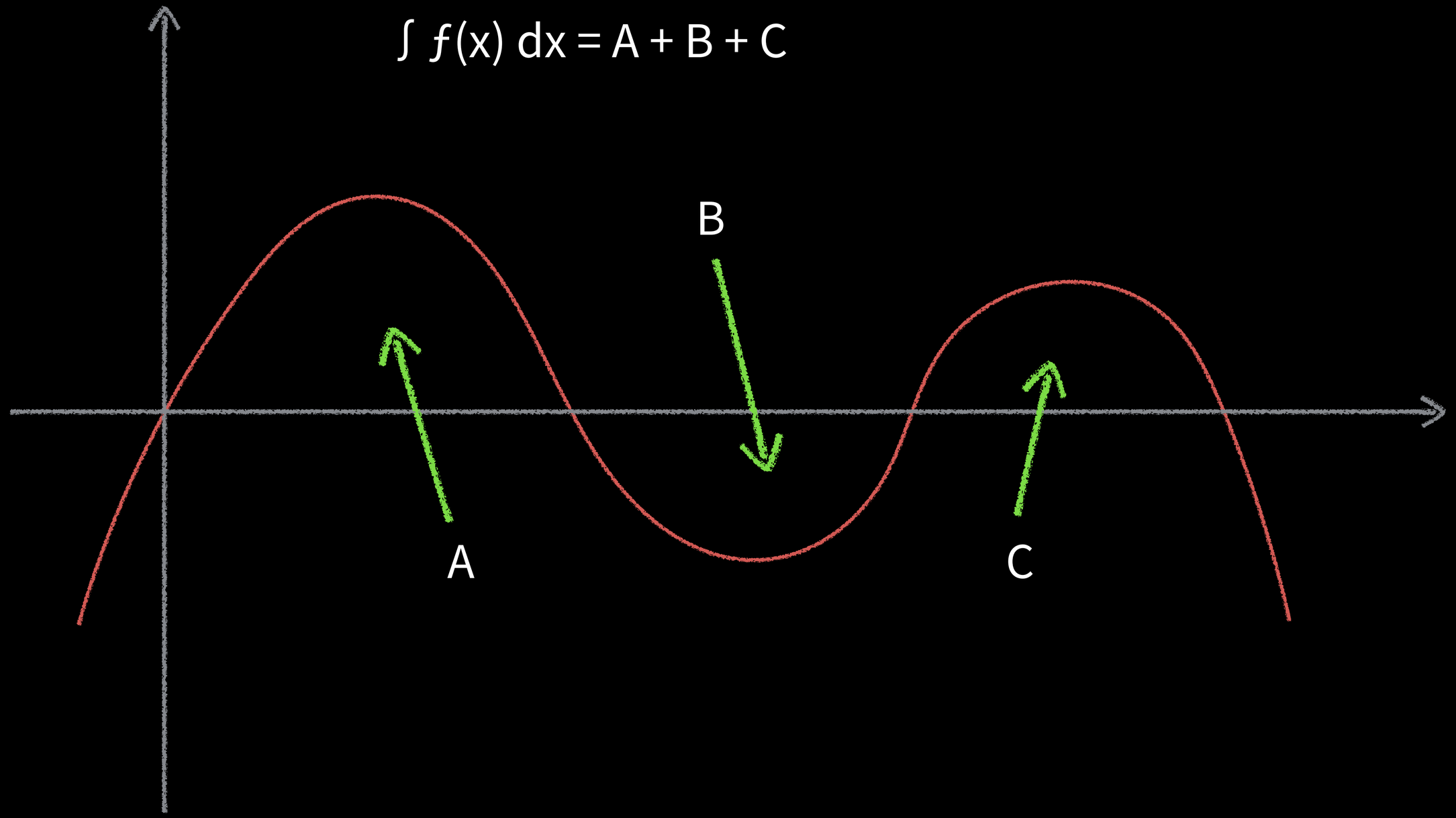
Consider a heavyweight calculation

$$\int f(x)\,dx \approx \sum f(x)\,\Delta x$$



But, by definition, we can:

