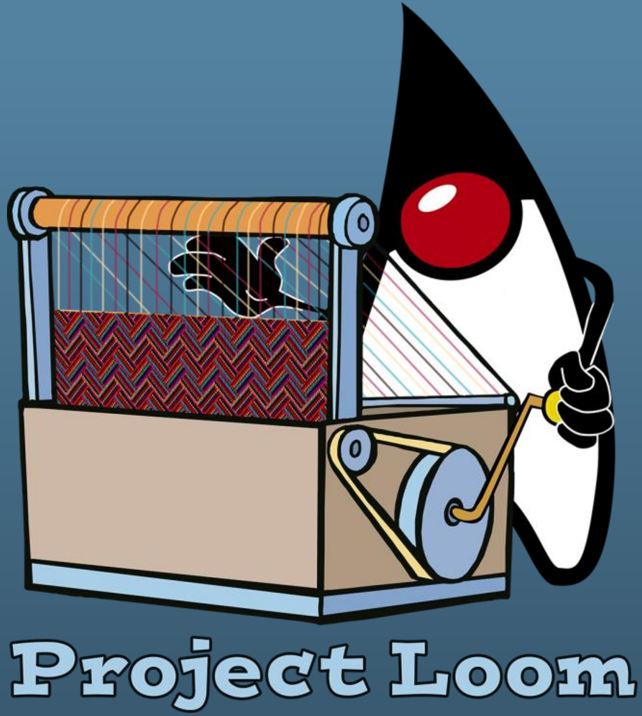
Project Loom Overview

Brian Goetz (@briangoetz)
Java Language Architect, Oracle



ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.

A brief history of async programming

- In the 80s, we programmed with asynchronous code
  - Hard to write, hard to read, hard to debug
- Since the early 90s, threads have been our primary concurrency tool
  - Allows users to program with simple, sequential code
  - Sequential code is readable by humans!
- But, threads are somewhat heavyweight abstractions
  - Megabyte-scale entities, can’t have millions of them
- Recently, people have been reaching for reactive APIs
  - Kind of like what we did in the 80s, with the same effect

Server programming with threads

- When a request comes in, it is dispatched to a thread
  - Holds onto that thread for the duration of the request
- Processing the request might involve a lot of waiting
  - Reading request from socket
  - Database or other service requests
  - Writing response to socket
- Easy, but doesn't scale to millions of threads
  - Easy to read, write, debug
  - Limited number of threads means limited number of concurrent requests
    - Even if CPU has lots of free cycles

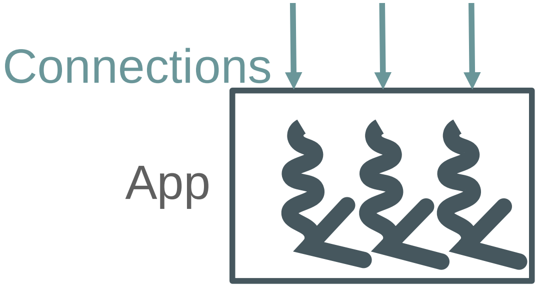
ORACLE

Server programming with reactive APIs

A transitional solution

- Reactive APIs let us express a sequence of async operations
  - “Do this, then this, then maybe that”
  - Request is not tied to a single thread for the duration
- More scalable, but *much* harder to debug
- Also harder to read
  - Framework ceremony obscures business logic
- Cannot mix synchronous and asynchronous APIs

