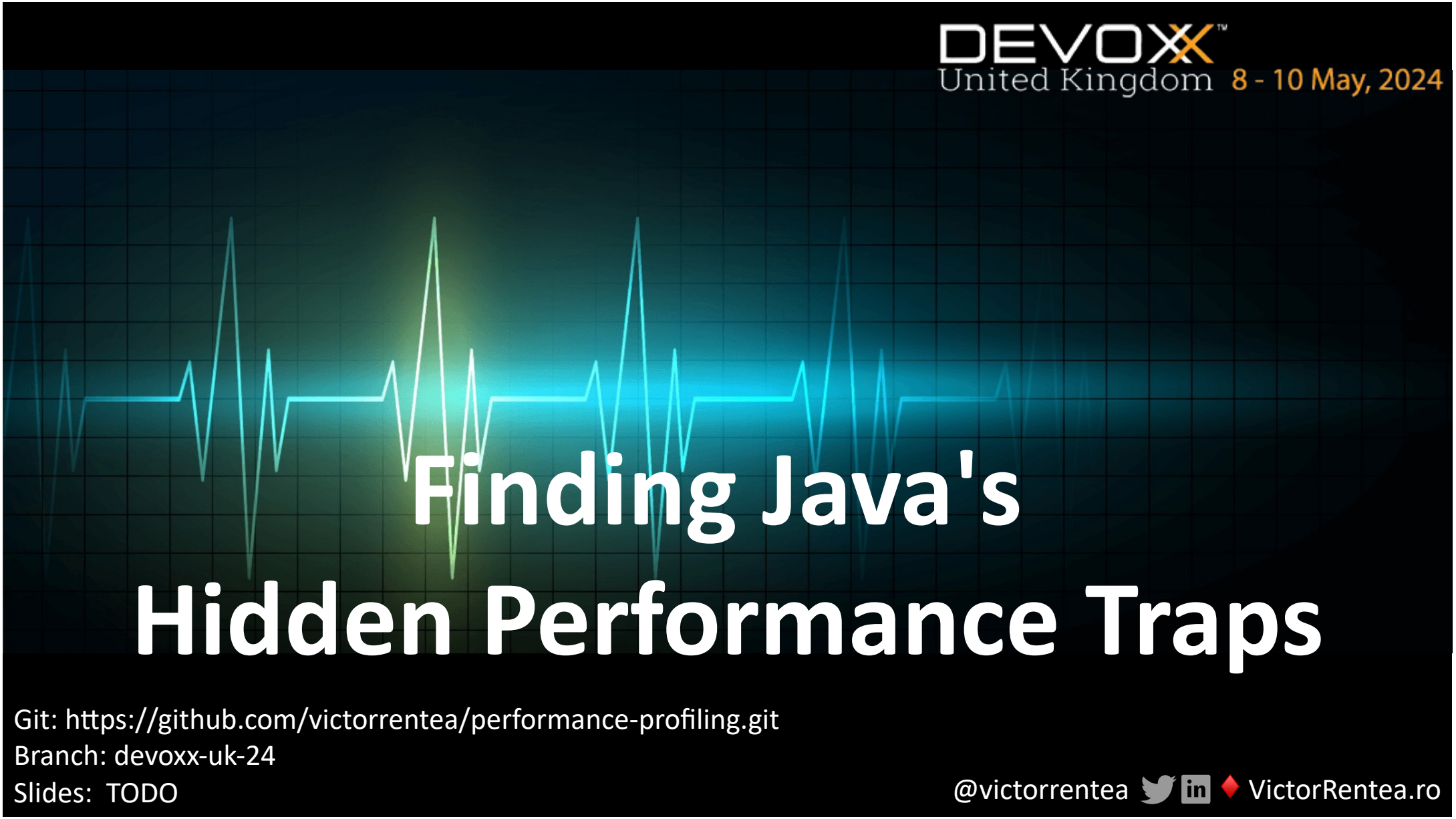
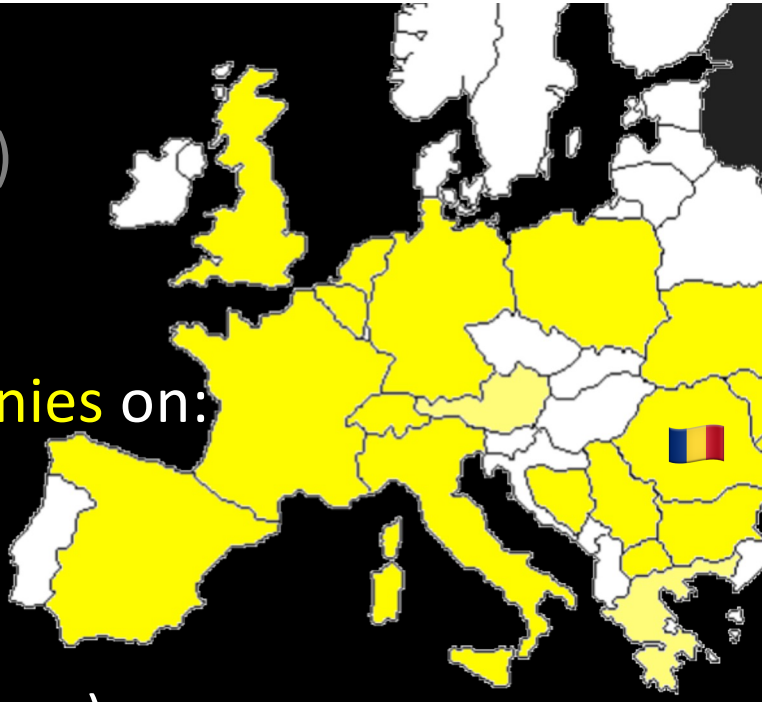