$$\int f(x)\,dx = A + B + C$$

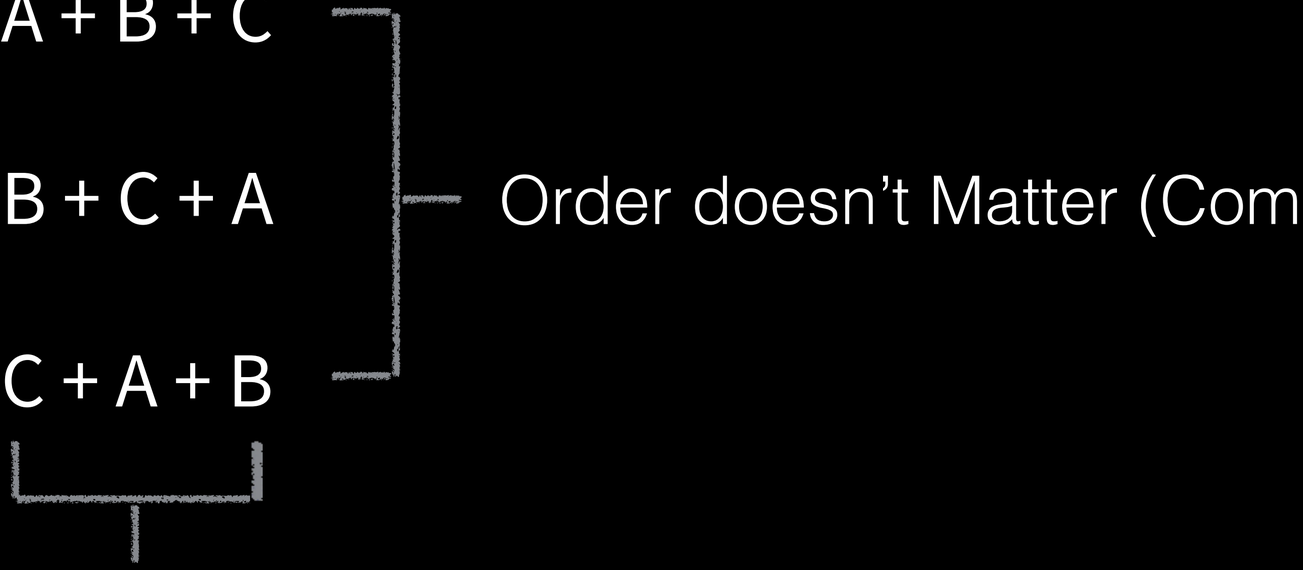


- $\int f(x)\,dx = A$
- $\int f(x)\,dx = B$
- $\int f(x)\,dx = C$

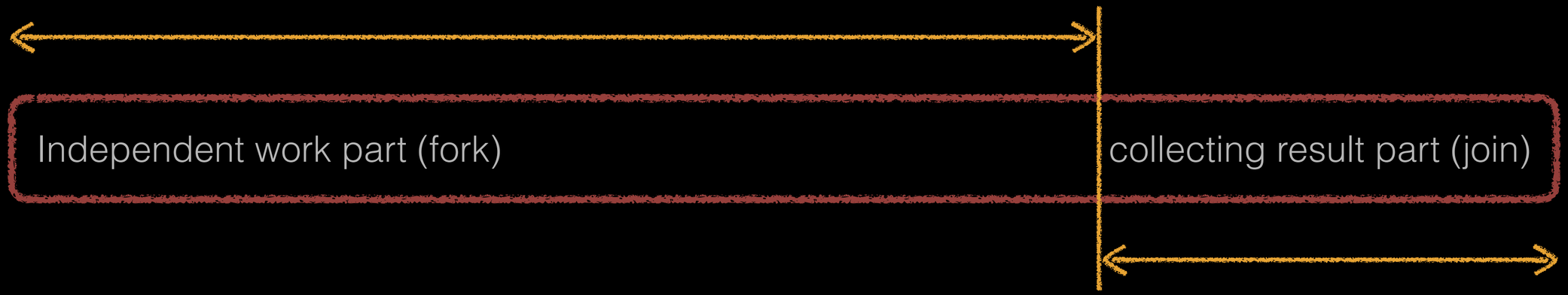
Independent Each Other

Resource Consuming, Heavyweight Task

- A + B + C
- B + C + A
- C + A + B


Order doesn’t Matter (Commutative)
Lightweight Calculation

Thread Safety Matters, or
Mutually Independent while Execution



Should be
SYNCHRONIZED

The integral emulation code

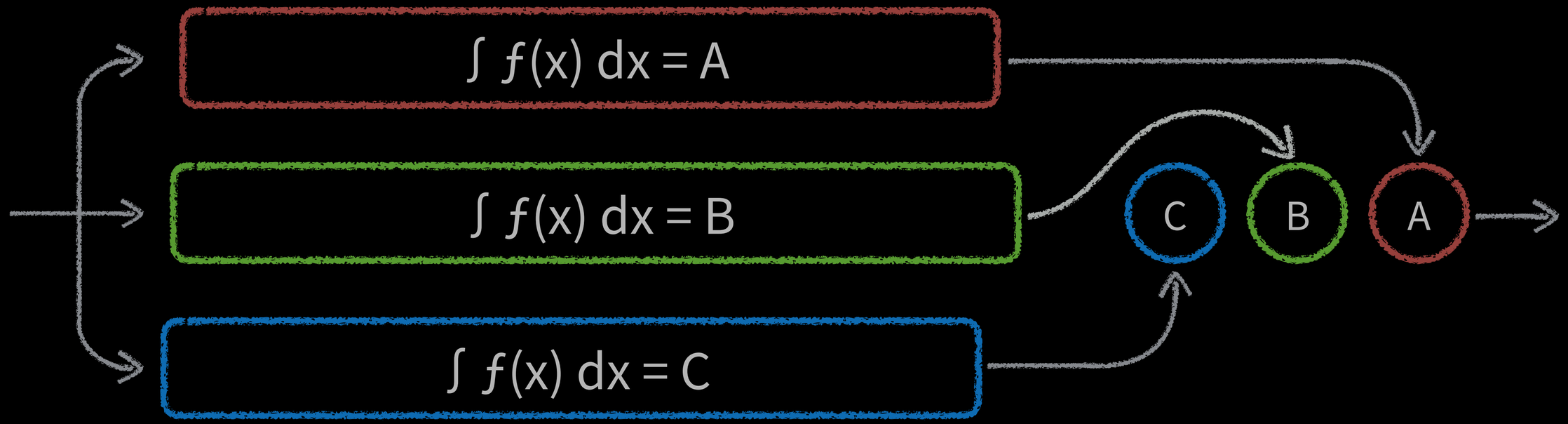
```
int total = 0;

int integral(String sector, int from, int to) {
    log("Sector %s integral from %d" , sector, from);
    sleep((to - from) * 1000L);                          // integral takes very long time
    log("Sector %s integral to %d" , sector, to);
    return (to - from) * 1000;
}

synchronized void accumulate(String sector, int partial) {
    log("Prepare adding sector %s + %d", sector, partial);
    sleep(500L);                                         // accumulation also tasks time
    total += partial;
    log("Done adding sector %s + %d", sector, partial);
}

class Partial implements Runnable {
    final String sector; final int from, to;

    Partial(String s, int f, int t) { sector = s; from = f; to = t; }

    public void run() {
        int partial = integral(sector, from, to);
        accumulate(sector, partial);
    }
}
```

Integral using thread

```
t1 = new Thread(new Partial("A",0,5));
t2 = new Thread(new Partial("B",5,10));
t3 = new Thread(new Partial("C",10,15));

t1.start();
t2.start();
t3.start();

t1.join();
t2.join();
t3.join();

log("RESULT: " + total);
```

~ 6 sec

```
15.121 [Th-2] Sector C integral from 10
15.121 [Th-1] Sector B integral from 5
15.121 [Th-0] Sector A integral from 0

20.135 [Th-2] Sector C integral to 15
20.135 [Th-2] Prepare adding sector C + 5000
20.136 [Th-0] Sector A integral to 5
20.136 [Th-1] Sector B integral to 10

20.639 [Th-2] Done adding sector C + 5000
20.639 [Th-1] Prepare adding sector B + 5000
21.143 [Th-1] Done adding sector B + 5000
21.144 [Th-0] Prepare adding sector A + 5000
21.649 [Th-0] Done adding sector A + 5000
21.649 [main] RESULT: 15000
```


∫ f(x) dx = A
∫ f(x) dx = B
∫ f(x) dx = C
C
B
A

Single play

```
total = integral("ABC", 0, 15);
log("RESULT: " + total);
```

```
26.520 [main] Sector ABC integral from 0
41.524 [main] Sector ABC integral to 15
41.525 [main] RESULT: 15000
```

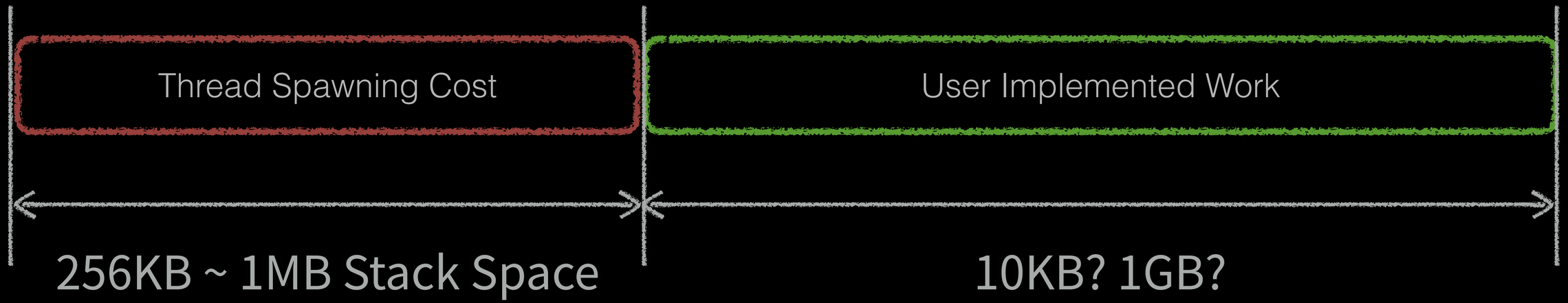
~ 15 sec !!!

Each heavyweight calculation runs on different CPUs

This is typical “CPU” bound

Reduced time drastically by “Parallelism”

A thread has an additional resource


Thread Spawning Cost
User Implemented Work
256KB ~ 1MB Stack Space
10KB? 1GB?

8 CPU, 16GB Machine:

Typical CPU Bound: Numerical Analysis

- 100% 1 CPU: 1 hour with 1 thread, 100 MB memory
- 100% 7 CPU: 8.6 min with 7 thread, 7 × 100 MB memory
  - 7 × 1MB of thread stack, so reduce to sum up is negligible!
  - (1 CPU for main thread)

7 thread is enough.

We are interested in HTTP: the I/O world

CPU : I/O

Network I/O	CPU	Difference
~ 10 ms	~ 50 ns	200,000x
”@_		**Difference**
~ 10 m/h	~ 2,000 km/h	200,000x

Network I/O

=

A ”@_ BLOCKING ✈

I/O typically consume very low CPU power

Basic idea:
Do not block, always I/O with its own thread

- main thread: 
- I/O thread #1: 
- I/O thread #2: 
- I/O thread #3: 
- . . .