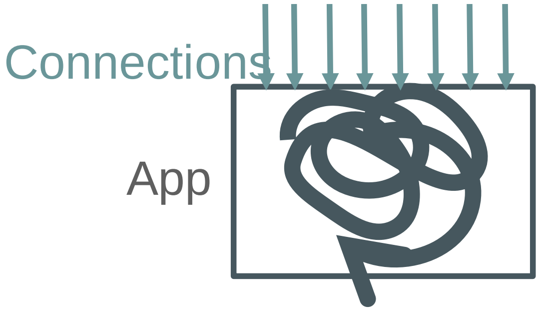
A bad choice



simple
less scalable

SYNC

OR



scalable,
complex,
non-interoperable,
hard to debug/profile

ASYNC

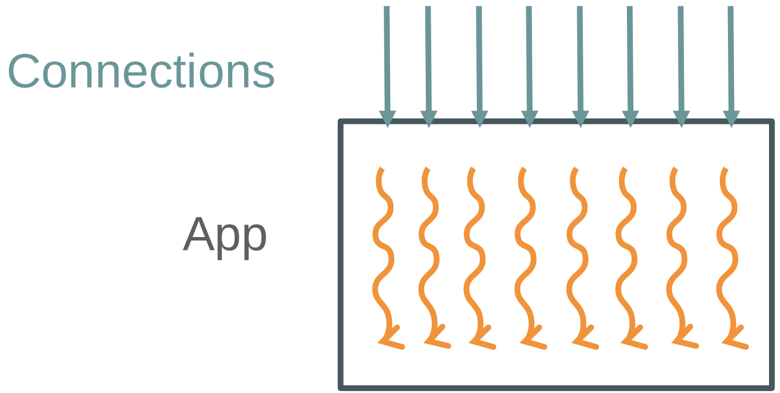
The best of both worlds?

- The threaded programming model is what users want
  - Code is readable, does what it looks like it does
  - But doesn’t (currently) scale beyond 10K threads
- If we could make threads scale better, we would be less tempted to reach for async!
- Project Loom aims to do just that
  - *Fibers* are lightweight threads
  - Few hundred bytes each, rather than megabytes
  - Can have millions of them
  - Same familiar, readable, debuggable programming model as threads

ORACLE

The best of both worlds?

Codes like sync, scales like async



Continuations

The low-level plumbing

- A *continuation* is a VM mechanism for restartable computations
  - Continuation extends Runnable
  - Low-level mechanism for creating concurrency primitives
- The task can *pause* the continuation
  - On pausing, control returns to the initiator
  - JVM unwinds call frames between initiator and pause point, stores in heap
  - Continuation can be resumed later – possibly on another thread
- Doesn’t tie up thread while paused!
  - Can pause before a blocking operation, and resume when it is complete
- Fast task switching

Fibers

Lightweight threads

- A *fiber* is a lightweight thread
  - Built on Continuation
  - Higher-level lightweight thread abstraction
  - Uses ForkJoinPool for task scheduling
- Fibers are lightweight
  - Few hundreds of bytes, not megabytes
  - Cheap to create and schedule
- JDK libraries instrumented to be fiber-aware
  - Blocking IO / JUC operations pause current fiber, resume when unblocked

Programming with Fibers

- Fibers have all the benefits of threads
  - Simple, sequential code
  - Easy to read and debug

```
Fiber<String> fiber = Fiber.schedule(task);
String result = fiber.join();
```

- But scale better
  - Can easily have 1M fibers on a desktop-class system
- And can interoperate with reactive code

```
CompletableFuture<?> result = Fiber.schedule(task).toFuture();
```


Show us the numbers!

- Here's a JAX-RS service that simulates a typical request
  - Assume computeValue() takes 100ms