DEVOXX
United Kingdom 8 - 10 May, 2024
Finding Java's
Hidden Performance Traps
Git: https://github.com/victorrentea/performance-profiling.git
Branch: devoxx-uk-24
Slides: TODO
@victorrentea
VictorRentea.ro

👋 I'm **Victor Rentea** 🇷🇴 Java Champion, PhD(CS)

18 years of **coding**

10 years of **training & consulting** at 130+ companies on:

- ❤️ **Refactoring, Architecture, Unit Testing**
- 🛠️ Java: Spring, Hibernate, Performance, Reactive



Lead of European Software Crafters (7K developers)

Join for free monthly online meetings from 17^{00} CET

Channel: YouTube.com/vrentea ▶ past events

Life += 👰 + 👧 + 🤦 + 🐈

in 🐦 f

@victorrentea

https://victorrentea.ro

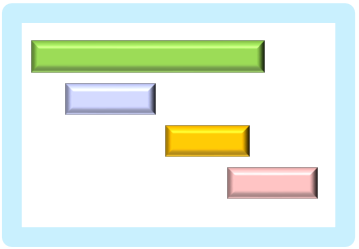
The response time of your endpoint is too long 😱

🎉 Congratulations! 🎉

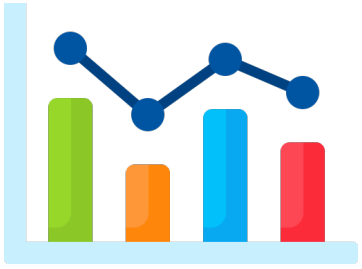
You made it to production, and
You're taking serious load!

How to start?

