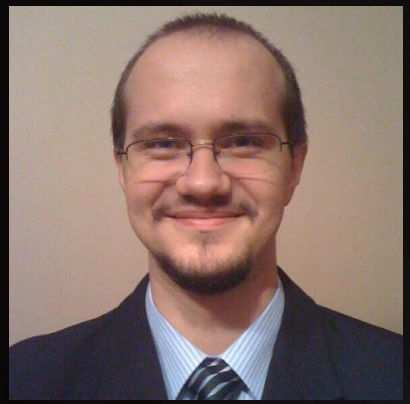
Demystifying Java Fibers

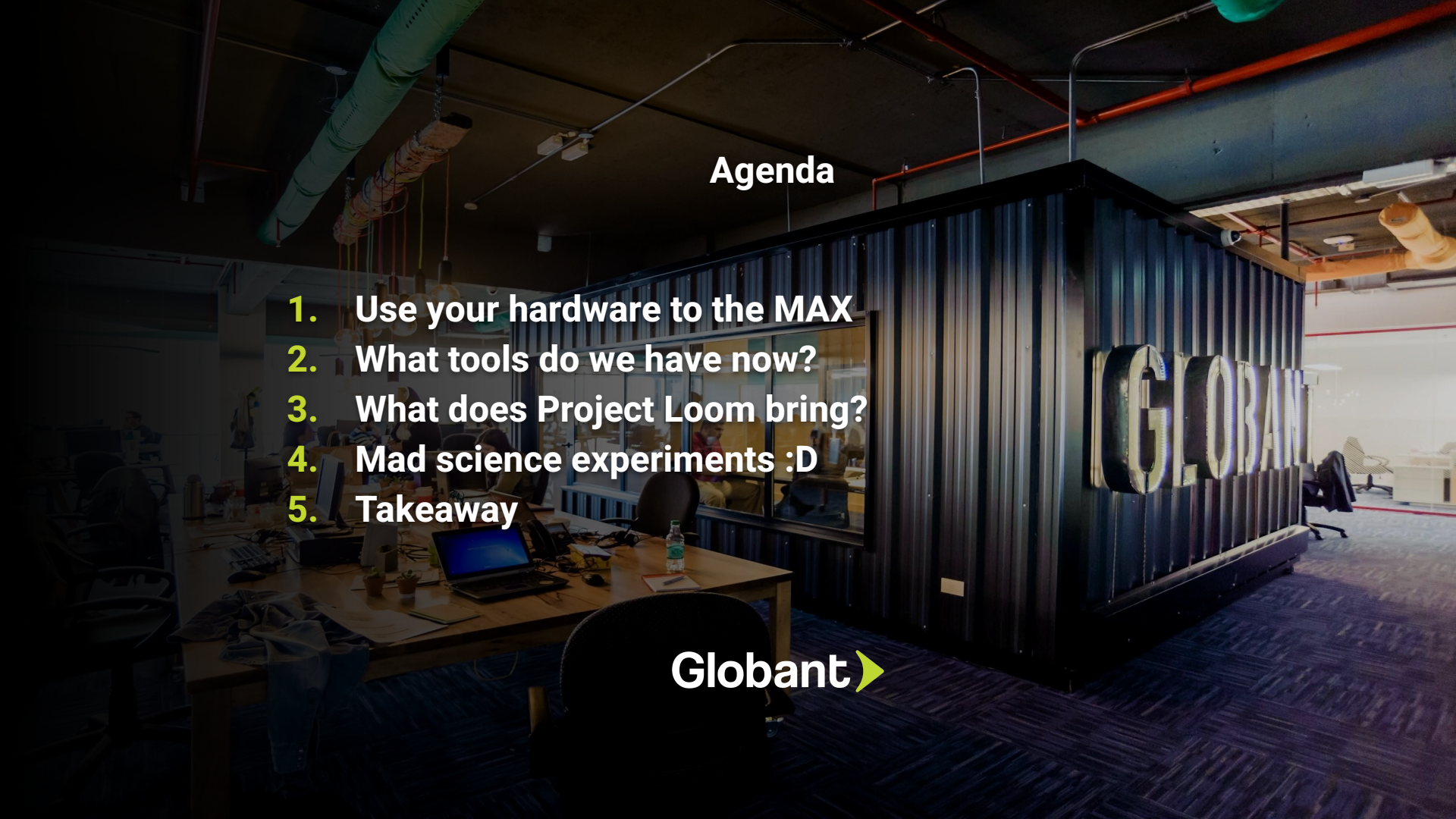
Who am I?
Globant TDC Cluj

Alex Dan Luca, Java architect in Globant Cluj



Andrei Draghia, Java architect in Globant Cluj

Globant



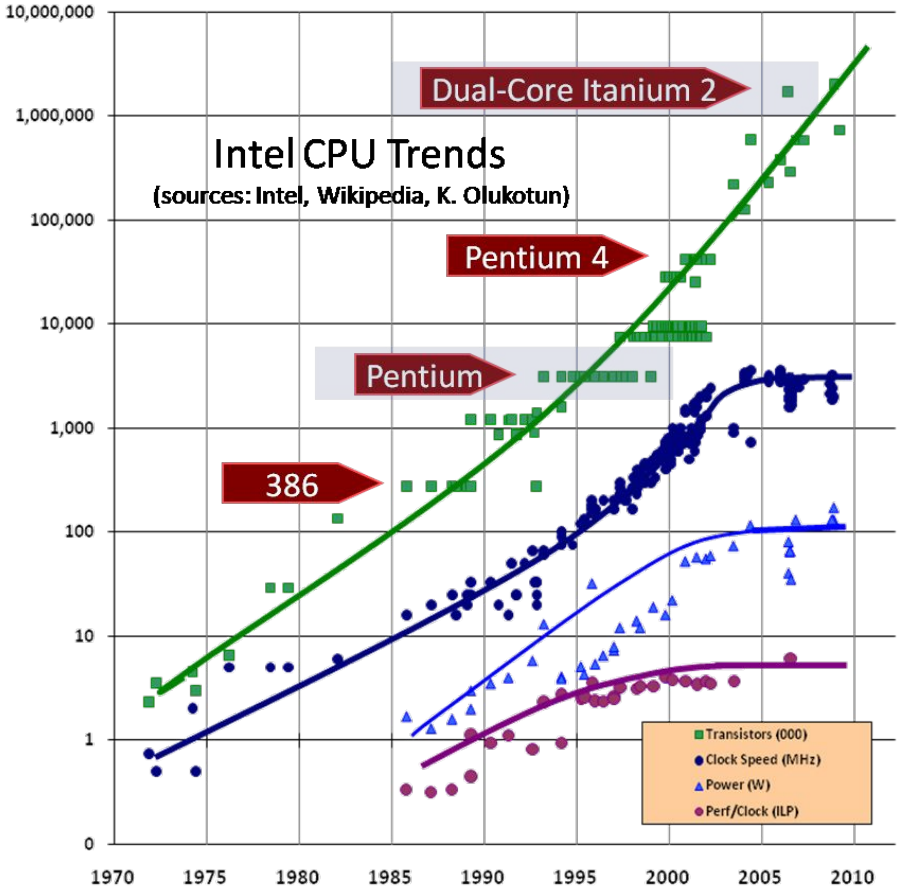
Agenda

1. Use your hardware to the MAX
2. What tools do we have now?
3. What does Project Loom bring?
4. Mad science experiments :D
5. Takeaway

Globant

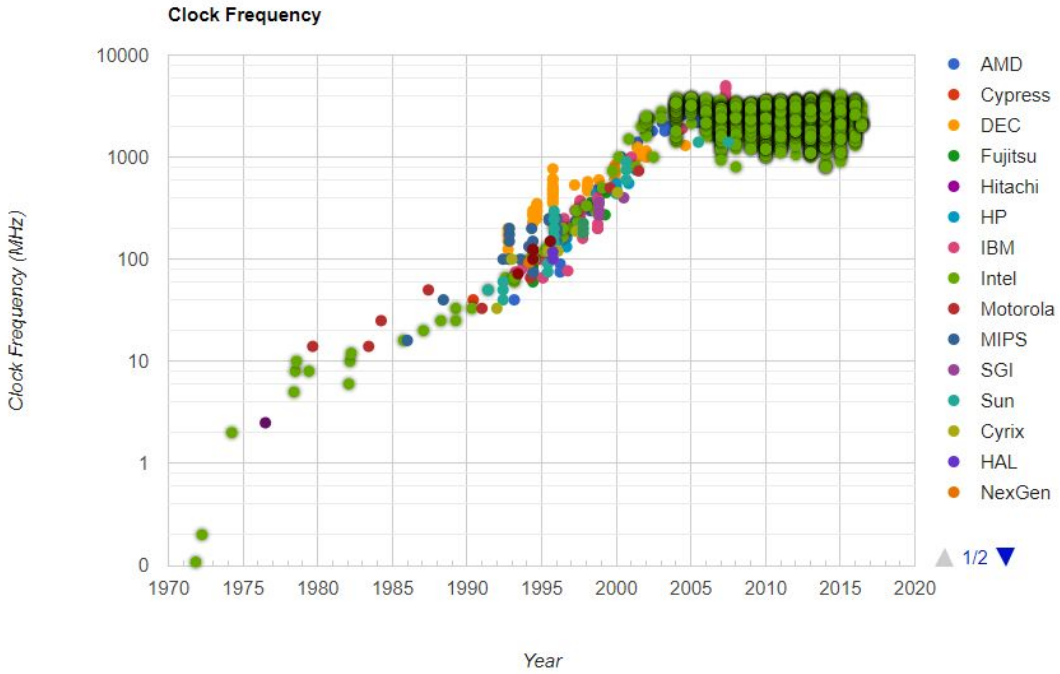

Use your hardware to the MAX
Free lunch?
Globant Development Center in NYC, USA

The free ‘performance’ lunch is over



Herb Sutter - 2005

Clock Frequency



Stanford - CPUDB

The free ‘performance’ lunch is over

Multi threading

I/O

Waste of resources

Advanced frameworks - require DEV training

What tools do we have now?

Globant Development Center in NYC, USA

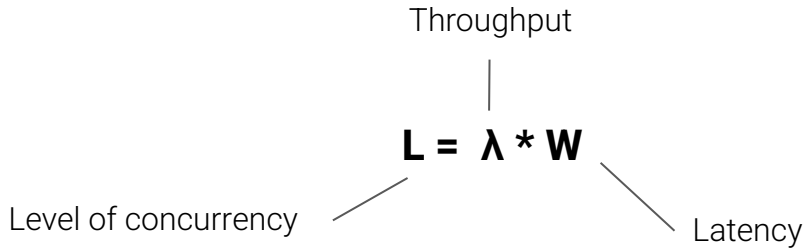
Concurrency v.s. Parallelism

Concurrency:

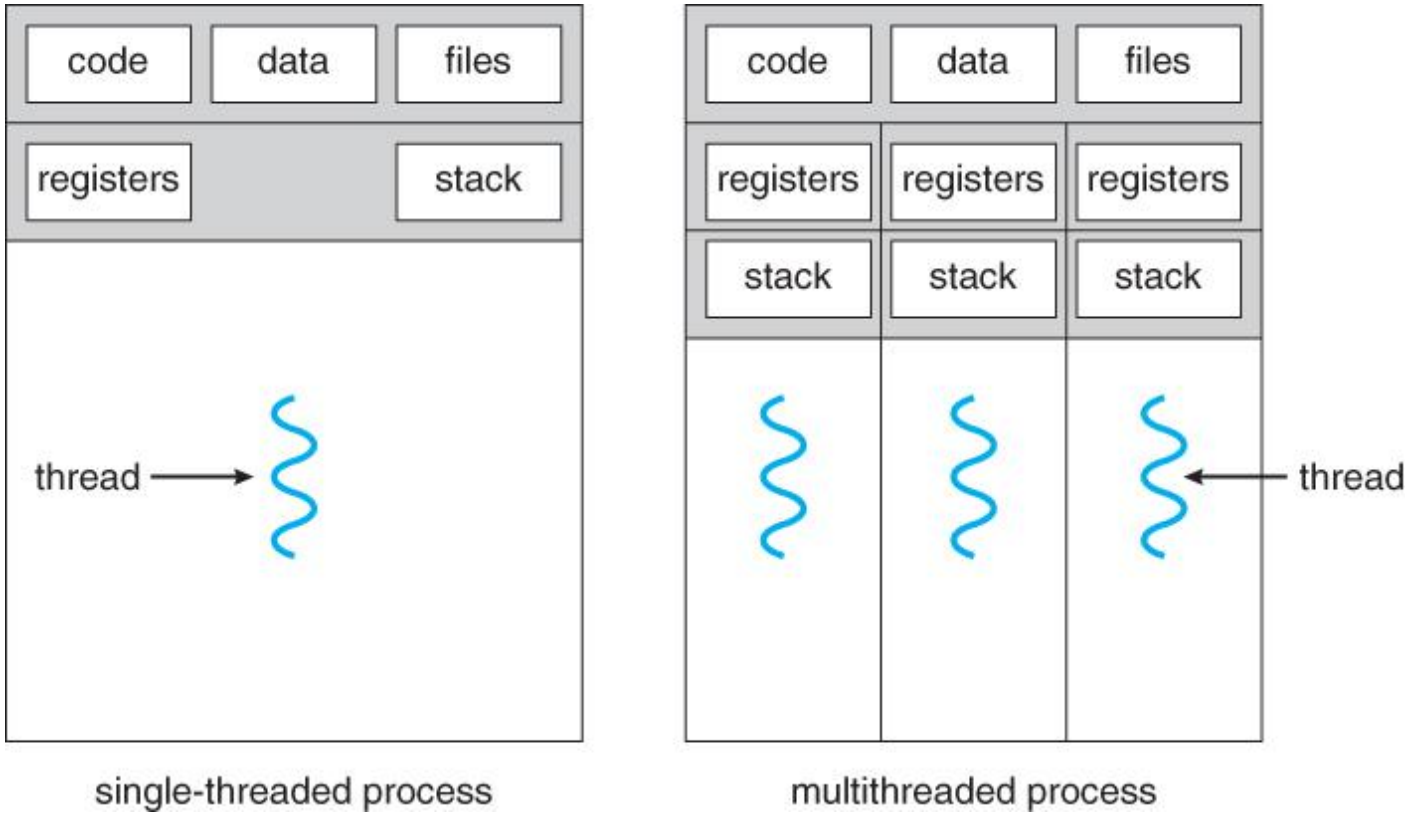
- Schedule multiple largely independent tasks to a set of computational resources
- Achieved by context switching
- Deals with a lot of things simultaneously
- Performance: *throughput* (tasks / time unit)