As many thread as possible by “time slicing”

But…

I/O Bound

CPU Bound

Typical I/O threading characteristics

- 1 concurrent request: 0.05% CPU, 100KB Memory
- 15,000 concurrent request:
  - 15,000 Thread, 15,000 × 100KB = 1.5 GB Memory
  - 0.05% × 15,000 ÷ 8 CPU = 93.75% / 1 CPU
  - **15, 000 × 1MB Thread Stack = 15 GB!!!**

- Thread stack itself
- Context Switching
- Garbage Collection
- …

In practice, we have “Thread Pool” to utilize resource

But, the “complexity” is another problem

And, error handling raises the complexity more & more

(Thread topic will stop here)

Java 8 Lambda
Higher-order Functions

- Lambda: interface with in/out generic
- Higher-order Function: ~ like “`addXxxxListener(...)`”
- Input Definition Factory: like “`List<InputType>`”

Lambda

```
// Nothing but “interface”
rx.functions.Func1<Integer, String> func = new Func1<Integer, String>() {
    public String call(Integer i) {
        return "prefix" + i;
    }
};

// In Java 8, the interface implementation is equivalent to:
rx.functions.Func1<Integer, String> func = i -> "prefix" + i;
```

```
// before Java 8
addFunctionRunner(new Func1<Integer, String>() {
    public String call(Integer i) {
        return "prefix" + i;
    }
});

// Since Java 8
addFunctionRunner(i -> "prefix" + i);
```

Which one do you prefer?

If you confused, just regard it as:

“Synthetic Sugar”

(But NOT forever)

Higher-order function

Similarly, higher-order function in Java is like:

“addSomeEventListener(lambdaFunction)”

(“Event” makes sense, but NOT forever too)

And we have “standard” set:

`filter()`	filter events by given predicate
`map()`	transform events by given transformer
`flatMap()`	transform + flatten(or, merge)
`reduce()`	apply reduction on events
`forEach()`	for each event, do something, or consuming

Input definition factory: like a `List<Event>`

And “type inference” for lambda

```
List<String> list = Arrays.asList("a", "b", "c");
// list is container for “string event”,
// so “x” is string by type inference
list.forEach(x -> log(x.toUpperCase());

List<Integer> list = Arrays.asList(1, 2, 3);
// list is container for “integer event”,
// so “x” is integer by type inference
list.forEach(x -> log(x + x));
```

And remember … the “Laziness”

Ready to Rock n’ Roll

RxJava: Concepts

ReactiveX: Introduction

The Reactive Manifesto (v2.0)

Responsive: REACT to user

Resilient: REACT to errors & failure

Elastic: REACT to workload

Message-driven: REACT to (asynchronous) events & messages

Iterable<T>, Future<T>, Observable<T>

	single	multiple
synchronous	T	Iterable<T>
asynchronous	Future<T>	Observable<T>

Observables fill the gap by being the ideal way to access asynchronous sequences of multiple items

http://reactivex.io/intro.html

Pull vs Push

event	Iterable:pull	Observable:push
retrieve data	`T next()`	`onNext(T)`
discover error	`throws Exception`	`onError(Exception)`
complete	`!hasNext()`	`onCompleted()`

An Observable is the asynchronous/push “dual”
to the synchronous/pull Iterable

http://reactivex.io/intro.html

Observable<T> Taxonomy

Operator Type	Source Type	Result Type	Example
Anamorphic (unfold)	T	Observable<T>	.create() .from() .just()
Bind (map)	Observable<T>	Observable<R>	.amb() .filter() .map()
Catamorphic (fold, or reduce)	Observable<T>	T	.first() .last() .take()

Cold vs Hot

Cold	Hot
Asynchronous requests (Observable.from)	UI Events (Mouse move/click/…)
Whenever Observable.create() is used	Timer Events
Subscriptions to queues	Broadcasts (ESB channels/ UDP network packets)
on-demand sequences	Price ticks from a trading exchange

departure: for-loop replacement

```
log("START");
try {
    List<Integer> l = // [1, 2, 3, null, 5]
    for (Integer each : l) {
        log("> just: " + each);
        if (!filter(each)) { continue; }
        log("> filter: " + each);
        Integer y = map(each);
        log("> map: " + y);
        log("subscribe: " + y);
    }
    log("completed");
} catch (Throwable t) {
    log("error: " + t);
} finally {
   log("END");
}

boolean filter(Integer x) {
    return x % 2 == 1;
}

Integer map(Integer in) {
    return in * 10;
}
```

```
log("START");
Observable.just(1, 2, 3, null, 5)
    .doOnNext(x -> log("> just: " + x))
    .filter(x -> x % 2 == 1)
    .doOnNext(x -> log("> filter: " + x))
    .map(x -> x * 10)
    .doOnNext(x -> log("> map: " + x))
    .subscribe(
        x  -> log("subscribe: " + x),
        ex -> log("error: " + ex),
        () -> log("completed")
    );
log("END");
```

```
11.119 [main] START
11.121 [main] > just: 1
11.124 [main] > filter: 1
11.125 [main] > map: 10
11.127 [main] subscribe: 10
11.128 [main] > just: 2
11.129 [main] > just: 3
11.129 [main] > filter: 3
11.130 [main] > map: 30
11.130 [main] subscribe: 30
11.131 [main] > just: null
11.140 [main] error: java.lang.NullPointerException
11.142 [main] END
```

from imperative to declarative

Old-fashioned loop

```
// List<Event> events = …
try {
    for (Event e : events) {
        onNextEvent(e);
    }

    onSuccessfulComplete();
} catch (Throwable t) {
    onErrorEvent(t);
}
```

Iterable.forEach()

```
// List<Event> events = …
try {
    events.forEach(e -> onNextEvent(e));

    onSuccessfulComplete();
} catch (Throwable t) {
    onErrorEvent(t);
}
```

Java 8 Stream

```
// Stream<Event> stream = …
try {
    stream
        .map(e -> transform(e))
        // more transformations
        .forEach(e -> onNextEvent(e));

    onSuccessfulComplete();
} catch (Throwable t) {
    onErrorEvent(t);
}
```

RxJava Observable

```
// Observable<Event> observable = …
observable
     .map(e -> transform(e))
     // more transformations
     .subscribe(
          e  -> onNextEvent(e),
          t  -> onSuccessfulComplete(),
          () -> onErrorEvent(t)
     );
```

RxJava with async

Event consuming(subscription)

for each long-running work

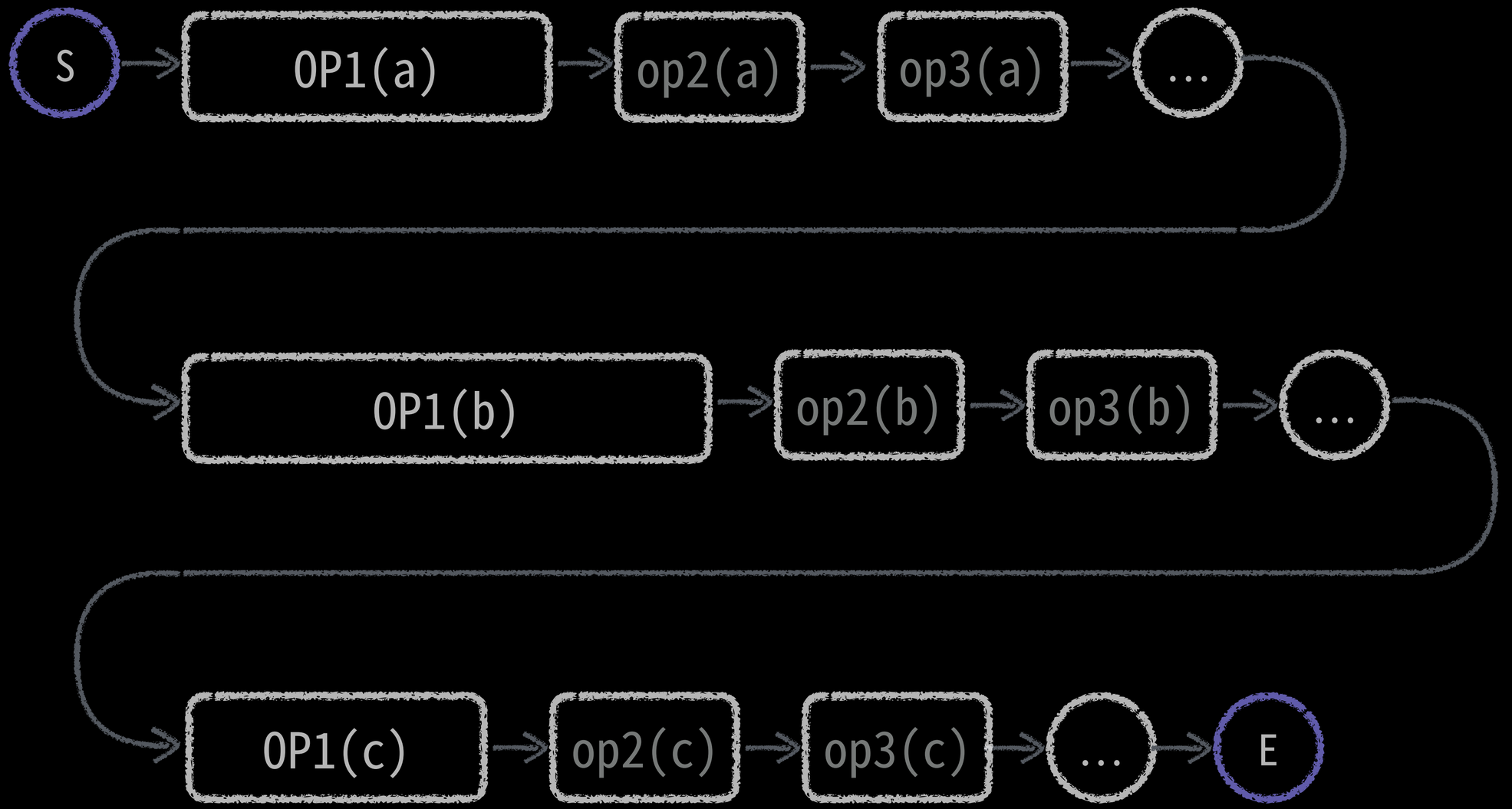
```
// input expressed as: ‘a’, ‘b’, ‘c’
Output hardWork(Input in) {
    log("Beginning Work: " + in.id());
    // Very HARD WORK such as Network I/O
    log("Done Work: " + in.id());
    return out;
}
```