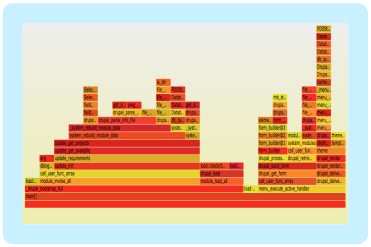
Agenda



1. Distributed Time Sampling

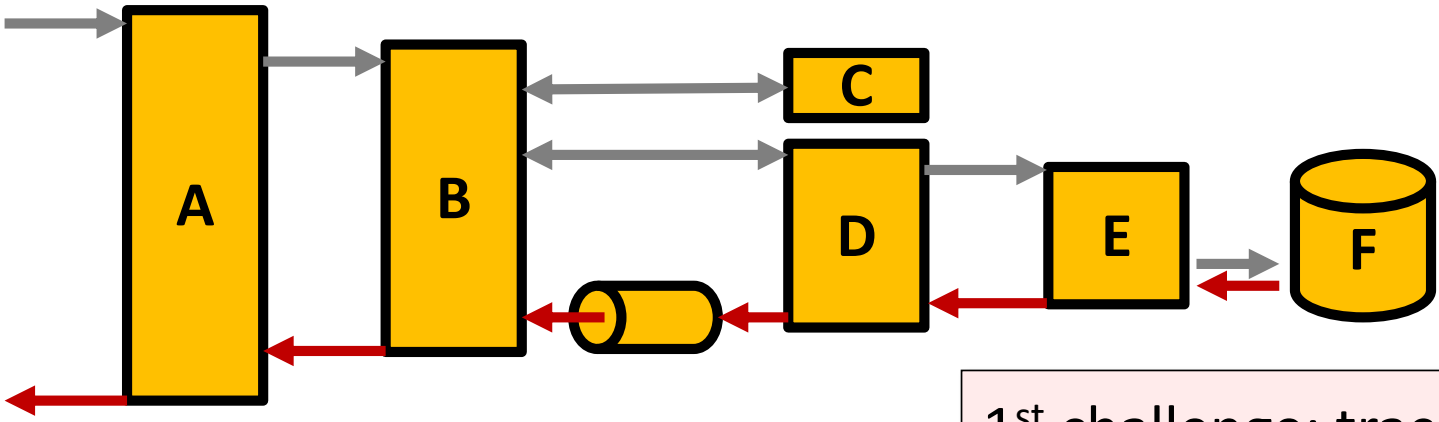


2. Metrics



3. Execution Profiling

Performance in a Distributed System



1st challenge: trace requests

Distributed Request Tracing

TraceId

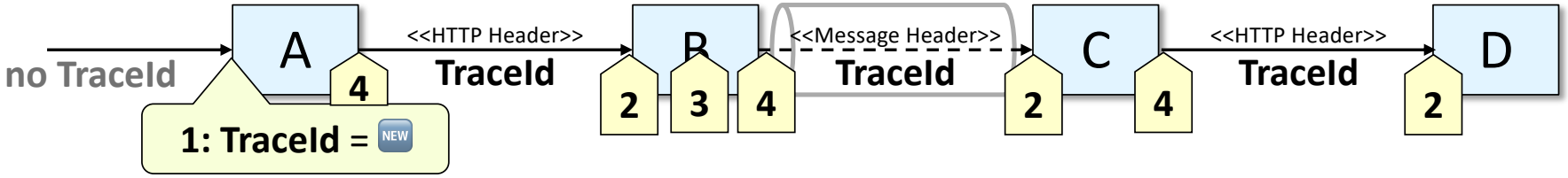
order-service.log:

```
2024-03-20T20:59:24.473 INFO [order,65fb320c74e6dd352b07535cf26506a8,776ae9fa7074b2c5]
60179 --- [nio-8088-exec-8] ...PlaceOrderRest   : Placing order for products: [1]
```

catalog-service.log:

```
2024-03-20T20:59:24.482 INFO [catalog,65fb320c74e6dd352b07535cf26506a8,9d3b54c924ab886d]
60762 --- [nio-8081-exec-7] ...GetPriceRest      : Returning prices for products: [1]
```