Parallelism:

- Speed up a task by splitting it to sub-tasks and exploiting multiple processing units
- Achieved by multiple CPUs
- Does a lot of things simultaneously
- Performance: *latency* (time unit)



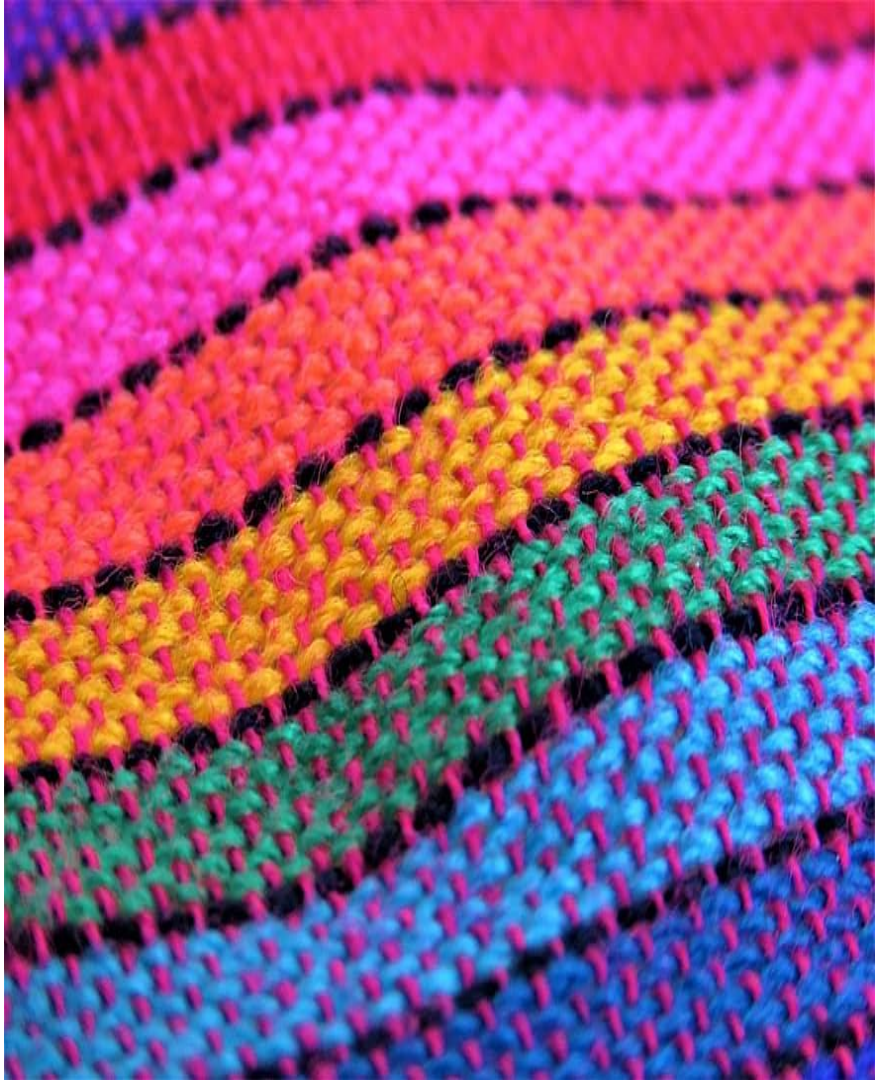
How do we achieve concurrency?



Threading models

- Single threaded - Sequential
- Multi-threaded - Thread pool
- Multi-threaded - Fork/Join
- Single threaded - Event Loop (e.g. Javascript)
- LMAX Disruptor
- Reactive programming
- Co-rutines - Async / Await (e.g. Python)
- User-land threads (e.g. Go)

* Not exhaustive



Single threaded - Sequential

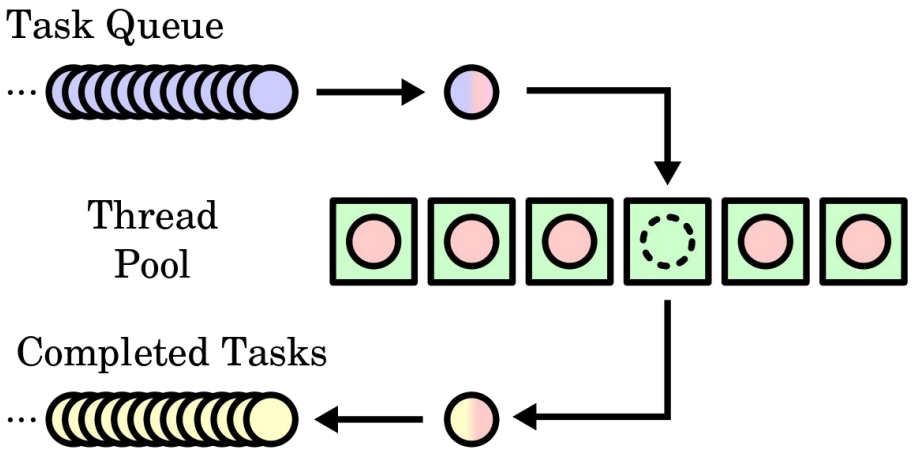
- How most of us learned to code
- How most languages were originally designed
- The code is executed in sequence, instruction by instruction
- No concurrent memory access
- Latency is the driving factor of the level of concurrency

Single threaded - Sequential

Model	Throughput	Parallelism	Ease of understanding	Development Experience	Error handling	Debugging	Profiling
Single threaded	1 task	None - low	Easy	Good	Easy	Easy	Easy
Thread pool							
Fork/join							
Event loop							
Reactive							
Async / await							
User-land threads							

Multi-threaded - **Thread Pool**

- Executing a set of tasks by a group of similar threads
- A task keeps the thread until it finished execution
- Increased complexity and problems caused by sharing (race conditions, thread local leaks etc.)
- Problematic cancellations
- Multiple types of thread pools: fixed, cached (elastic), scheduled, single threaded etc.
- Number of threads and efficient usage directly influences the *level of concurrency*.

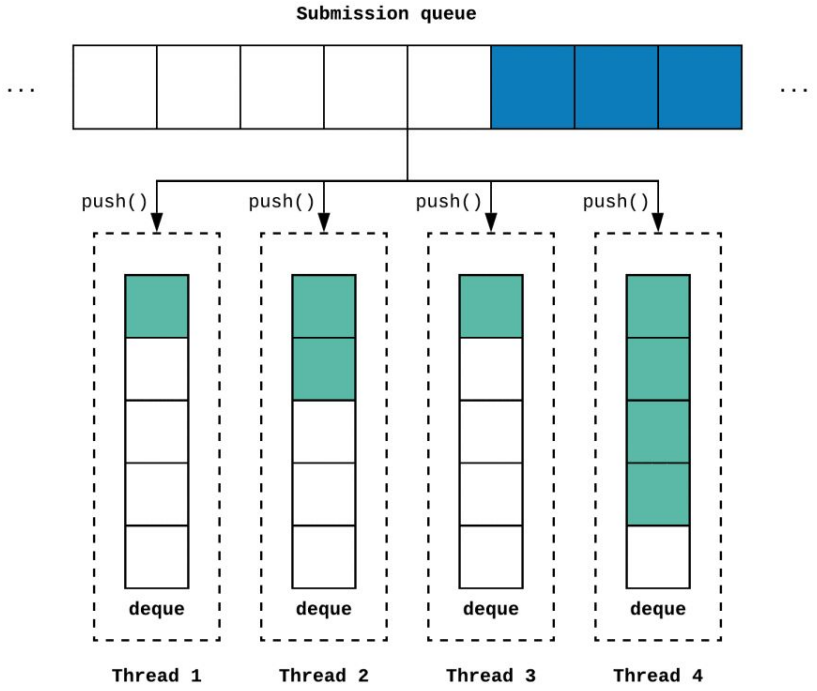


Multi-threaded - **Thread Pool**

Model	Throughput	Parallelism	Ease of understanding	Development Experience	Error handling	Debugging	Profiling
Single threaded	1 task	None - low	Easy	Good	Easy	Easy	Easy
Thread pool	Task/thread	Medium++	Easy	Good	Easy	Medium--	Medium--
Fork/join pool							
Event loop							
Reactive							
Async / await							
User-land threads							

Multi-threaded - Fork/Join

- Thread Pool model + divide and conquer
- Each thread in the pool has a Dequeue
- Implements a work-stealing algorithm
- Increases efficiency for high number of tasks
- Shares same challenges with thread pools
- Not friendly with blocking code

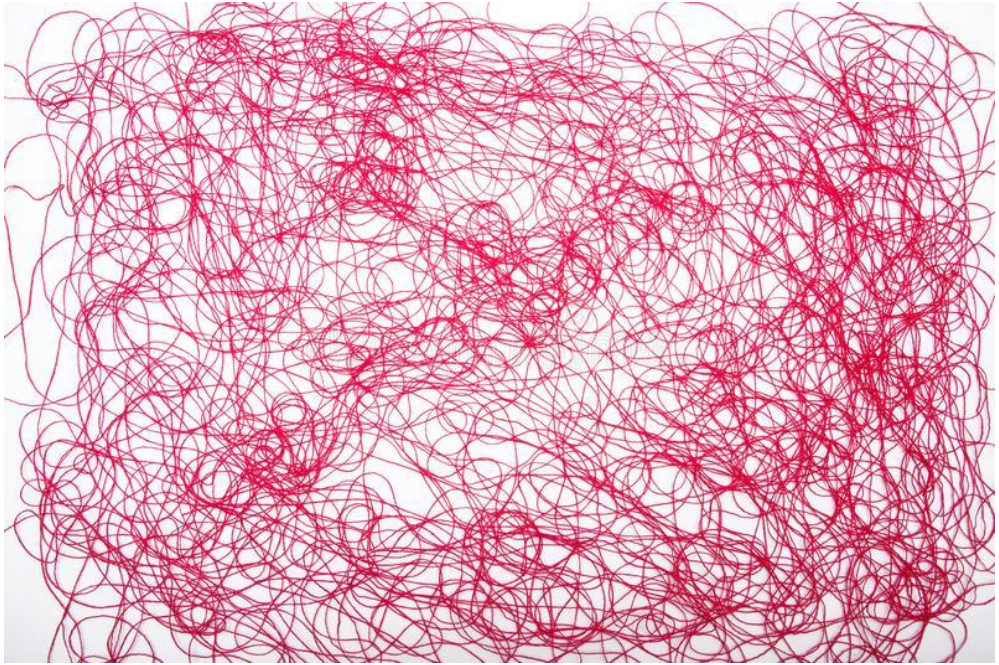


Multi-threaded - Fork/Join

Model	Throughput	Parallelism	Ease of understanding	Development Experience	Error handling	Debugging	Profiling
Single threaded	1 task	None - low	Easy	Good	Easy	Easy	Easy
Thread pool	Task/thread	Medium++	Easy	Good	Easy	Medium--	Medium--
Fork/join pool	Task/thread++	High*	Medium	Average	Medium--	Medium	Medium
Event loop							
Reactive							
Async / await							
User-land threads							

Single threaded - Event Loop

- Only 1 thread handles tasks from a queue in a while loop
- Makes use of callbacks and non-blocking I/O
- Avoids race conditions and general leaks
- Often incompatible APIs (in Java)
- Introduces new challenges:
  - Callback hell
  - Lost context
  - Unpredictable executions
- Cooperative concurrency model