Observable is serial by default

```
log("start")
observable.map(x -> hardWork(x))
    .filter(x -> f(x))
    .operator(x -> g(x))
    ...
    .subscribe(x -> log("result: " + x);
log("end");
```

```
55.798 [main] start
55.939 [main] Beginning Work: a
56.441 [main] Done Work: a
56.441 [main] result: a
56.441 [main] Beginning Work: b
57.042 [main] Done Work: b
57.043 [main] result: b
57.044 [main] Beginning Work: c
57.447 [main] Done Work: c
57.447 [main] result: c
57.448 [main] end
```

~ 1.65 sec

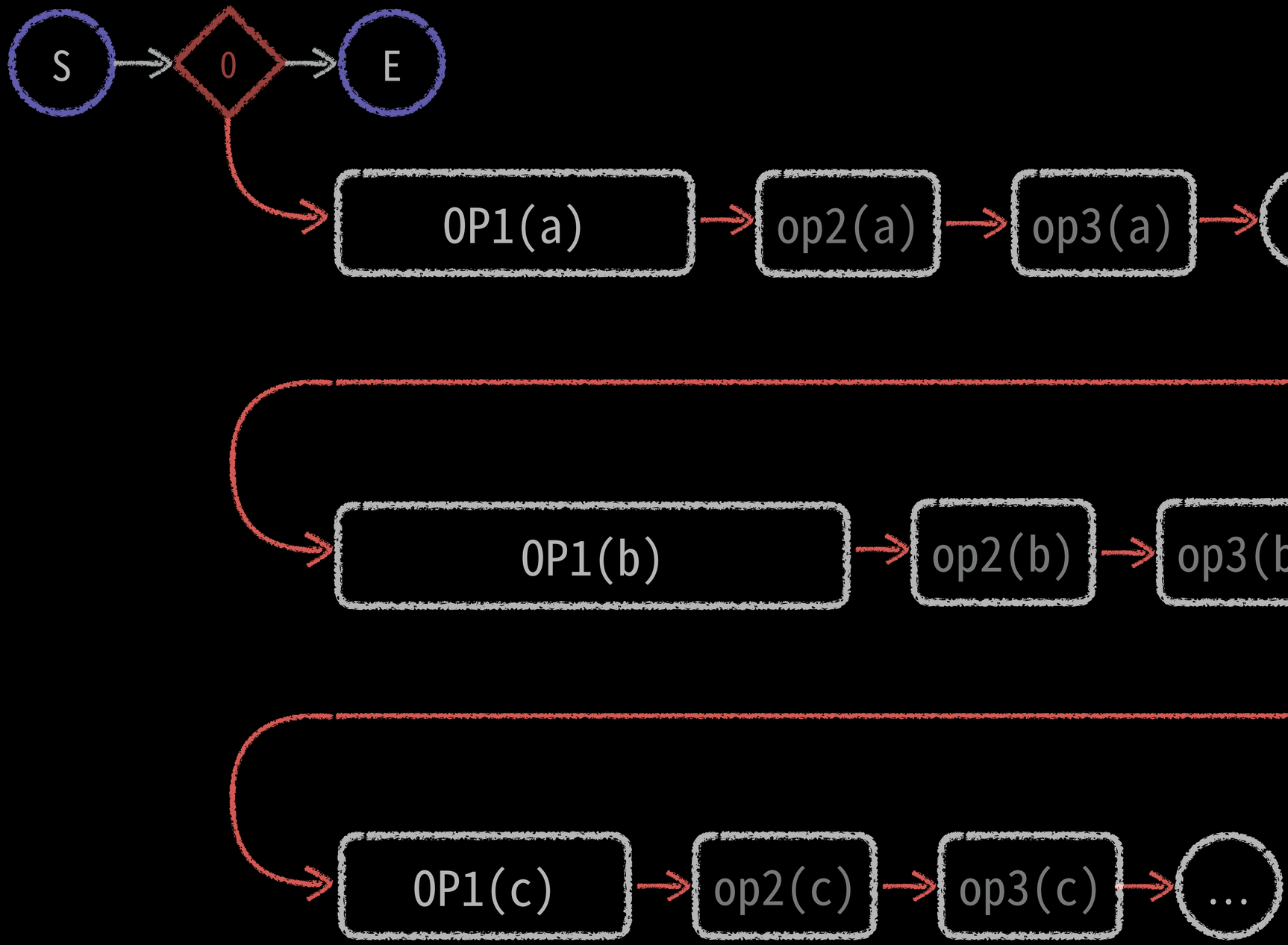

S
OP1(a)
op2(a)
op3(a)
...
OP1(b)
op2(b)
op3(b)
...
OP1(c)
op2(c)
op3(c)
...
E

subscribeOn() unexpected

```
log("start")
observable.map(x -> hardWork(x))
    .filter(x -> f(x))
    .operator(x -> g(x))
    .subscribeOn(scheduler)
    ...
    .subscribe(x -> log("result: " + x);
log("end");
```

```
31.165 [main] start
31.476 [main] end
31.482 [S-AO] Beginning Work: a
31.984 [S-AO] Done Work: a
31.985 [S-AO] result: a
31.987 [S-AO] Beginning Work: b
32.591 [S-AO] Done Work: b
32.592 [S-AO] result: b
32.592 [S-AO] Beginning Work: c
32.996 [S-AO] Done Work: c
32.996 [S-AO] result: c
```

1.831 sec ?!