Distributed Request Tracing



= All systems involved in a flow share the same TraceId

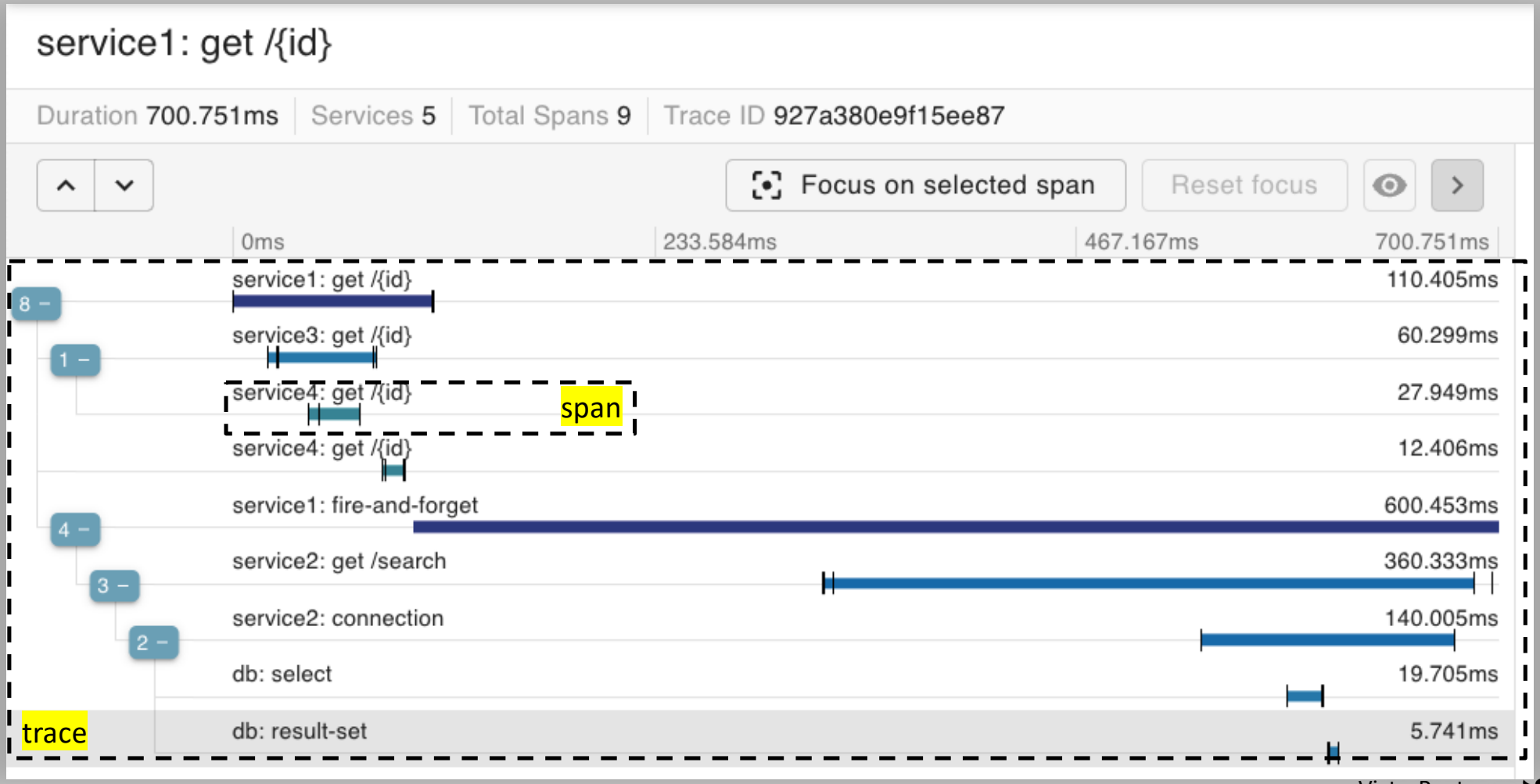
- TraceId is **generated** by the first system in a chain (eg API Gateway) (1)
- TraceId is **received as a header** in the incoming HTTP Request or Message(2)
  - Standardized by OTEL (https://opentelemetry.io)
- TraceId is prefixed to **every log line** (prev. slide)
- TraceId **propagates magically** with method calls (3)
  - via ThreadLocal => Submit all async work to Spring-injected executors ⚠
  - via ReactorContext in WebFlux => Spring must subscribe to your reactive chains ⚠
- TraceId is **sent as a header** in any outgoing Request or Message (4)
  - ⚠ By injected beans: RestTemplate, WebClient, KafkaSender...