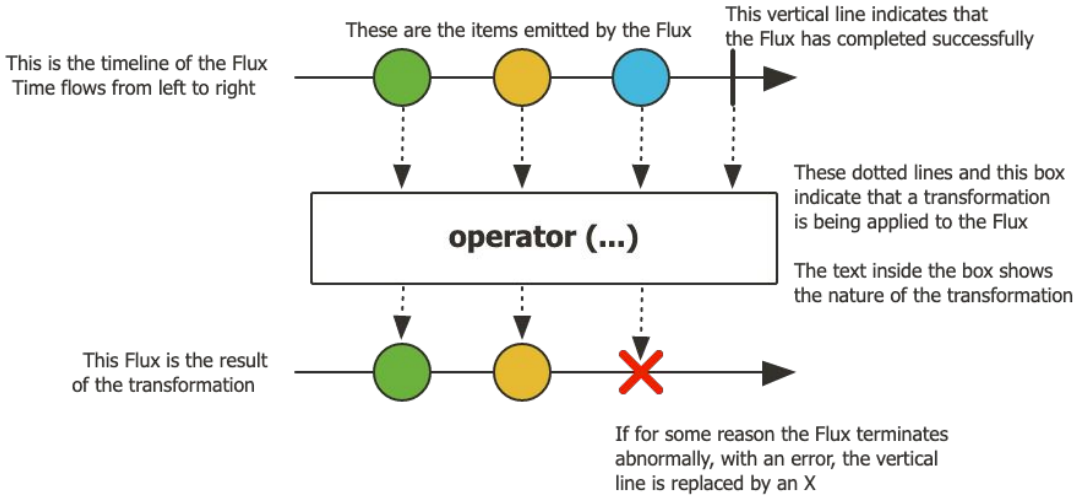


Single threaded - Event Loop

Model	Throughput	Parallelism	Ease of understanding	Development Experience	Error handling	Debugging	Profiling
Single threaded	1 task	None - low	Easy	Good	Easy	Easy	Easy
Thread pool	Task/thread	Medium++	Easy	Good	Easy	Medium--	Medium--
Fork/join pool	Task/thread++	High	Medium	Average	Medium--	Medium	Medium
Event loop	High	None - low	Medium	Average - Bad	Medium	Medium	Hard
Reactive							
Async / await							
User-land threads							

Reactive model

- Makes the push-based publisher - subscriber pattern a first level citizen
- Logic is encapsulated in steps of data stream processing
- Threads become abstracted by Schedulers (sort of thread pools)
- Concurrency mainly by publishOn / subscribeOn
- Tooling support gradually evolved



Reactive model

```
userService.getFavorites(userId, new Callback<List<String>>() { 1
  public void onSuccess(List<String> list) { 2
    if (list.isEmpty()) { 3
      suggestionService.getSuggestions(new Callback<List<Favorite>>() {
        public void onSuccess(List<Favorite> list) { 4
          UiUtils.submitOnUiThread(() -> { 5
            list.stream()
                .limit(5)
                .forEach(uiList::show); 6
            });
        }

        public void onError(Throwable error) { 7
          UiUtils.errorPopup(error);
        }
      });
    } else {
      list.stream() 8
          .limit(5)
          .forEach(favId -> favoriteService.getDetails(favId, 9
            new Callback<Favorite>() {
              public void onSuccess(Favorite details) {
                UiUtils.submitOnUiThread(() -> uiList.show(details));
              }

              public void onError(Throwable error) {
                UiUtils.errorPopup(error);
              }
            }
          ));
    }
  }

  public void onError(Throwable error) {
    UiUtils.errorPopup(error);
  }
});
```

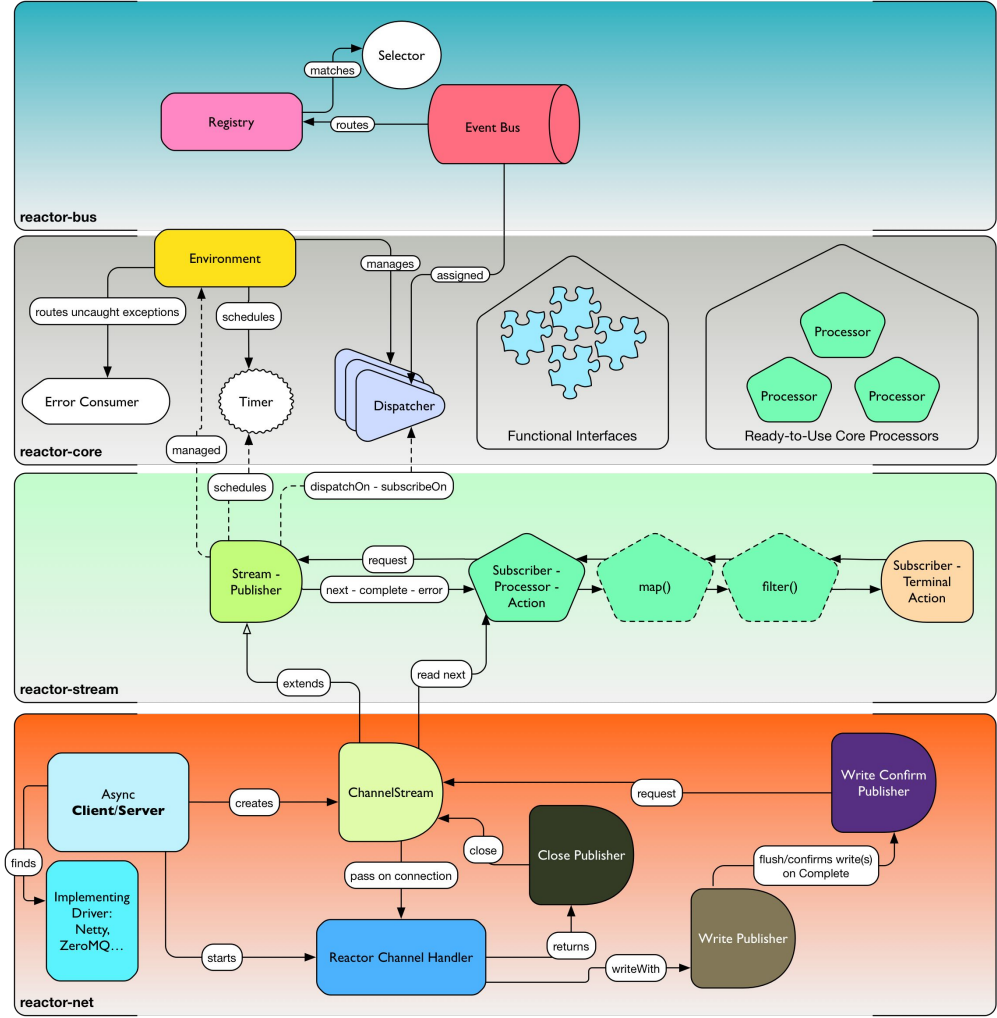
```
userService.getFavorites(userId)
           .timeout(Duration.ofMillis(800)) 1
           .onErrorResume(cacheService.cachedFavoritesFor(userId)) 2
           .flatMap(favoriteService::getDetails) 3
           .switchIfEmpty(suggestionService.getSuggestions())
           .take(5)
           .publishOn(UiUtils.uiThreadScheduler())
           .subscribe(uiList::show, UiUtils::errorPopup);
```

- Solves the callback hell
- Backpressure handling
- Complex logic becomes more concise
- Can signal end of stream
- Can re-create algorithms like fork-join, disruptor etc.

Reactive model

- Completely new API; low compatibility
- Steep learning curve
- Easy to do the wrong thing
- Can get quite complex
- Tooling support gradually evolved, but still not as good

```
java.lang.IndexOutOfBoundsException: Source emitted more than one item
        at reactor.core.publisher.MonoSingle$SingleSubscriber.onNext(MonoSingle.java:129)
        at
reactor.core.publisher.FluxOnAssembly$OnAssemblySubscriber.onNext(FluxOnAssembly.java:375)
(1)
...
(2)
...
        at reactor.core.publisher.Mono.subscribeWith(Mono.java:3204)
        at reactor.core.publisher.Mono.subscribe(Mono.java:3090)
        at reactor.core.publisher.Mono.subscribe(Mono.java:3057)
        at reactor.core.publisher.Mono.subscribe(Mono.java:3029)
        at reactor.guide.GuideTests.debuggingActivated(GuideTests.java:1000)
        Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException: (3)
Assembly trace from producer [reactor.core.publisher.MonoSingle] : (4)
        reactor.core.publisher.Flux.single(Flux.java:6676)
        reactor.guide.GuideTests.scatterAndGather(GuideTests.java:949)
        reactor.guide.GuideTests.populateDebug(GuideTests.java:962)
        org.junit.rules.TestWatcher$1.evaluate(TestWatcher.java:55)
        org.junit.rules.RunRules.evaluate(RunRules.java:20)
Error has been observed by the following operator(s): (5)
        |_      Flux.single ⇢ reactor.guide.GuideTests.scatterAndGather(GuideTests.java:949)
(6)
```



Reactive model

Model	Throughput	Parallelism	Ease of understanding	Development Experience	Error handling	Debugging	Profiling
Single threaded	1 task	None - low	Easy	Good	Easy	Easy	Easy
Thread pool	Task/thread	Medium++	Easy	Good	Easy	Medium--	Medium--
Fork/join pool	Task/thread++	High	Medium	Average	Medium--	Medium	Medium
Event loop	High	None - low	Medium	Average - Bad	Medium	Medium	Hard
Reactive	Very High	High	Hard	Good	Medium	Hard	Hard
Async / await							
User-land threads							

Coroutines - Async / Await

- Builds on top of the Event loop thread model
- Adds support for *explicit* means to transfer control to other coroutines - async / await
- Can create millions of coroutines; lightweight
- Generators, actor models, state machines
- Cooperative concurrency model

Corutines - Async / Await

Model	Throughput	Parallelism	Ease of understanding	Development Experience	Error handling	Debugging	Profiling
Single threaded	1 task	None - low	Easy	Good	Easy	Easy	Easy
Thread pool	Task/thread	Medium++	Easy	Good	Easy	Medium--	Medium--
Fork/join pool	Task/thread++	High	Medium	Average	Medium--	Medium	Medium
Event loop	High	None - low	Medium	Average - Bad	Medium	Medium	Hard
Reactive	Very High	High	Hard	Good	Medium	Hard	Hard
Async / await	High	None - low	Medium	Average	Easy	Medium	Medium
User-land threads							

User-land threads

- Coroutines multiplexed on threads a.k.a Green Threads or Fibers
- *Implicitly* yields control.
- Implementation relies a lot on the language specifics.
- Keeps a simpler, more classic programming model
- Re-introduces challenges appearing with shared resources
- Provides new ways to tackle them: channels, better race conditions detection

User-land threads

Model	Throughput	Parallelism	Ease of understanding	Development Experience	Error handling	Debugging	Profiling
Single threaded	1 task	None - low	Easy	Good	Easy	Easy	Easy
Thread pool	Task/thread	Medium++	Easy	Good	Easy	Medium--	Medium--
Fork/join pool	Task/thread++	High	Medium	Average	Medium--	Medium	Medium
Event loop	High	None - low	Medium	Average - Bad	Medium	Medium	Hard
Reactive	Very High	High	Hard	Good	Medium	Hard	Hard
Async / await	High	None - low	Medium	Average	Easy	Medium	Medium
User-land threads	Very High	High	Easy	Good	Easy	Medium	Medium

What does Project Loom bring?

Globant Development Center in NYC, USA

"I think [Project] Loom is going to kill reactive programming"

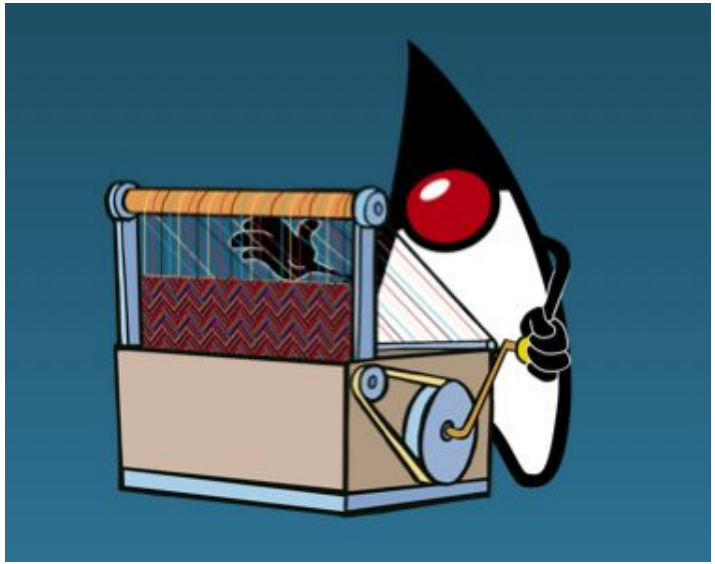
Brian Goetz
Java Language Architect at Oracle.
Author, Java Concurrency in Practice

What is Project Loom?