S
0
E
OP1(a)
op2(a)
op3(a)
...
OP1(b)
op2(b)
op3(b)
...
OP1(c)
op2(c)
op3(c)
...

subscribeOn() Right Way

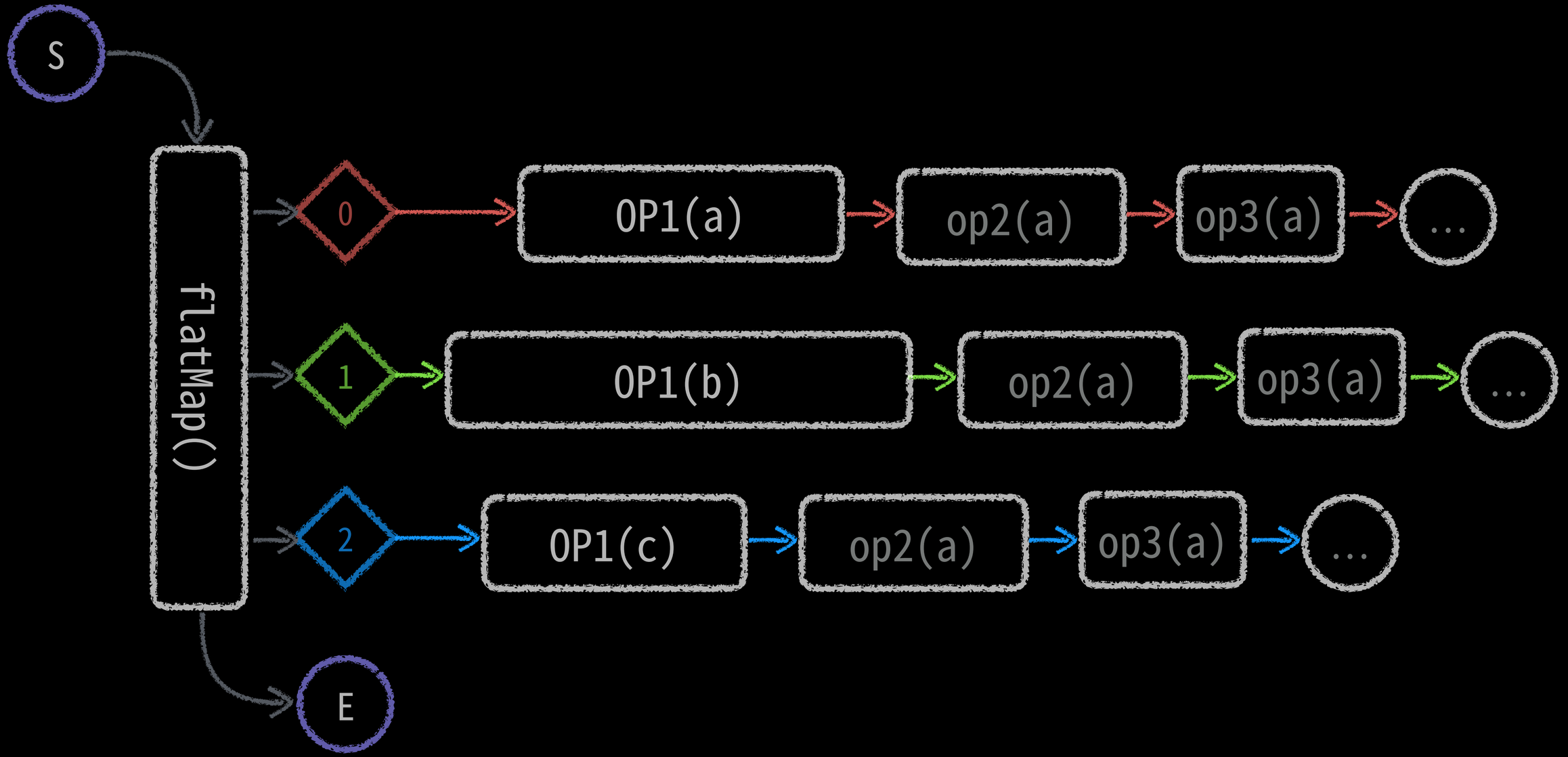
```
log("start")
observable
    .flatMap(x ->
        Observable.fromCallable(() -> hardWork(x))
            .subscribeOn(scheduler)
    )
    .filter(x -> f(x))
    .operator(x -> g(x))
    ...
    .subscribe(x -> log("result: " + x);
log("end");
```

subscribeOn() Alternative Way

```
log("start")
observable
    .flatMap(x ->
        Observable.defer(() ->
            Observable.just(hardWork(x))
        ).subscribeOn(scheduler)
    )
    .filter(x -> f(x))
    .operator(x -> g(x))
    ...
    .subscribe(x -> log("result: " + x);
log("end");
```

```
42.241 [main] start
42.493 [main] end
42.496 [S-A1] Beginning Work: b
42.496 [S-A2] Beginning Work: c
42.496 [S-A0] Beginning Work: a
42.901 [S-A2] Done Work: c
42.901 [S-A2] result: c
43.003 [S-A0] Done Work: a
43.003 [S-A0] result: a
43.101 [S-A1] Done Work: b
43.101 [S-A1: result: b
```

0.860 sec !!

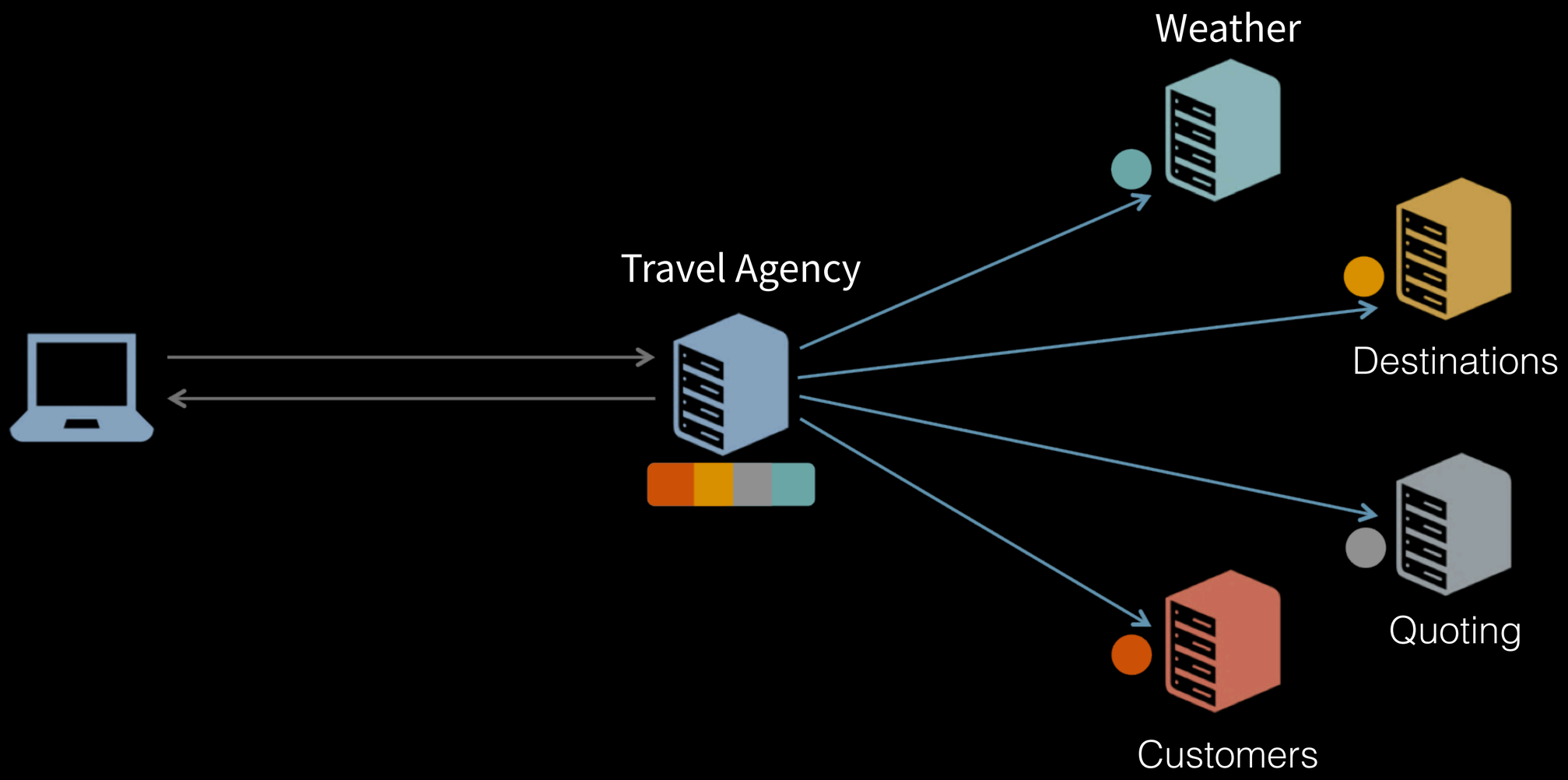

S
flatMap()
0
OP1(a)
op2(a)
op3(a)
...
1
OP1(b)
op2(a)
op3(a)
...
2
OP1(c)
op2(a)
op3(a)
...
E