https://spring.io/blog/2022/10/12/observability-with-spring-boot-3/

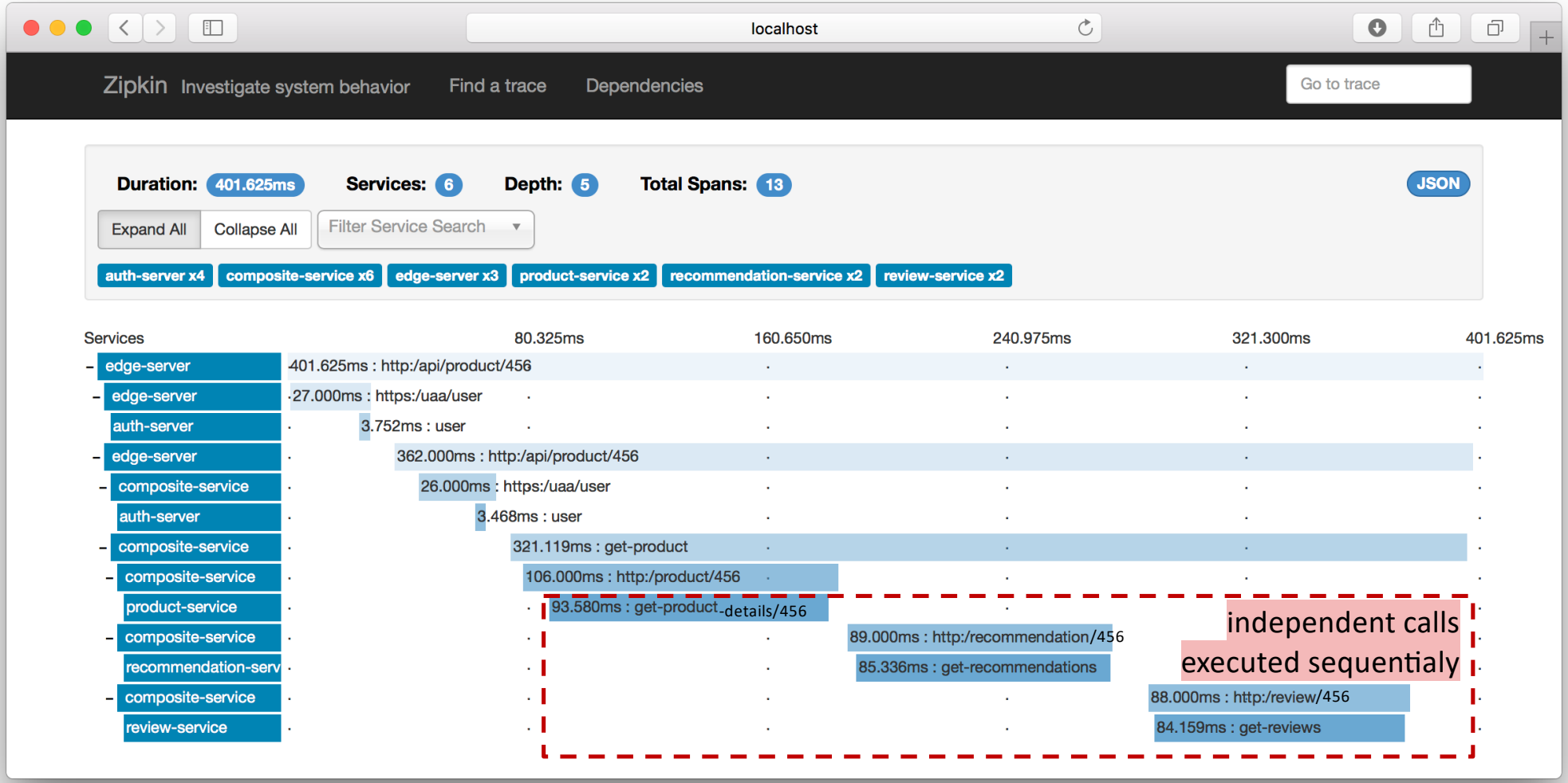
Distributed Time Budgeting

- Every system reports the time spent (***span***) working on a TraceId
- ...to a central system that tracks where time was spent
- Only a sample of traces are reported (eg 1%), for performance

Distributed Time Budgeting



Distributed Time Budgeting



Performance Anti-Patterns in a Distributed System

- **Identical repeated API calls** by accident 😱
- **Calls in a loop**: GET /items/1, /2, /3... ➔ **batch fetch**
- **System-in-the-middle**: A → B(just a proxy) → C ➔ **A→C**
- **Long chains of REST calls**: A → B ... → E, paying network latency
- **Independent API calls** ➔ **call in parallel**
- Long spans not waiting for other systems? 🤔 => Zoom in 🔎

Main Suspect:

One System

Measuring Method Runtime – High-School Style

t1-t0

```
long t0 = currentTimeMillis();
List<CommentDto> comments = client.fetchComments(loanId); // suspect code
long t1 = currentTimeMillis();
System.out.println("Took millis: " + (t1 - t0));
```

Log timestamps

```
log.info("Before call"); // 2023-05-23 00:01:50.000 ...
// suspect code
log.info("After call");  // 2023-05-23 00:01:53.100 ...
```

=> Log Clutter

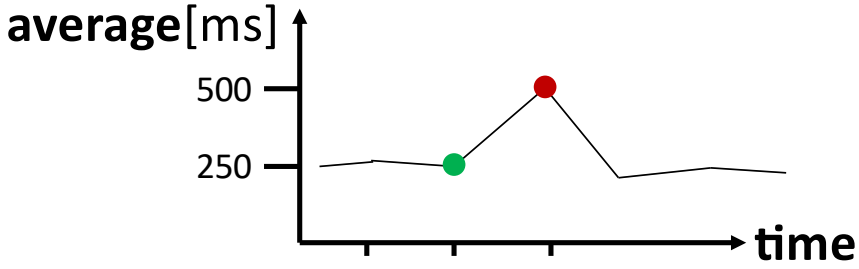
Micrometer

💖 (OpenTelemetry.io) 💖

```
@Timed // an AOP proxy intercepts the call and monitors its execution time
List<CommentDto> fetchComments(Long id) {
  // suspect: API call, heavy DB query, CPU-intensive section, ...
}
```