A. Teaching Java duke to weave carpets with the loom? NOPE

B. Preventing a looming danger regarding Java usage? MAYBE

C. Attempt to modernise concurrency in Java? APPROVED



Project Loom Goals

- Reduce the difficulty of writing efficient concurrent applications
- Eliminate the tradeoff between simplicity and efficiency in writing concurrent programs

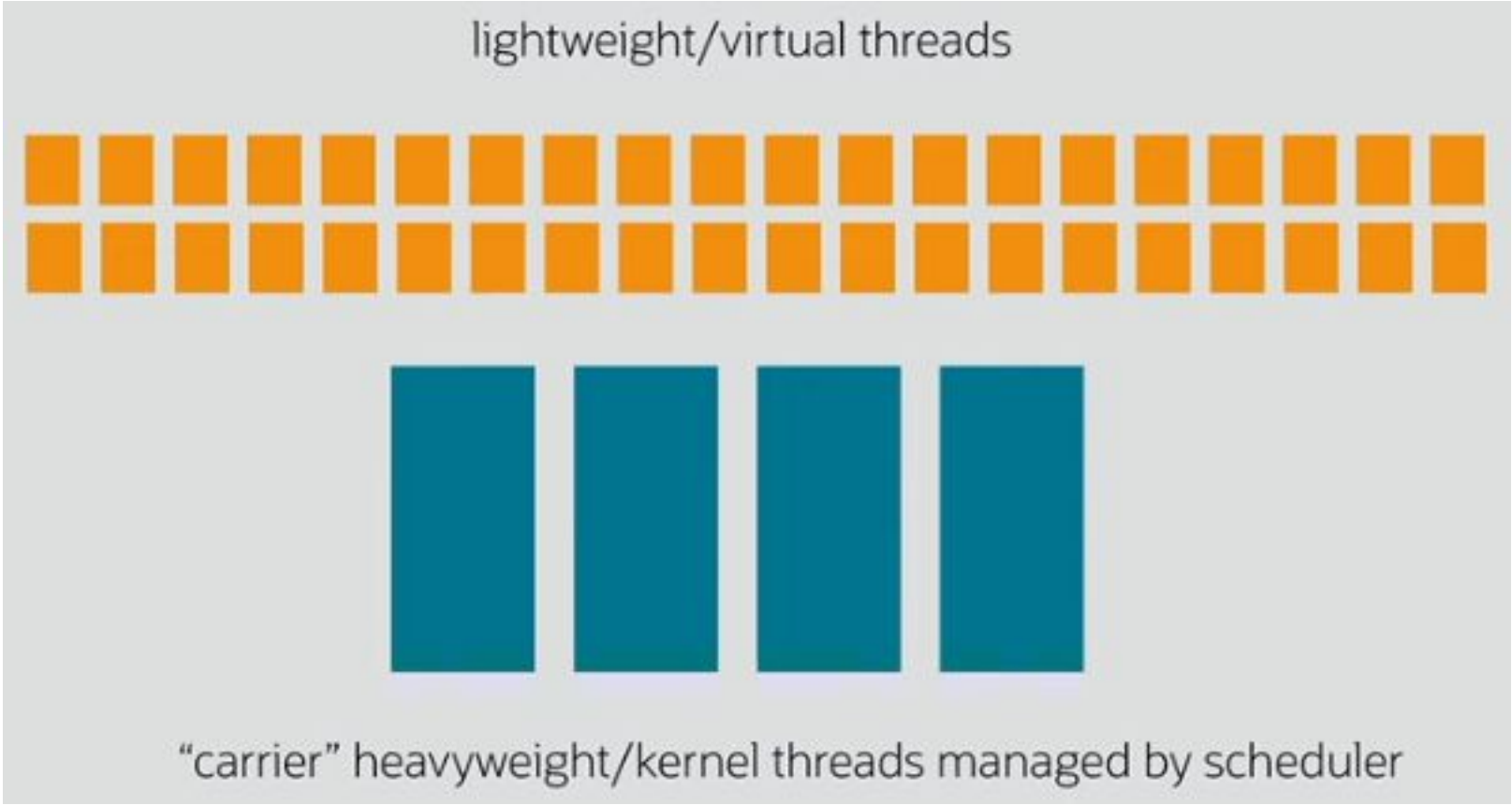
- Ensure easy adoption - leverage the existing APIs
- Thread.currentThread() and ThreadLocal - keep working

>2KB metadata >1MB of stack	200-300Kb metadata Pay-as-you-go stack
1-10µs	~200ns

Terminology

1. Virtual threads `Thread.startVirtualThread(Runnable)`
2. Continuations suspend, resume tasks - behind the scenes
3. Schedulers ForkJoinPool used underneath

Multiplexing virtual threads



Alan Bateman - Project Loom: Modern scalable concurrency for the Java platform

Operations that are virtual thread friendly

API	Method(s)	Notes
java.lang.Thread	sleep, join	join to wait for a virtual thread to terminate
java.lang.Process	waitFor	Linux/macOS only
java.util.concurrent	All blocking operations	
java.net.Socket	connect, read, write	Socket constructors with a host name parameter may need to do a lookup with InetAddress, see below
java.net.ServerSocket	accept	
java.net.DatagramSocket/MulticastSocket	receive	connect, disconnect and send do not block
java.nio.channels.SocketChannel	connect, read, write	
java.nio.channels.ServerSocketChannel	accept	
java.nio.channels.DatagramChannel	read, receive	connect, disconnect, send, and write do not block
java.nio.channels.Pipe.SourceChannel	read	
java.nio.channels.Pipe.SinkChannel	write	
Console streams (System.in, out, err)	read, write, printf	Linux/macOS only

Operations that pin the thread

API	Method(s)	Notes
java.lang.Object	wait	
java.lang.Process	waitFor	Windows only
java.io.File	All file I/O operations	
java.io.FileInputStream	open, read, skip	
java.io.FileOutputStream	open, write	
java.io.RandomAccessFile	open, read, write, seek	
java.net.InetAddress	All lookup operations	InetAddress SPI in the works that will allow deploying a virtual thread friendly name resolver
java.nio.MappedByteBuffer	force	
java.nio.channels.Selector	All blocking selection operations	
java.nio.channels.FileChannel	read, write, lock, truncate, force, transferTo	
java.nio.file	All file I/O operations	

Operations that pin the thread

In general, operations that get an object monitor, pin the carrier thread
synchronized + Virtual threads = not love

API	Method(s)	Notes
java.lang.Thread	join	join to wait for a kernel thread to terminate
java.lang.Process	All operations on the input/output/error streams	
Console streams (System.in, out, err)	read, write, printf	Windows only
java.io.Console	All read, format, printf operations	

Virtual threads - Troubleshooting

Detecting pinning:

- -Djdk.tracePinnedThreads(=full|short)
- Prints stack trace of virtual thread when parking pins it’s carrier thread

- Full = prints complete stack trace
- Short = includes only frames with problematic code

Thread dumps:

- Doesn’t work by default
- Tooling currently in development
- Some prototypes available that will include all virtual threads started with an Executor

* May evolve until official release

Virtual threads - Debugger support

Debugger changes:

- You can’t view all the virtual threads in the debugger when breaking
- Not supported: stop, interrupt, popFrame, forceEarlyReturn, setValue

Virtual threads + breakpoints:

- Hitting a breakpoint pins the virtual thread to carrier
- Single stepping works until virtual thread yields
- *Recommendation:* suspend all threads when breakpoint is hit and resume all threads when stepping

* May evolve until official release


Mad science experiments :D
Globant Development Center in NYC, USA

How stuff works in code

- virtual threads are java entities, independent of OS threads
- **java.lang.Thread** is used for both kinds of threads, virtual and OS
- virtual threads require carrier threads - OS Threads - to run on
    a. a carrier thread runs a virtual thread by mounting it
    b. if the VT blocks, the stack is stored and the VT is unmounted to be resumed later
- millions of virtual threads can run on few carrier threads

Hitchhiking

Hello World

```
public static void main(String[] args) {
    Thread.startVirtualThread(() -> {
        System.out.println("Hello Loom from "+Thread.currentThread()+"!");
    });

    System.out.println("Hello World from "+Thread.currentThread()+"!");

}
```

C:\Program Files\Java\jdk-17\bin>java Test.java

Hello Loom from VirtualThread[#15,ForkJoinPool-1-worker-1,CarrierThreads]!

Hello World from Thread[main,5,main]!

-Djdk.defaultScheduler.parallelism=N,

```java
public static void main(String[] args) throws InterruptedException {

    final boolean USE_VIRTUAL_THREADS = false;
    final int CARRIER_THREAD_COUNT = 1;
    final int TASK_COUNT = 2;

    // plain old thread factory and thread pool using the new builder
    ThreadFactory carrierTF = Thread.ofPlatform().name("carrier#", 0).daemon(true).factory();
    ExecutorService carrierPool = Executors.newFixedThreadPool( CARRIER_THREAD_COUNT, carrierTF);
    ExecutorService executor;

    if(USE_VIRTUAL_THREADS) {

        // factory for virtual threads scheduled on the carrier pool
        ThreadFactory virtualTF = Thread.ofVirtual()
                .scheduler(carrierPool)
                .name("virtual#", 0).factory();

        // thread executor will spawn a new virtual thread for each task
        executor = Executors.newThreadExecutor(virtualTF);

    } else {
        executor = carrierPool;
    }

    for (int i = 0; i < TASK_COUNT; i++)
        executor.submit(new WaitAndHurry(i));

    executor.shutdown();
    executor.awaitTermination(20, TimeUnit.SECONDS);
}
```

```java
private final static class WaitAndHurry implements Runnable {

    private final static long START_TIME = System.nanoTime();

    private Integer index = 0;

    WaitAndHurry(Integer index) {
        this.index = index;
    }

    @Override
    public void run() {
        doIO();    // block for 2s
        doWork();  // compute something for ~2s
        print("done");
    }

    private void doIO() {
        print("io");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ex) {
            throw new RuntimeException(ex);
        }
    }

    private void doWork() {
        print("work");
        long number = 479001599;
        boolean prime = true;
        for(long i = 2; i <= number/2; ++i) {
            if(number % i == 0)  {
                prime = false;
                break;
            }
        }
        if (!prime) {throw new RuntimeException("wrong result");} // to prevent the JIT to optimize everything away
    }

    private void print(String msg) {
        double elapsed = (System.nanoTime()-START_TIME)/1_000_000_000.0d;
        String timestamp = String.format("% 2fs", elapsed);
```

Regular 1 Thread Pool

```
final boolean USE_VIRTUAL_THREADS = false;
final int CARRIER_THREAD_COUNT = 1;
final int TASK_COUNT = 2;
```

```
c:\Program Files\Java\jdk-17\bin>java TestPrez.java
Task 0: 0.00s Thread[carrier#0,5,main] io
Task 0: 2.02s Thread[carrier#0,5,main] work
Task 0: 3.85s Thread[carrier#0,5,main] done
Task 1: 3.85s Thread[carrier#0,5,main] io
Task 1: 5.86s Thread[carrier#0,5,main] work
Task 1: 7.69s Thread[carrier#0,5,main] done
```

7.69 s

	2s	1.85s	2s	1.85s
CPU:	IDLE	WORK	IDLE	WORK
Carrier0:	Task 0 - WAIT	Task 0 - WORK	Task 1 - WAIT	Task 1 - WORK

Virtual Thread Pool with 1 Thread Carrier Pool

```
final boolean USE_VIRTUAL_THREADS = true;
final int CARRIER_THREAD_COUNT = 1;
final int TASK_COUNT = 2;
```

```
c:\Program Files\Java\jdk-17\bin>java TestPrez.java
Task 0: 0.02s VirtualThread[virtual#0,carrier#0,main] io
Task 1: 0.05s VirtualThread[virtual#1,carrier#0,main] io
Task 1: 2.05s VirtualThread[virtual#1,carrier#0,main] work
Task 1: 3.87s VirtualThread[virtual#1,carrier#0,main] done
Task 0: 3.87s VirtualThread[virtual#0,carrier#0,main] work
Task 0: 5.72s VirtualThread[virtual#0,carrier#0,main] done
```

5.72 s

	2s	1.85s	1.85s
CPU:	IDLE	WORK	WORK
Carrier0:	IDLE	Task 1 - WORK	Task 0 - WORK
Virtual0:	Task 0 - WAIT		Task 0 - WORK
Virtual1:	Task 1 - WAIT	Task 1 - WORK	