go further: more realistic pattern

```
log("prepare");
Observable<Long> a = Observable.just("bread")
    .doOnNext(x -> log("substream: " + x))
    .map(x -> repository.purchase(x, 1))
    .subscribeOn(schedulerA());

Observable<Long> b = Observable.just("cucumber")
    .doOnNext(x -> log("substream: " + x))
    .map(x -> repository.purchase(x, 3))
    .subscribeOn(schedulerA());

Observable<Long> c = Observable.just("beef")
    .doOnNext(x -> log("substream: " + x))
    .map(x -> repository.purchase(x, 5))
    .subscribeOn(schedulerA());

log("start");
Observable.just(a, b, c)
    .flatMap(x -> x)
    .observeOn(schedulerB())
    //  .reduce((x, y) -> x.add(y))
    .subscribe(x -> log("Q'ty: " + x));

log("end");
```

```
14.186 [main] prepare
14.359 [main] start
14.489 [main] end

14.490 [S-A1] substream: cucumber
14.490 [S-A2] substream: beef
14.490 [S-A0] substream: bread

14.492 [S-A0] >>> Work: 1 bread
14.491 [S-A2] >>> Work: 5 beef
14.491 [S-A1] >>> Work: 3 cucumber
14.893 [S-A2] <<< Work: 5 beef
14.894 [S-B1] Q'ty: 20
14.993 [S-A0] <<< Work: 1 bread
14.993 [S-B2] Q'ty: 5
15.293 [S-A1] <<< Work: 3 cucumber
15.294 [S-B3] Q'ty: 24
```

Example #1: Jersey

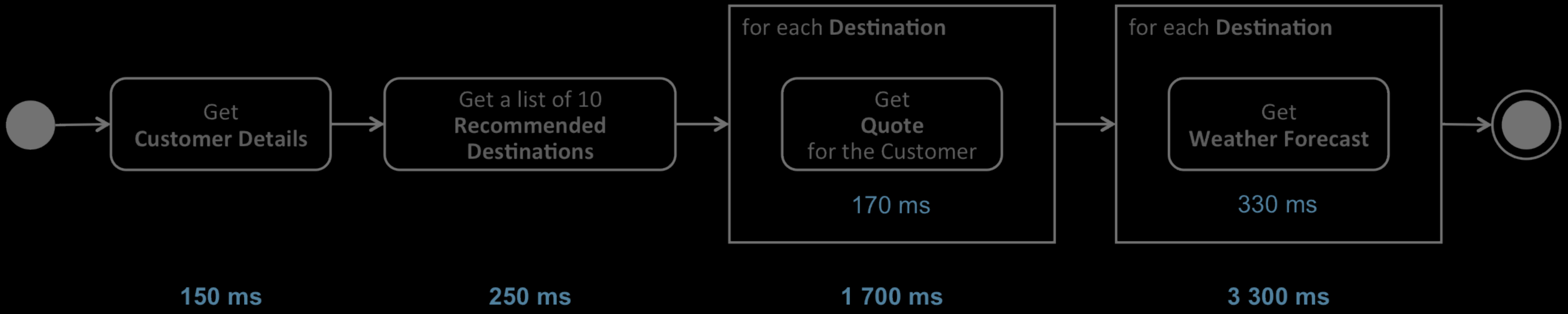


https://jersey.java.net/documentation/latest/user-guide.html#rx-client

Naive Approach

```
List<Destination> recommended = Collections.emptyList();
try {
    recommended = destination.path("recommended").request()
            .header("Rx-User", “Sync”).   // Identify the user.
            .get(new GenericType<List<Destination>>() {});  // Return a list of destinations.
} catch (Throwable throwable) {
    errors.offer("Recommended: " + throwable.getMessage());
}

// Forecasts. (depend on recommended destinations)
Map<String, Forecast> forecasts = new HashMap<>();
for (Destination dest : recommended) {
    try {
        forecasts.put(dest.getDestination(),
                forecast.resolveTemplate("destination", dest.getDestination()).request().get(Forecast.class));
    } catch (Throwable throwable) {
        errors.offer("Forecast: " + throwable.getMessage());
    }
}
```

https://jersey.java.net/documentation/latest/user-guide.html#rx-client



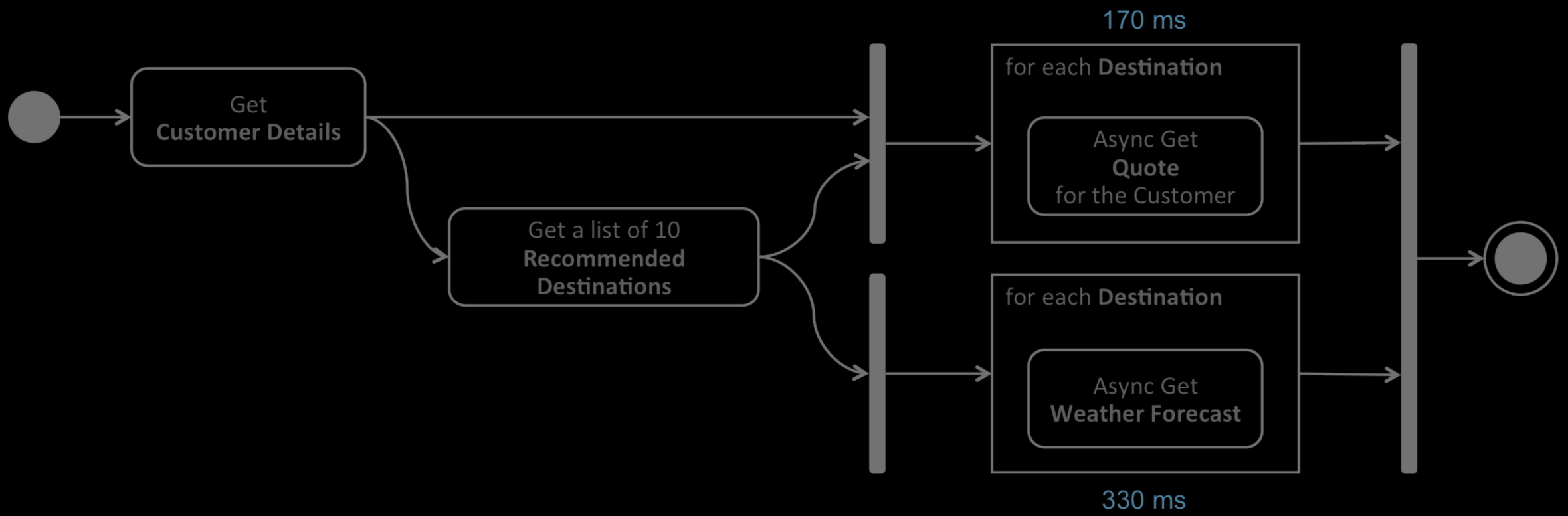
5 400 ms

https://jersey.java.net/documentation/latest/user-guide.html#rx-client

Optimized (Async) Approach

```
destination.path("recommended").request().header("Rx-User", "Async").async()
        .get(new InvocationCallback<List<Destination>>() {
            public void completed(final List<Destination> recommended) {
                final CountDownLatch innerLatch = new CountDownLatch(recommended.size());
                final Map<String, Forecast> forecasts = Collections.synchronizedMap(new HashMap<>());
                for (final Destination dest : recommended) {
                    forecast.resolveTemplate("destination", dest.getDestination()).request().async()
                            .get(new InvocationCallback<Forecast>() {
                                public void completed(final Forecast forecast) {
                                    forecasts.put(dest.getDestination(), forecast); innerLatch.countDown();
                                }

                                public void failed(final Throwable throwable) {
                                    errors.offer("Forecast: " + throwable.getMessage()); innerLatch.countDown();
                                }
                            });
                }

                try {
                    if (!innerLatch.await(10, TimeUnit.SECONDS))
                        errors.offer("Inner: Waiting for requests to complete has timed out.");
                } catch (final InterruptedException e) {
                    errors.offer("Inner: Waiting for requests to complete has been interrupted.");
                }

                // Continue with processing.
            }

            public void failed(final Throwable throwable) { errors.offer("Recommended: " + throwable.getMessage()); }
        });
```

https://jersey.java.net/documentation/latest/user-guide.html#rx-client



730 ms

https://jersey.java.net/documentation/latest/user-guide.html#rx-client

Optimized, but another problem: “Callback Hell”