GET /actuator/prometheus (in a Spring Boot app) :

```
method_timed_seconds_sum{method="fetchComments"}   = 100 seconds  ≃ 250 millis/call
method_timed_seconds_count{method="fetchComments"} = 400 calls
```



5 seconds later:

```
method_timed_seconds_sum{...} =   150 - 100 = 50 seconds  ≃ 500 millis/call
method_timed_seconds_count{...} = 500 - 400 = 100 calls   (⚠double over the last 5 seconds)
```

Micrometer

https://www.baeldung.com/micrometer

```
// eg. "95% requests took ≤ 100 ms"
@Timed(percentiles = {0.95}, histogram = true)
```

⚠**Usual AOP Pitfalls**: @Timed works on methods of beans injected by Spring, and method is NOT final & NOT static; can add CPU overhead to simple methods.

```
// programatic alternative to @Timed:
meterRegistry.timer("name").record(() → { /*suspect code*/ });
meterRegistry.timer("name").record(t1 - t0, MILLISECONDS);

// accumulator
meterRegistry.counter("earnings").increment(purchase);

// current value
meterRegistry.gauge("player-count", currentPlayers.size());
```

Metrics

- **Micrometer** exposes metrics
  - your own (prev. slide)
  - critical framework metrics to track common issues (next slide)
  - optional: Tomcat, executors, ...
- ...that are pulled and stored by **Prometheus** every 5 seconds
- **Grafana** can query them later to:
  - Display impressive charts
  - Aggregate metrics across time/node/app
  - Raise alerts
- Other (paid) solutions:
  - NewRelic, DynaTrace, DataDog, Splunk, AppDynamics

Example Metrics

Exposed automatically at http://localhost:8080/actuator/prometheus

- http_server_requests_seconds_sum{method="GET",uri="/profile/showcase/{id}",} 165.04
  http_server_requests_seconds_count{method="GET",uri="/profile/showcase/{id}",} 542.0
- tomcat_connections_current_connections{...} 298 ⚠ 98 requests waiting in queue
  tomcat_threads_busy_threads{...} 200 ⚠ = using 100% of the threads => thread pool starvation?
- jvm_gc_memory_allocated_bytes_total 1.8874368E9
  jvm_memory_usage_after_gc_percent{...} 45.15 ⚠ used>30% => increase max heap?
- hikaricp_connections_acquire_seconds_sum{pool="HikariPool-1",} 80.37 ⚠ conn pool starvation?
  hikaricp_connections_acquire_seconds_count{pool="HikariPool-1",} 551.0 ➔ 145 ms waiting
- cache_gets_total{cache="cache1",result="miss",} 0.99 ⚠ 99% miss = bad config? wrong key?
  cache_gets_total{cache="cache1",result="hit",} 0.01
- logback_events_total{level="error",} 1255.0 ⚠ + 1000 errors over the last 5 seconds => 😱
- my_own_custom_metric xyz

Finding common performance problems should be done now.

Best Practice

Capture and Monitor metrics for any recurrent performance issue

Bottleneck in code that does not expose metrics?

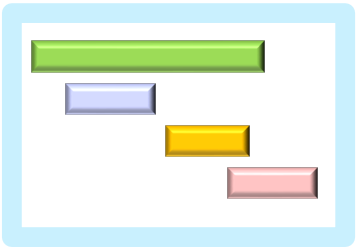
(library or unknown legacy code)

Let's measure execution time of *all* method?

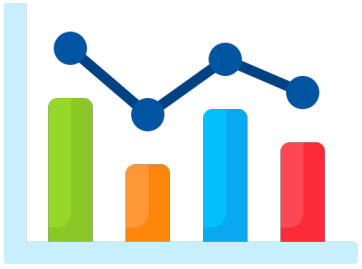
Naive: instrument your JVM with a -javaagent
that measures the execution any executed method

OMG, NO!
Adds huge CPU overhead!