Regular 1 Thread Pool

```
final boolean USE_VIRTUAL_THREADS = false;
final int CARRIER_THREAD_COUNT = 2;
final int TASK_COUNT = 6;
```

```
c:\Program Files\Java\jdk-17\bin>java TestPrez.java
Task 1: 0.01s Thread[carrier#1,5,main] io
Task 0: 0.00s Thread[carrier#0,5,main] io
Task 1: 2.02s Thread[carrier#1,5,main] work
Task 0: 2.02s Thread[carrier#0,5,main] work
Task 1: 3.86s Thread[carrier#1,5,main] done
Task 2: 3.86s Thread[carrier#1,5,main] io
Task 0: 3.87s Thread[carrier#0,5,main] done
Task 3: 3.87s Thread[carrier#0,5,main] io
Task 2: 5.87s Thread[carrier#1,5,main] work
Task 3: 5.88s Thread[carrier#0,5,main] work
Task 3: 7.58s Thread[carrier#0,5,main] done
Task 4: 7.58s Thread[carrier#0,5,main] io
Task 2: 7.71s Thread[carrier#1,5,main] done
Task 5: 7.71s Thread[carrier#1,5,main] io
Task 4: 9.60s Thread[carrier#0,5,main] work
Task 5: 9.71s Thread[carrier#1,5,main] work
Task 4: 11.32s Thread[carrier#0,5,main] done
Task 5: 11.44s Thread[carrier#1,5,main] done
```

11.44 s

Regular 1 Thread Pool

```
final boolean USE_VIRTUAL_THREADS = true;
final int CARRIER_THREAD_COUNT = 2;
final int TASK_COUNT = 6;
```

```
c:\Program Files\Java\jdk-17\bin>java TestPrez.java
Task 1: 0.02s VirtualThread[virtual#1,carrier#1,main] io
Task 0: 0.02s VirtualThread[virtual#0,carrier#0,main] io
Task 2: 0.04s VirtualThread[virtual#2,carrier#0,main] io
Task 3: 0.04s VirtualThread[virtual#3,carrier#0,main] io
Task 4: 0.04s VirtualThread[virtual#4,carrier#0,main] io
Task 5: 0.04s VirtualThread[virtual#5,carrier#0,main] io
Task 0: 2.05s VirtualThread[virtual#0,carrier#0,main] work
Task 4: 2.05s VirtualThread[virtual#4,carrier#1,main] work
Task 0: 3.89s VirtualThread[virtual#0,carrier#0,main] done
Task 5: 3.89s VirtualThread[virtual#5,carrier#0,main] work
Task 4: 3.92s VirtualThread[virtual#4,carrier#1,main] done
Task 3: 3.92s VirtualThread[virtual#3,carrier#1,main] work
Task 3: 5.67s VirtualThread[virtual#3,carrier#1,main] done
Task 2: 2.05s VirtualThread[virtual#2,carrier#1,main] work
Task 5: 5.79s VirtualThread[virtual#5,carrier#0,main] done
Task 1: 2.05s VirtualThread[virtual#1,carrier#1,main] work
Task 2: 7.52s VirtualThread[virtual#2,carrier#1,main] done
Task 1: 7.66s VirtualThread[virtual#1,carrier#0,main] done
```

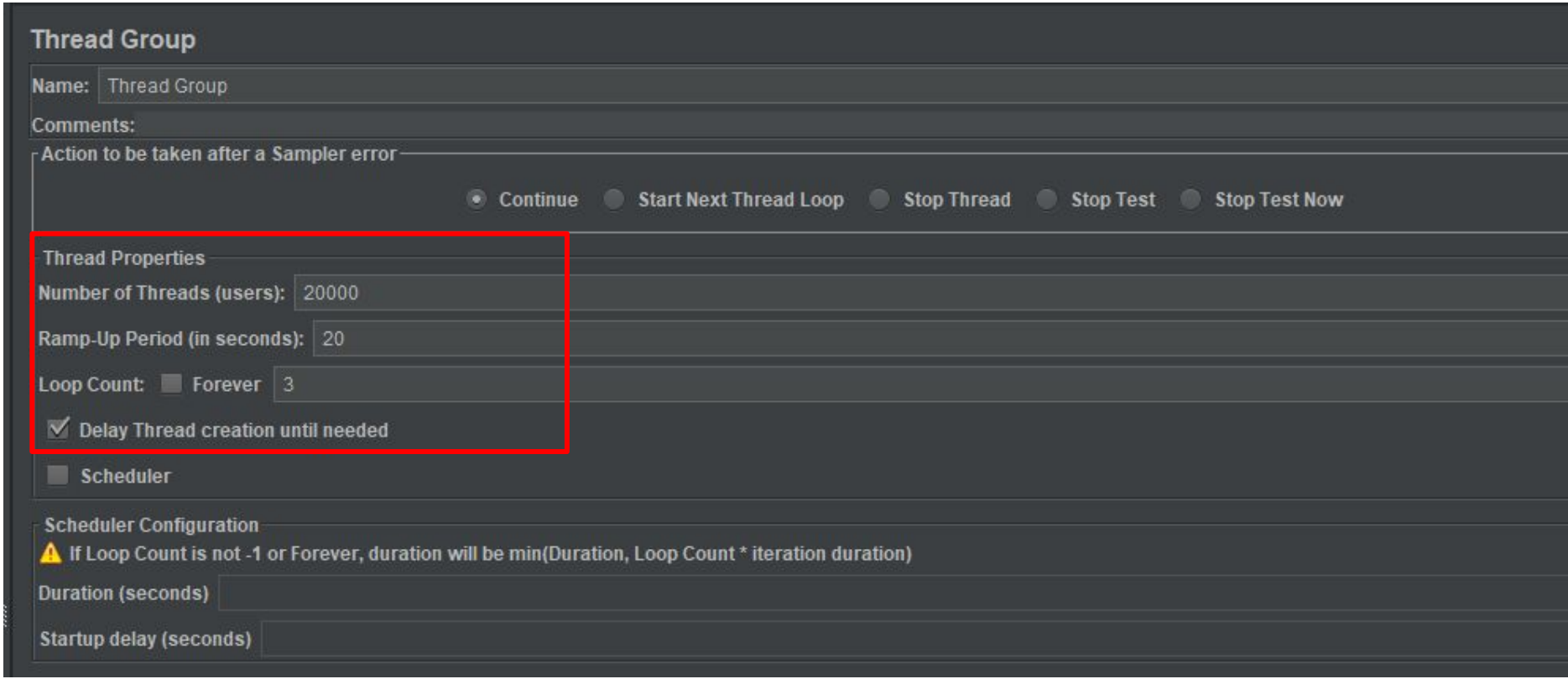
7.66 s

Simple HTTP Server + JMeter test

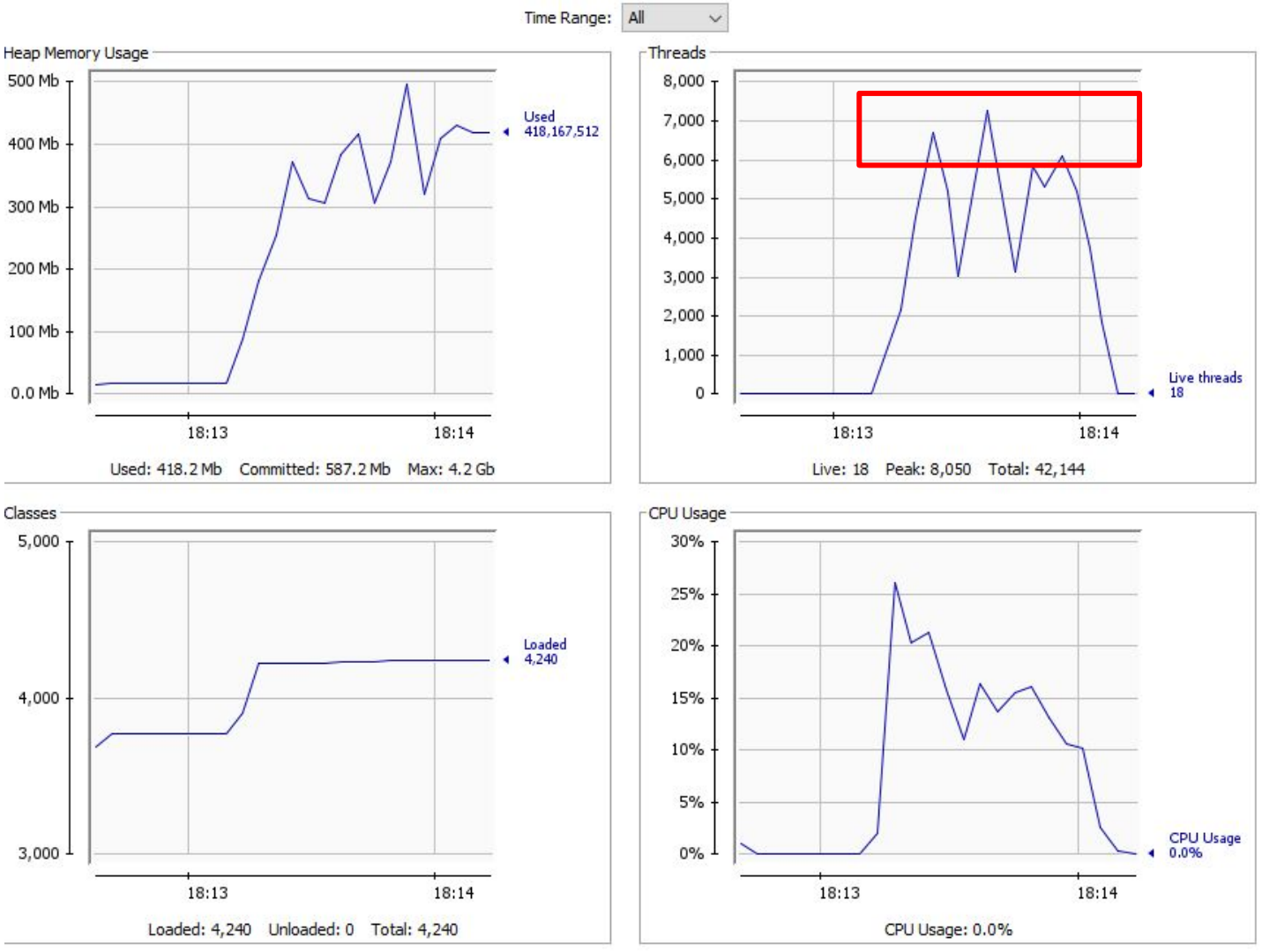
```java
public class Test {

    public static Double count = 0.0;


    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress(8500), 0);
        HttpContext context = server.createContext("/test");
        context.setHandler(Test::handleRequest);
        server.start();
        System.out.println("Server started on port 8500");
    }

    private static void handleRequest(HttpExchange exchange) throws IOException {
        new Thread(new Runnable() {
            @Override
            public void run() {
                String responseString = "Request count: "+Test.count++;
                exchange.getResponseHeaders().add("Access-Control-Allow-Origin", "*");
                exchange.getResponseHeaders().add("Access-Control-Allow-Headers","origin, content-type, ac
                exchange.getResponseHeaders().add("Access-Control-Allow-Credentials", "true");
                exchange.getResponseHeaders().add("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE,
                exchange.getResponseHeaders().set("Content-Type", "application/json; charset=UTF-8");
                try {
                    Thread.sleep(6000);
                    exchange.sendResponseHeaders(200, responseString.getBytes().length);
                    OutputStream os = exchange.getResponseBody();
                    os.write(responseString.getBytes());
                    os.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();

    }
}
```

```java
public class Test {

    public static Double count = 0.0;


    public static void main(String[] args) th
        HttpServer server = HttpServer.create
        HttpContext context = server.createCo
        context.setHandler(Test::handleReques
        server.start();
        System.out.println("Server LOOM start
    }

    private static void handleRequest(HttpExc
        Thread.startVirtualThread(new Runnabl
            @Override
            public void run() {
                String responseString = "Requ
                exchange.getResponseHeaders()
                exchange.getResponseHeaders()
                exchange.getResponseHeaders()
                exchange.getResponseHeaders()
                exchange.getResponseHeaders()
                try {
                    Thread.sleep(6000);
                    exchange.sendResponseHeac
                    OutputStream os = exchang
                    os.write(responseString.g
                    os.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });

    }
}
```