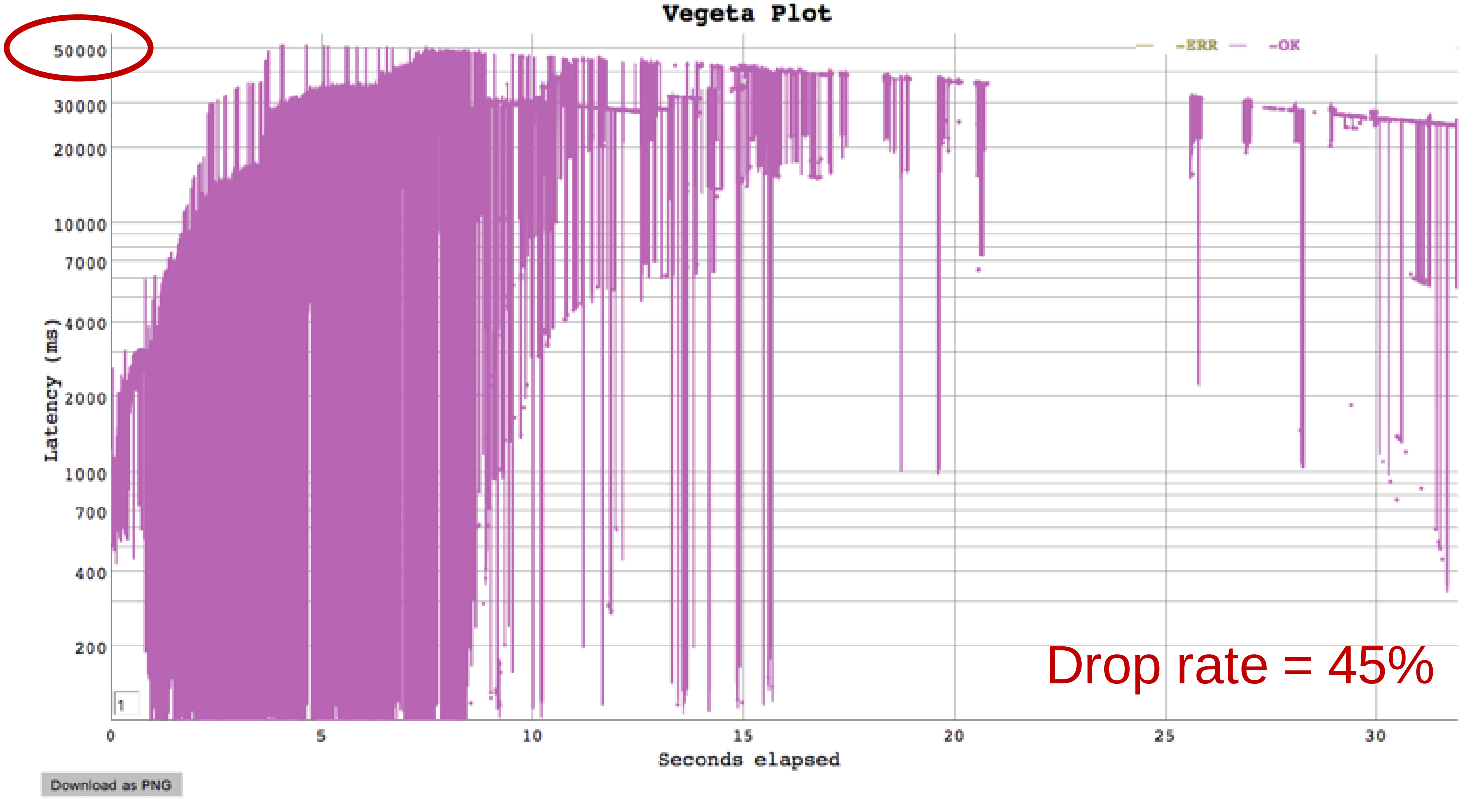
```
@GET
@Path("greeting")
@Produces(MediaType.APPLICATION_JSON)
public String greeting() {
    return "{ \"message\": \"" + computeValue() + "\" }";
}
```

- Run it on Jetty+Jersey with 200 threads…
  - Little's Law says we can only get 2000 reqs/sec through this
  - Let's throw some additional load at it…

ORACLE

Show us the numbers!