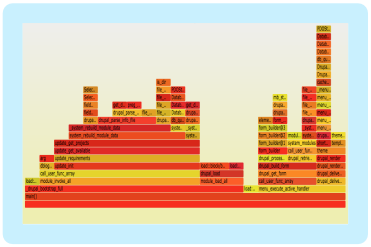
Agenda



~~1. Distributed Time Sampling~~



~~2. Metrics~~



3. Execution Profiling


a training by
Rentea
Java Flight Recorder
The JVM Profiler
VictorRentea.ro

Java Flight Recorder (JFR)

1. **Captures stack traces** of running threads every second = samples
2. **Records internal JVM events** about:
   - Locks, pinned Carrier Threads (Java 21)
   - File I/O
   - Network Sockets
   - GC
   - Your own custom events
3. **Very-low overhead** (<2%) => **Can be used in production** 💖
4. **Free** since Java 11 (link)

Using JFR

- **Local JVM**, eg via **GlowRoot, Java Mission Control** or **IntelliJ**

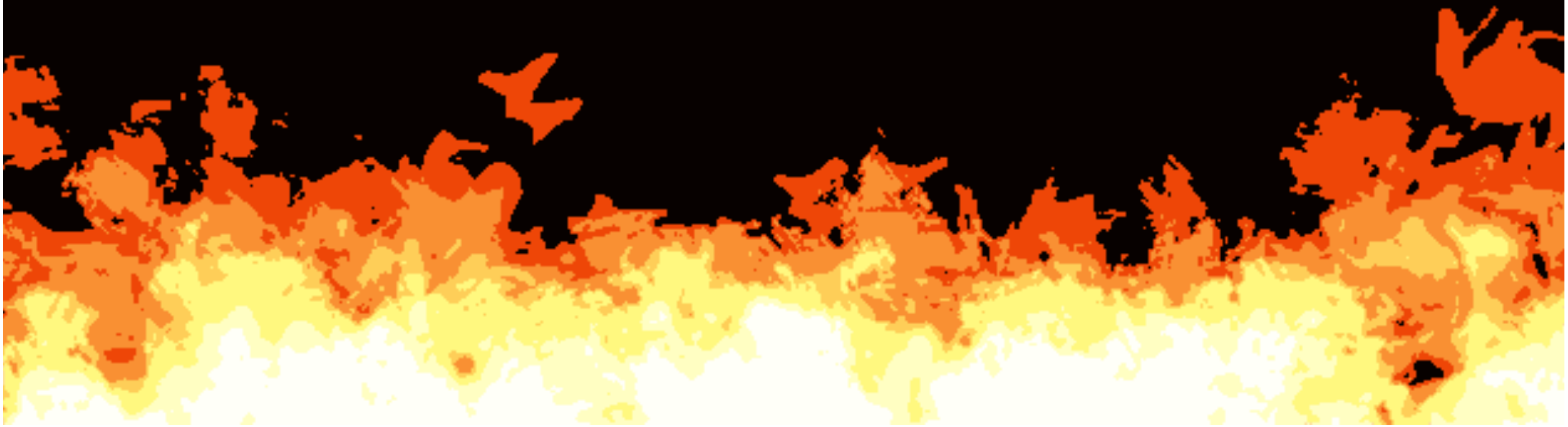
- **Remote JVM:** ⏺ ... ⏹ => analyze generated .jfr file in JMC/IntelliJ

  First, **enable JFR** via -XX:+FlightRecorder

  - Auto-start **on startup**: -XX:StartFlightRecording=....
  - Start via **command-line** (SSH): jcmd <JAVAPID> JFR.start
  - Start via **endpoints**: JMX or Spring Boot Actuator
  - Start via **tools**: DynaTrace, DataDog, Glowroot (in our experiment)

Flame Graph

Visualize the samples captured by an execution profiler



JFR captures stack traces of running threads once every second

```
13    public void dummy() {
14      entry();
15    }
16    private void entry() {
17      f(); 33%
18      g(); 66%
19    }
20    private void f() {
21      Thread.sleep( millis: 2010);
22    }
23    private void g() {
24      Thread.sleep( millis: 4010);
25    }
```

2 samples (33%)

Thread.sleep(Native Method)
Flame.f(Flame.java:21)
Flame.entry(Flame.java:17)
Flame.dummy(Flame.java:14)

4 samples (66%)

Thread.sleep(Native Method)
Flame.g(Flame.java:24)
Flame.entry(Flame.java:18)
Flame.dummy(Flame.java:14)

Anatomy of a Flame Graph

JFR captures stack traces of running threads once every second

Length of each bar is proportional to the number of samples (≈time)