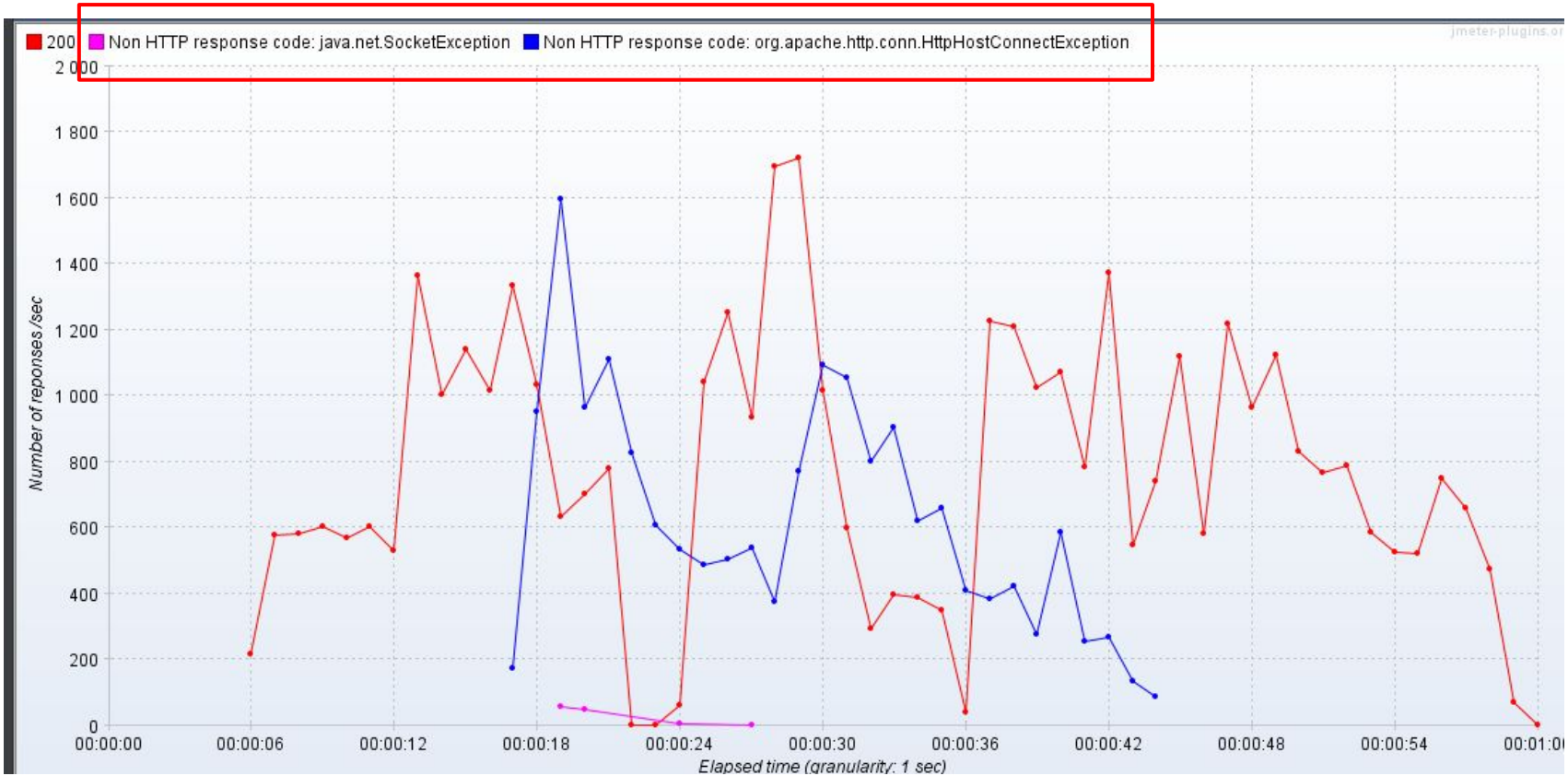
JMeter test settings



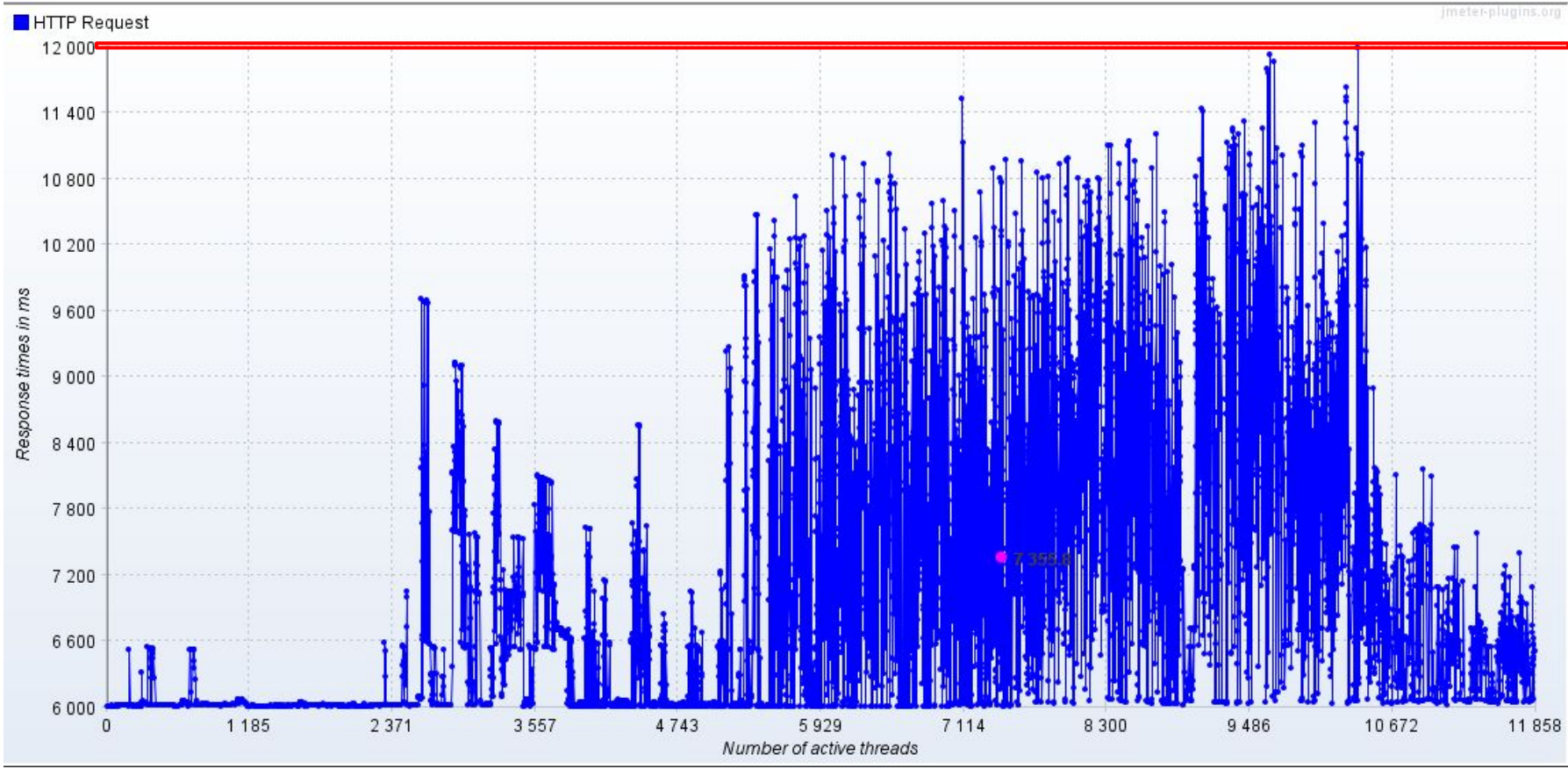
Dumb Threads
JConsole



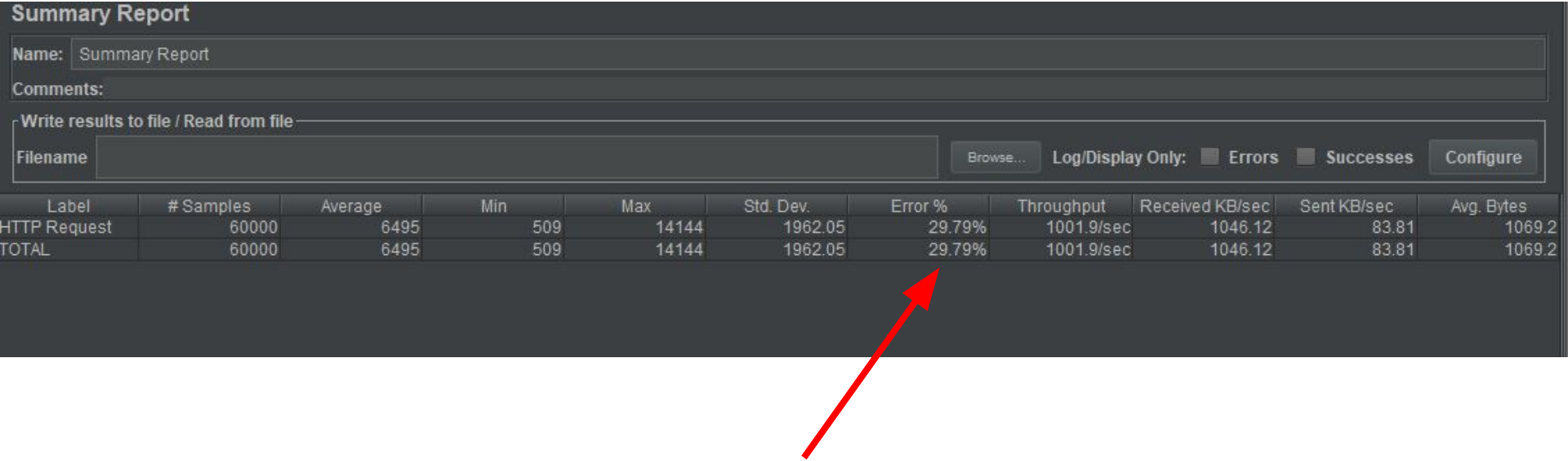
Dumb Threads - response codes



Dumb Threads - response time per thread count

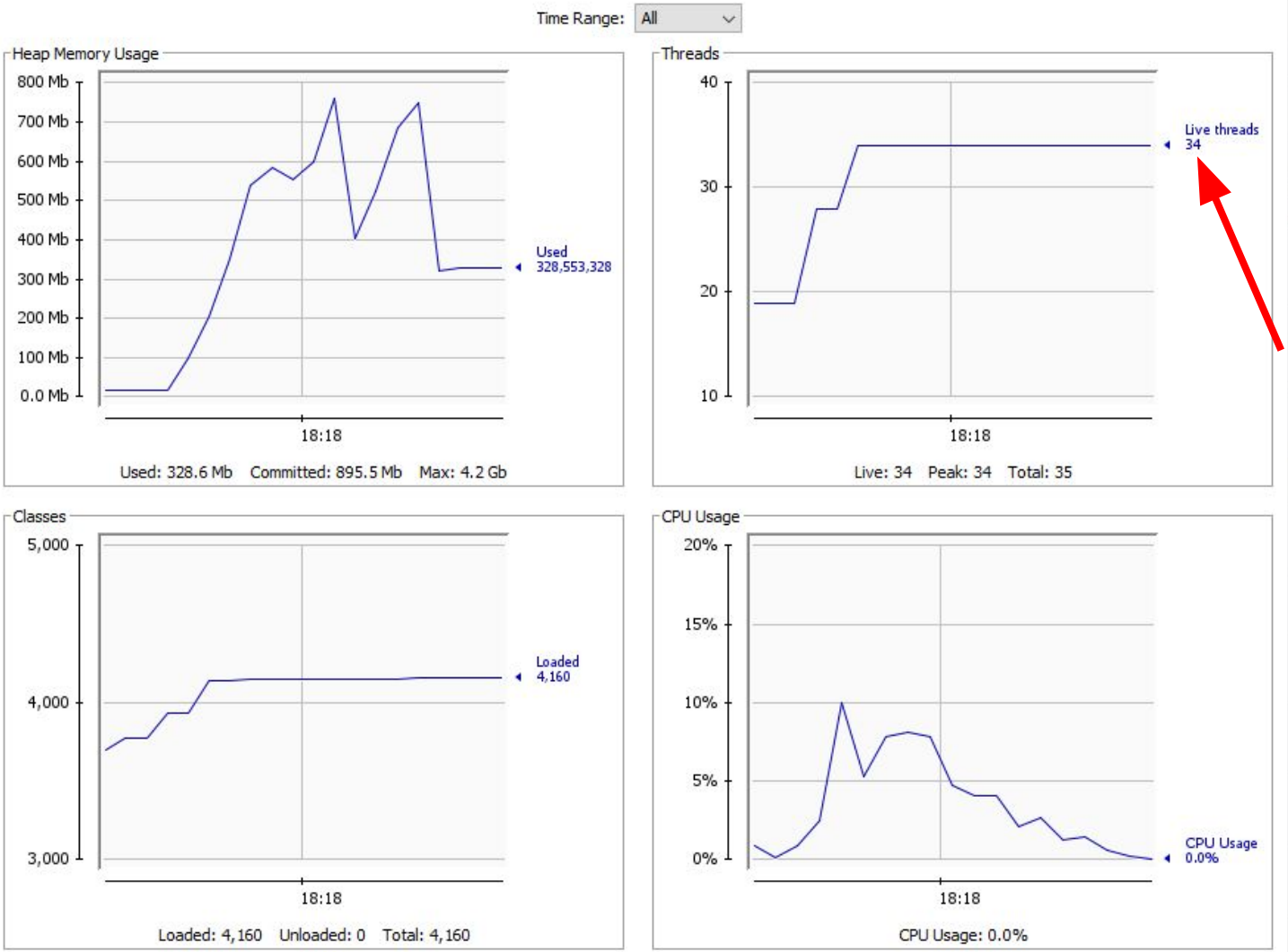


Dumb Threads - summary - 29% errors



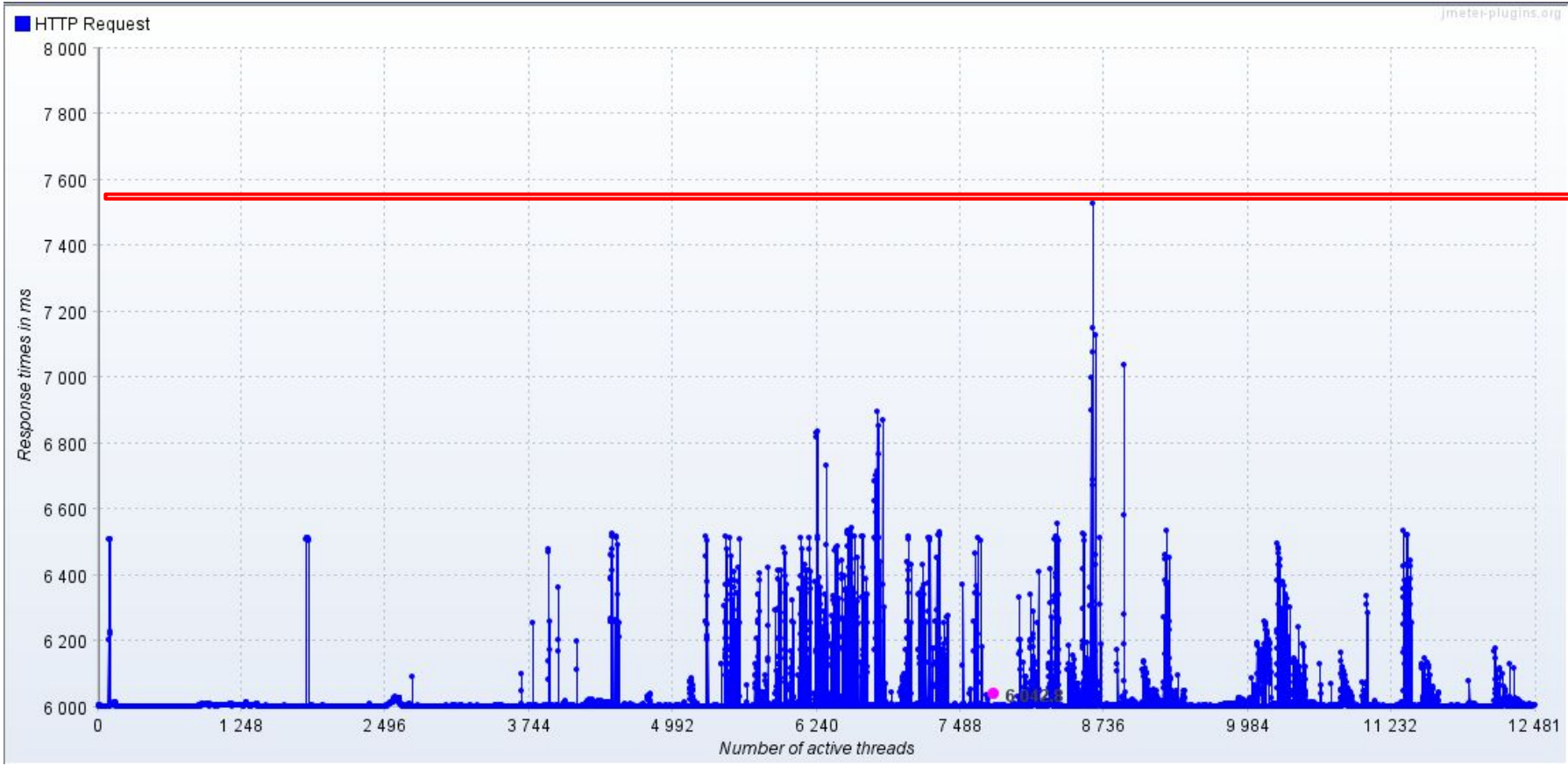
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	60000	6495	509	14144	1962.05	29.79%	1001.9/sec	1046.12	83.81	1069.2
TOTAL	60000	6495	509	14144	1962.05	29.79%	1001.9/sec	1046.12	83.81	1069.2