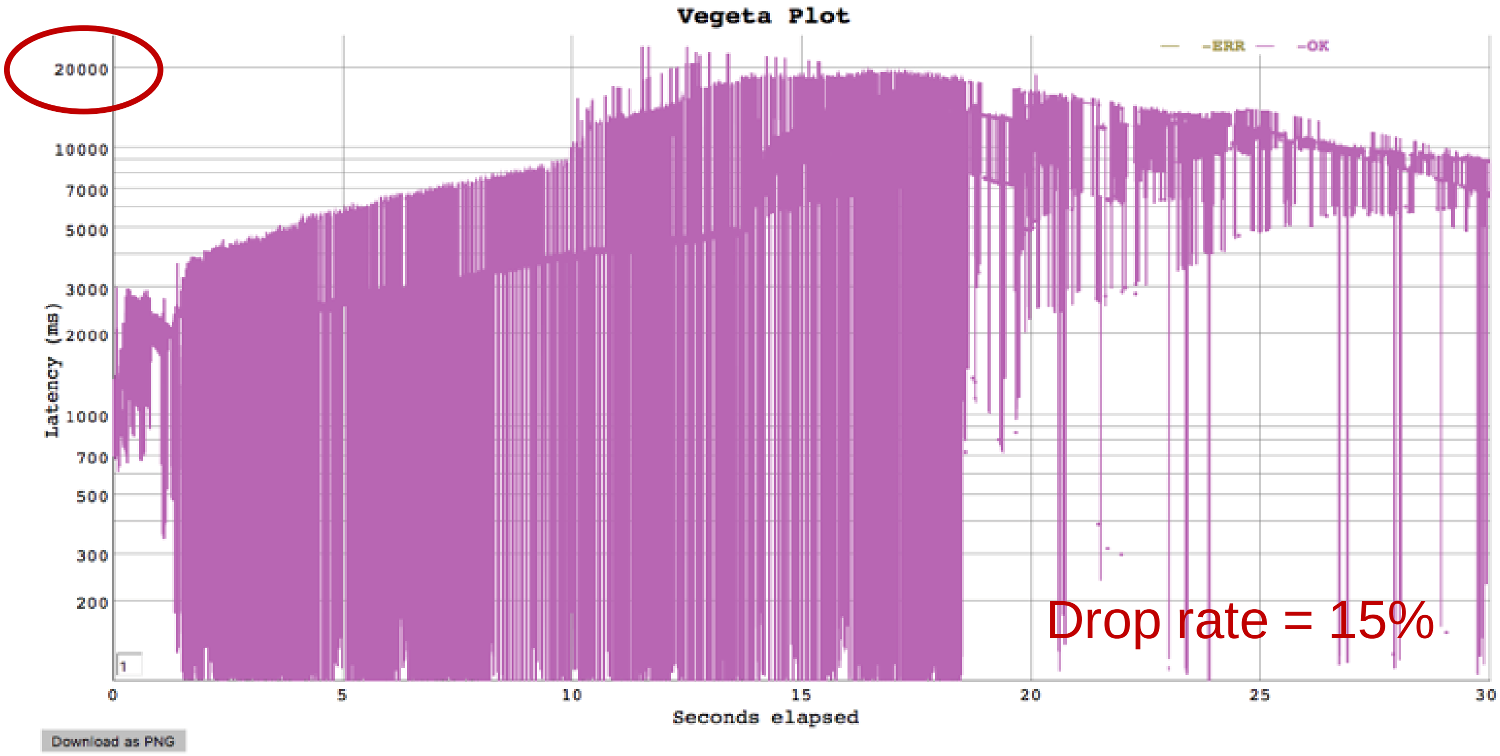
Response time plot with 200 threads, 5000 reqs/sec



ORACLE

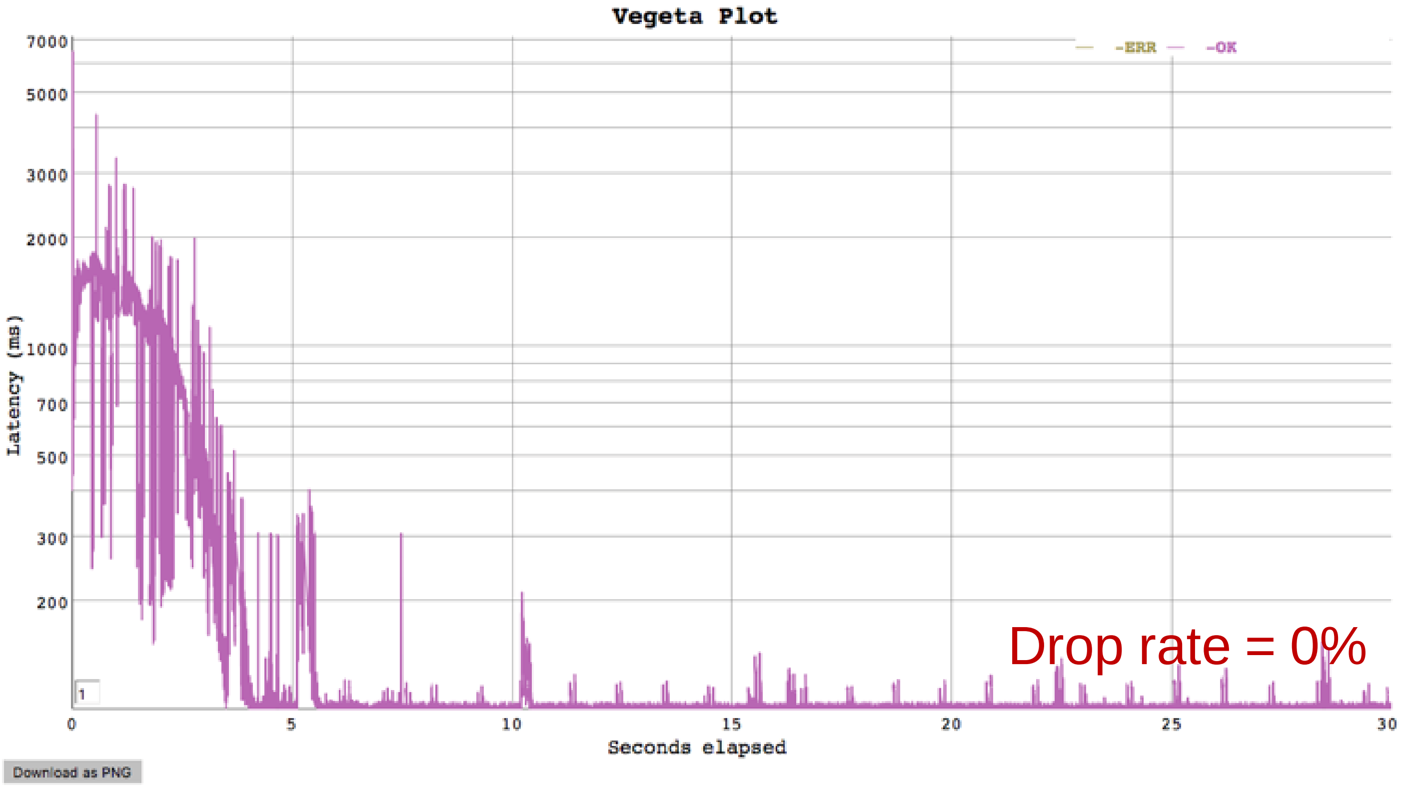
Show us the numbers!