Reactive Approach

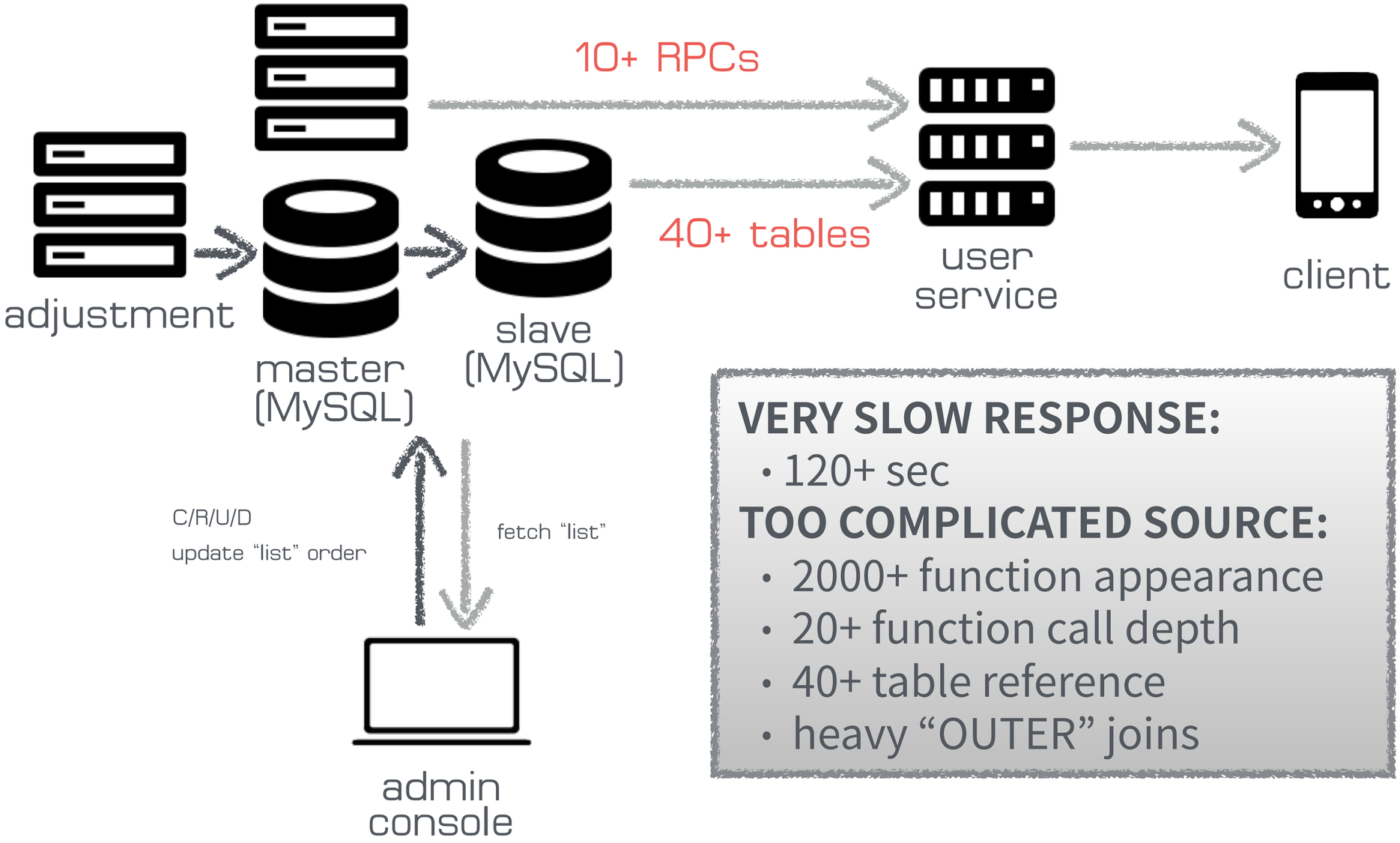
```
Observable<Destination> recommended = RxObservable.from(dest)
        .path("recommended")
        .request()
        .header("Rx-User", "RxJava")
        .rx().get(new GenericType<List<Destination>>() {})
        .onErrorReturn(t -> emptyList())
        .flatMap(Observable::from)
        .cache();

Observable<Forecast> forecasts = recommended.flatMap(dest ->
            RxObservable.from(forecast)
                .resolveTemplate("destination", dest.getDestination())
                .request().rx().get(Forecast.class)
                .onErrorReturn(t -> new Forecast(dest.getDestination(), "N/A")
        )
);

Observable<Recommendation> recommendations = Observable.zip(
        recommended,
        forecasts,
        Recommendation::new);
```

https://jersey.java.net/documentation/latest/user-guide.html#rx-client

Example #2: Original

1. In the beginning, … darkness all across the code

Existing Structure (SpringCamp 2016)



VERY SLOW RESPONSE:

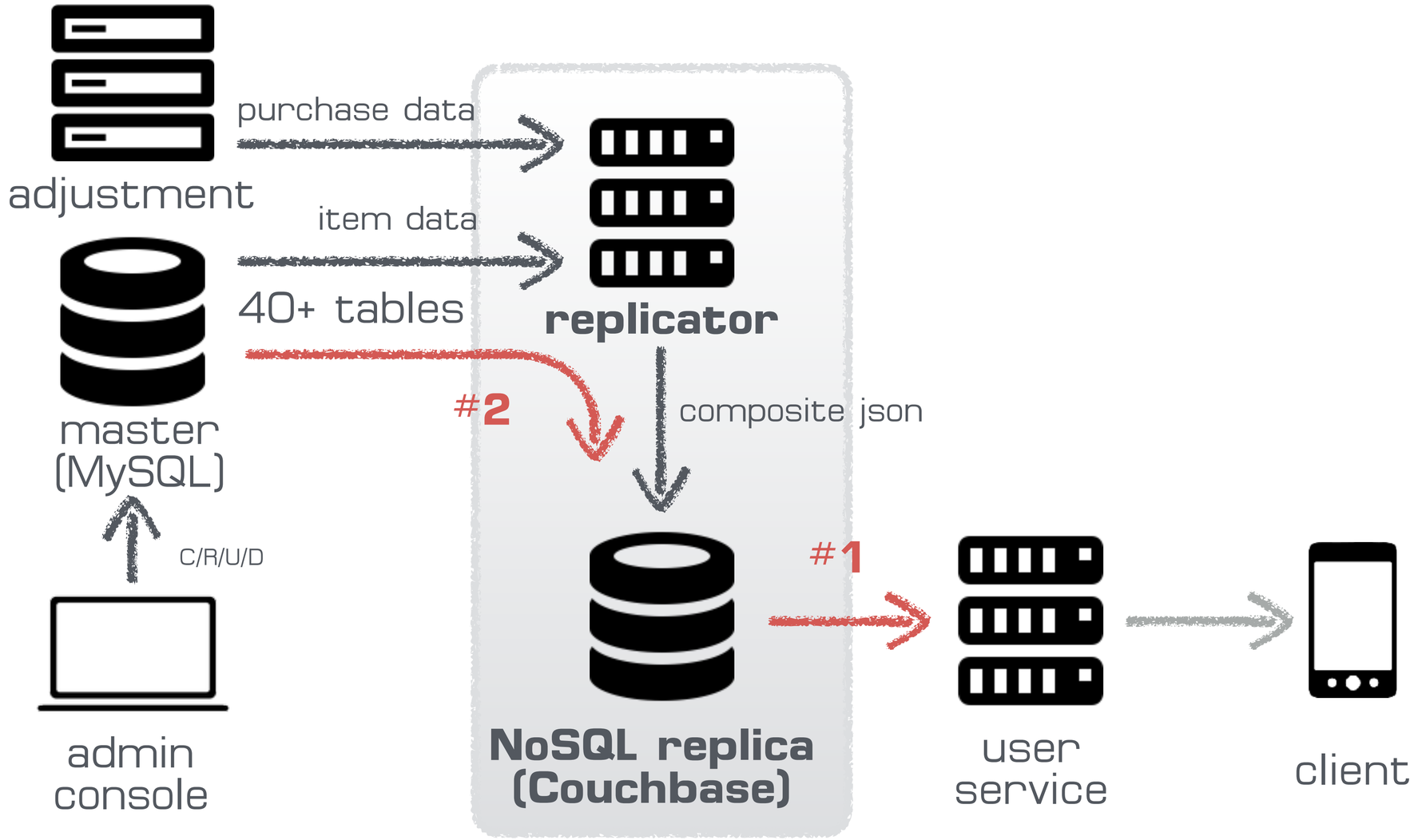
- 120+ sec

TOO COMPLICATED SOURCE:

- 2000+ function appearance
- 20+ function call depth
- 40+ table reference
- heavy “OUTER” joins

2. “Let there be light,” and there was light

Asymmetric Replication (SpringCamp 2016)



#1-1: Fetch list of items (barely reactive)

```
List<FixedDeal> fixed = fixedDealService.find(cid, bid);
FixedDealPagination pagination = FixedDealPagination.paginate(fixed, limit);

FixedDealPage currentPage = pagination.find(pageNo);
PaginationOffset offset = PaginationOffset.adjust(currentPage);

List<String> plain = plainDealService.find(cid, bid, offset);

List<String> collatedIds = PageRankCollator.collate(currentPage, plain)

return Observable.from(collatedIds)
        .concatMapEager(repository::get)
        .map(JsonDocument::content)
        .map(stickerFilter::apply)
        .map(priceFilter::apply)
        .map(viewAttributeFilter::apply)
        .toList()
        .toBlocking()
        .singleOrDefault(Json.DEAL_LIST_DEFAULT);
```

#1-2: Fetch list of items (almost reactive)

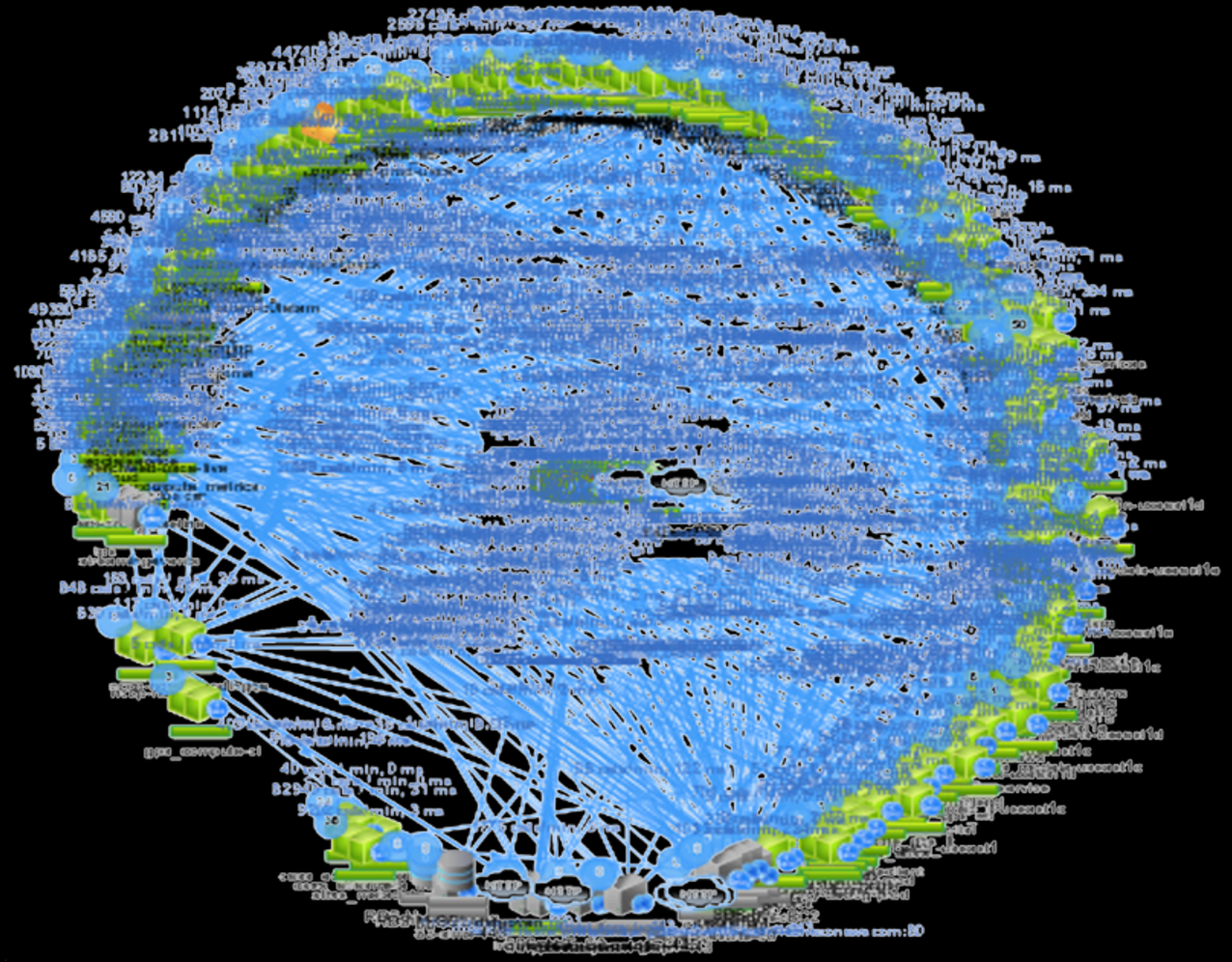
```
Observable<FixedDeal> fixed = fixedDealService.prepare(cid, bid, pageNo, limit);
Observable<PlainDeal> plain = plainDealService.prepare(cid, bid, pageNo, limit);

return Observable.zip(fixed, plain, DealId::collate)
    .flatMap(DealId::targets)
    .concatMapEager(repository::get)
    .map(JsonDocument::content)
    .map(stickerFilter::apply)
    .map(priceFilter::apply)
    .map(viewAttributeFilter::apply)
    .toList()
    .toBlocking()
    .singleOrDefault(Json.DEAL_LIST_DEFAULT);
```

#2: Replication (totally reactive)

```
// 1. gather from N tables concurrently, then aggregate
// 2. insert to replica persistence
Observable<Deal> deal = Observable.just(dealNo)
    .flatMap(id -> dealRepository.findDeferred(id))
    .retry(3)
    .onErrorReturn(throwable -> Deal.ERROR);

Observable<DealCategory> category = Observable.just(categoryNo)
    .flatMap(id -> dealCategoryRepository.findDeferred(id))
    .retry(3)
    .onErrorReturn(throwable -> DealCategory.ERROR);

Observable<DealScore> score = Observable.just(dealNo)
    .flatMap(id -> dealScoreRepository.findDeferred(id))
    .onErrorReturn(throwable -> DealScore.DEFAULT);

Observable.zip(deal, category, score, DealJson::aggregate)
    .observeOn(schedulerSlot.dealSourceScheduler())
    .subscribe(src -> replicaService.send(src));
```

Example #3: Netflix

A Death Star Diagram: Netflix Server Topology



References

- http://reactivex.io/
- http://www.reactivemanifesto.org/
- http://www.introtorx.com/content/v1.0.10621.0/14_HotAndColdObservables.html
- https://jersey.java.net/documentation/latest/index.html
- https://en.wikipedia.org/wiki/Anamorphism
- https://en.wikipedia.org/wiki/Catamorphism
- Reactive Programming with RxJava (Book)