Virtual Threads
JConsole

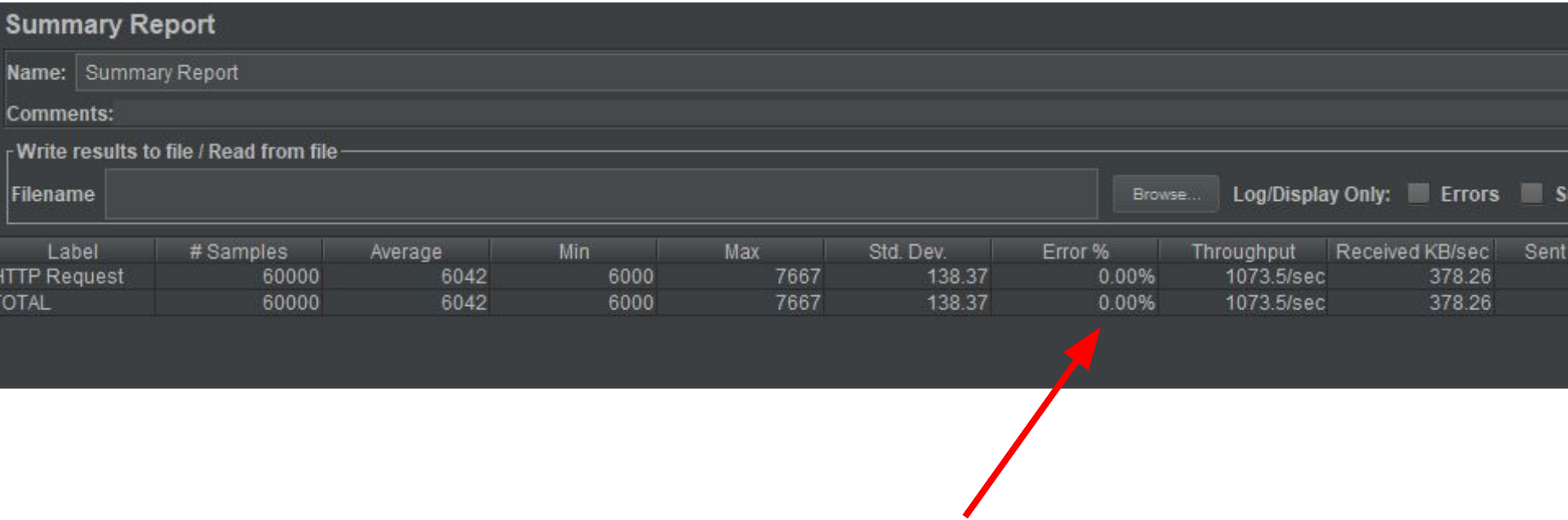


Virtual Threads
JConsole

Virtual Threads
JConsole



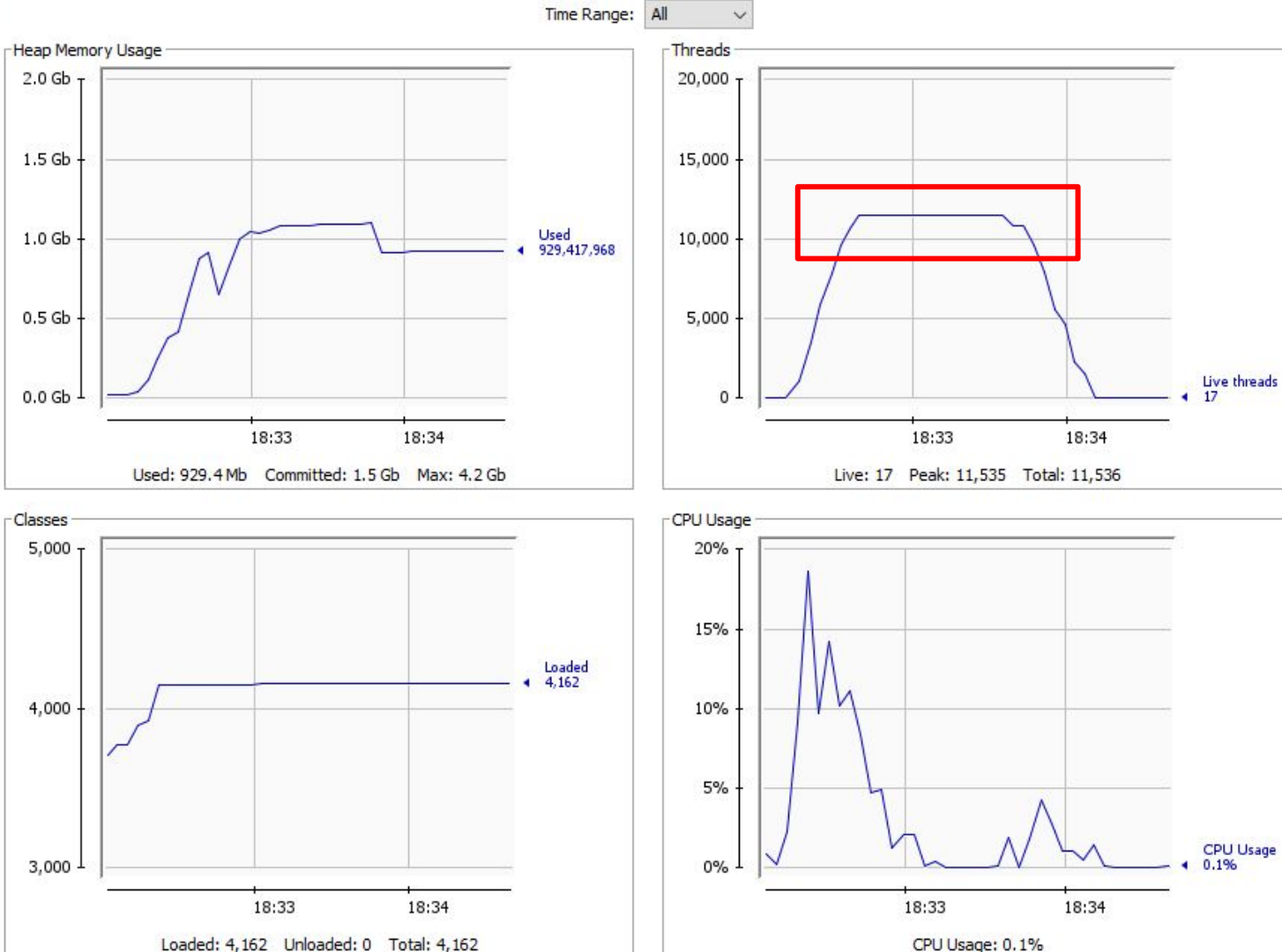
Virtual Threads
JConsole



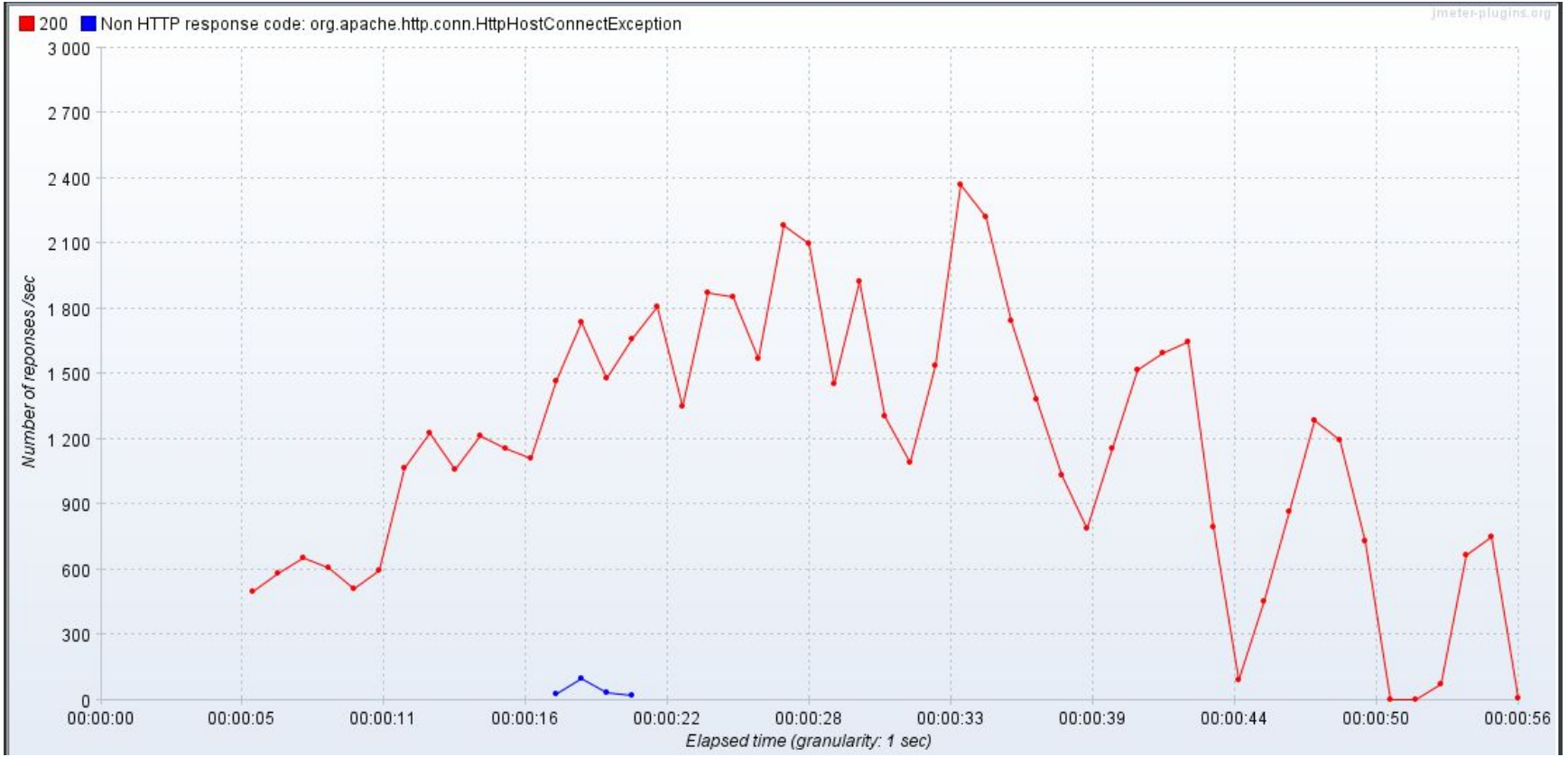
Smart Threads - Executor Code

```java
public class Test {

    public static Double count = 0.0;

    private static ExecutorService executorService = Executors.newCachedThreadPool();

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress(8500), 0);
        HttpContext context = server.createContext("/test");
        context.setHandler(Test::handleRequest);
        server.start();
        System.out.println("Server started on port 8500");
    }

    private static void handleRequest(HttpExchange exchange) throws IOException {
        executorService.execute(new Runnable() {
            @Override
            public void run() {
                String responseString = "Request count: "+Test.count++;
                exchange.getResponseHeaders().add("Access-Control-Allow-Origin", "*");
                exchange.getResponseHeaders().add("Access-Control-Allow-Headers","origin, content-type, a
                exchange.getResponseHeaders().add("Access-Control-Allow-Credentials", "true");
                exchange.getResponseHeaders().add("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE
                exchange.getResponseHeaders().set("Content-Type", "application/json; charset=UTF-8");
                try {
                    Thread.sleep(6000);
                    exchange.sendResponseHeaders(200, responseString.getBytes().length);
                    OutputStream os = exchange.getResponseBody();
                    os.write(responseString.getBytes());