Increase thread pool to 400 threads, same load



Show us the numbers!

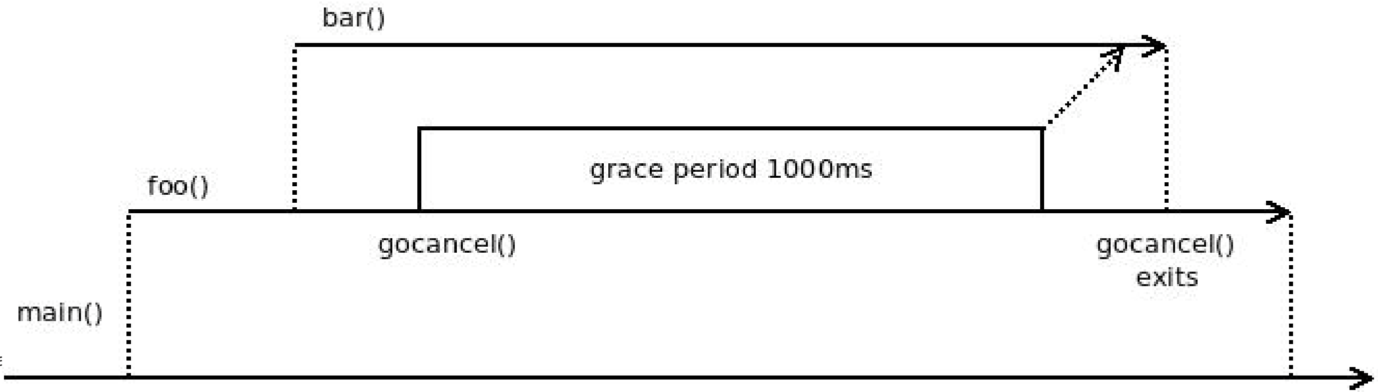
Replace thread pool with fibers, same load



Drop rate = 0%

Structured concurrency

- Other languages are starting to embrace *structured concurrency*
  - Sustrik, *Structured Concurrency*
  - Smith, *Go statement considered harmful*
- All fibers have a *parent fiber*
  - Parent cannot exit until all children exit
- Creates a natural locus for cancellation, deadlines, etc



Structured concurrency

- Fibers live in a *fiber scope*
  - Fiber scopes can be nested arbitrarily deeply, forming a tree
  - Cancelling / interrupting a scope will cancel all fibers in that scope
  - Scopes are AutoClosable, where close() waits for all children
    - Plays nicely with try-with-resources

```
try (var scope = FiberScope.cancellable()) {
    Fiber<?> child1 = scope.schedule(task1);
    Fiber<?> child2 = scope.schedule(task2);
}
```

Q & A