                    os.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });

    }
}
```

Smart Threads - Executor JConsole



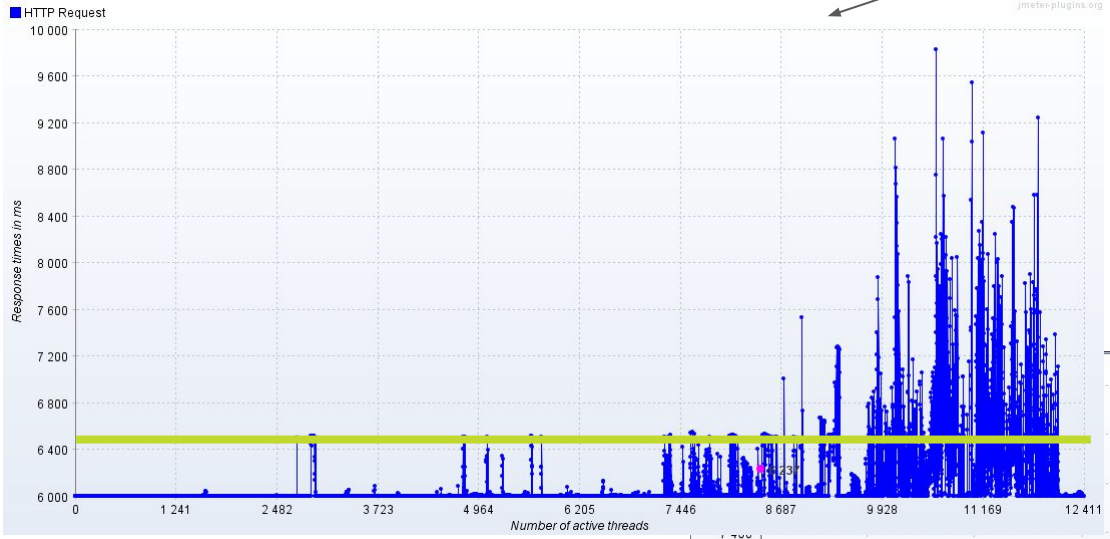
Smart Threads - Executor
JConsole



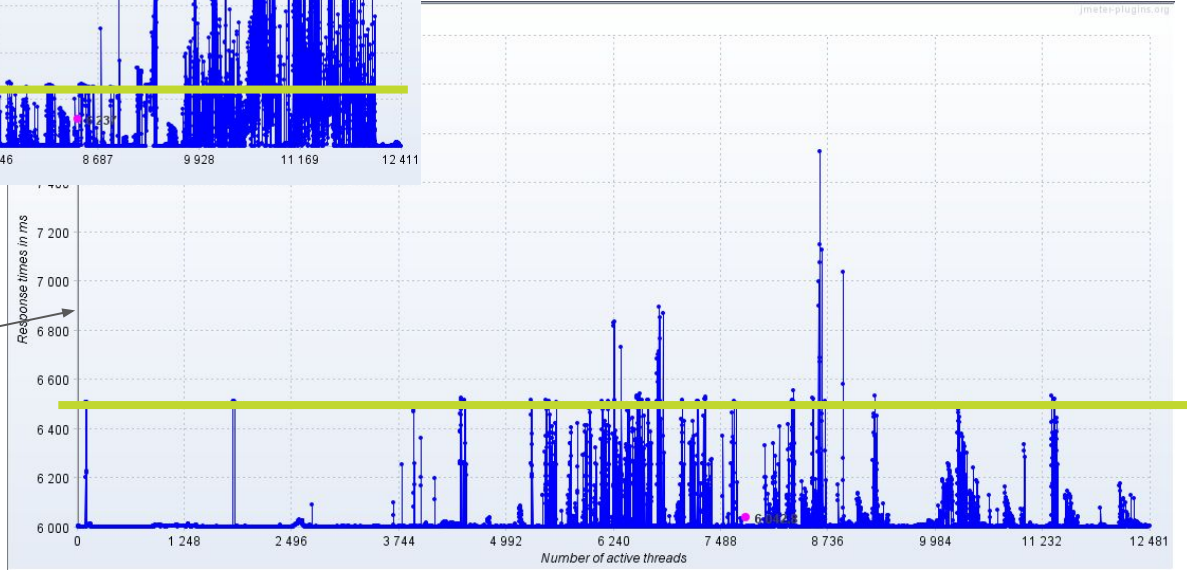
Smart Threads - Executor JConsole

Thread pool implementation

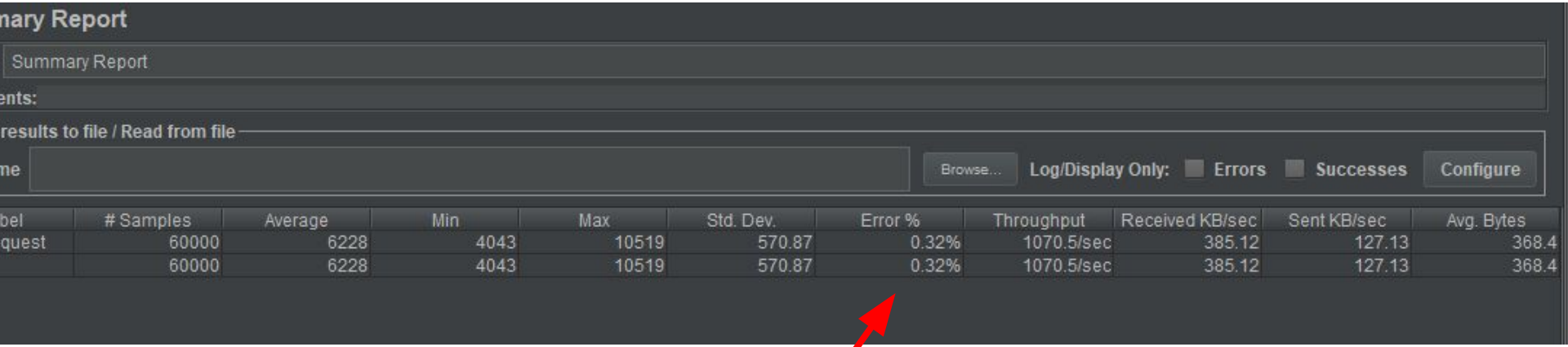
Green line at 6.5 sec



Virtual Threads



Smart Threads - Executor
JConsole



Takeaway

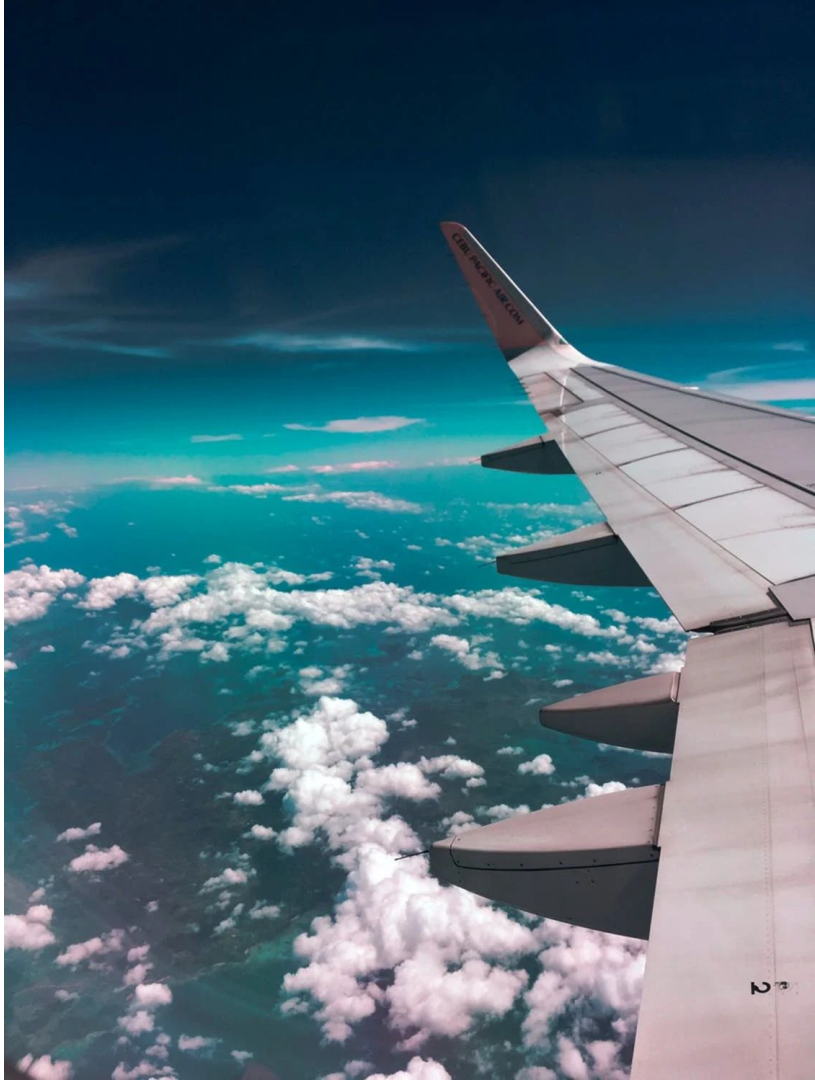
Globant Development Center La Plata, Argentina

What we know so far

1. **Project Loom is delivering on it's promise -** reducing the difficulty of writing efficient concurrent applications

2. Virtual threads implicitly convert blocking APIs into a async/await pattern - and you won't even have to be aware of it as user of an API (most of the time at least).

3. Release? Not JDK17 - fingers crossed for JDK 18 ? No promises have been made...



Useful links

More Info

1. Taking a look at Virtual Threads (Project Loom)
   - One code example showed was from this article - attribution license
2. Loom Proposal.md
   - Hear it from the dev lead of the project himself - what Loom wants to achieve
3. Java's Project Loom, Virtual Threads and Structured Concurrency with Ron Pressler
   - Podcast about Project Loom - with Ron - the Dev Lead of the project
4. **Download JDK18 + Loom Builds - to run it yourself**
   - **https://jdk.java.net/loom/**
   - The existence of EA builds does not imply that the functionality being tested will be present in any particular GA release.
5. **All examples are on Github here: https://github.com/lucaalex87/java-loom-test**

Globant Development Center in NYC, USA


Thank You!
Globant Development Center in NYC, USA