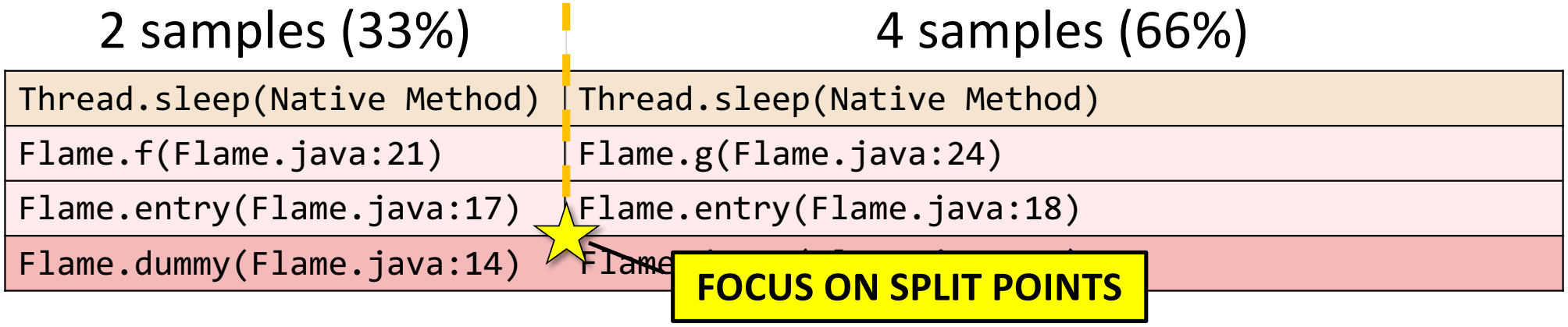
2 samples (33%)	4 samples (66%)
Thread.sleep(Native Method)	Thread.sleep(Native Method)
Flame.f(Flame.java:21)	Flame.g(Flame.java:24)
Flame.entry(Flame.java:17)	Flame.entry(Flame.java:18)
Flame.dummy(Flame.java:14)	Flame.dummy(Flame.java:14)

Anatomy of a Flame Graph

JFR captures stack traces of running threads once every second

Length of each bar is proportional to the number of samples (≈time)

Identical bars are merged



Anatomy of a Flame Graph

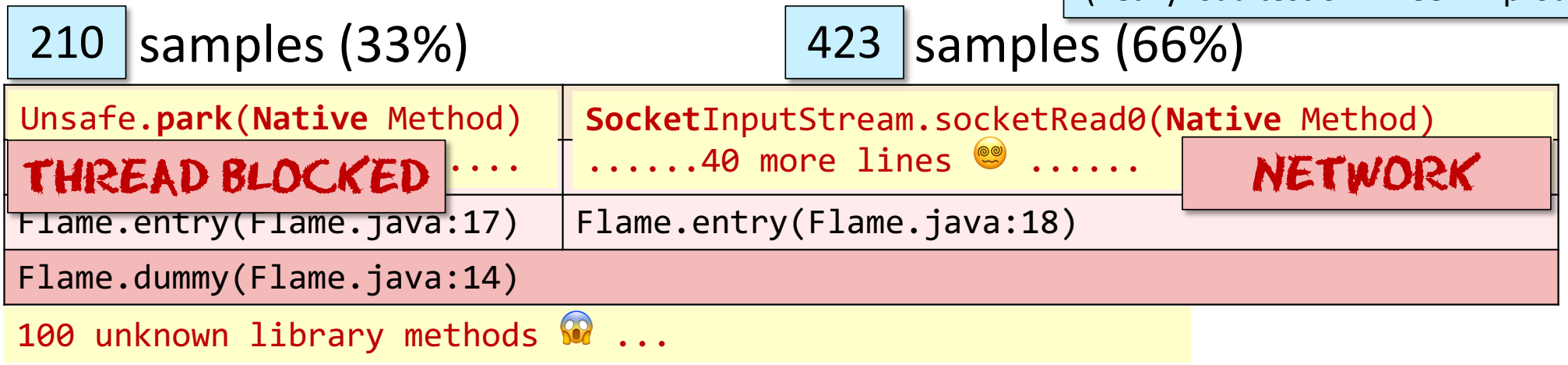
IN REAL LIFE

JFR samples the stack traces of running threads every second

Length of each bar is proportional to the number of samples (≈time)

Identical bars are merged

More samples improve accuracy
(heavy load test or 1 week in prod)



Experiment Environment

Load Tests

Java app

(Gatling.io)

23 parallel requests (closed world model)

or jMeter, K6..

ProfiledApp

Spring Boot 3+

http://localhost:8080

JFR

Glowroot.org

-javaagent:glowroot.jar

http://localhost:4000

display captured flamegraph

ToxiProxy

github.com/Shopify/toxiproxy

in a Local Docker

localhost:55432

+3ms network latency

PostgreSQL

in a Local Docker

localhost:5432

WireMock

in a Local Docker

http://localhost:9999

remote API responses with 40ms delay

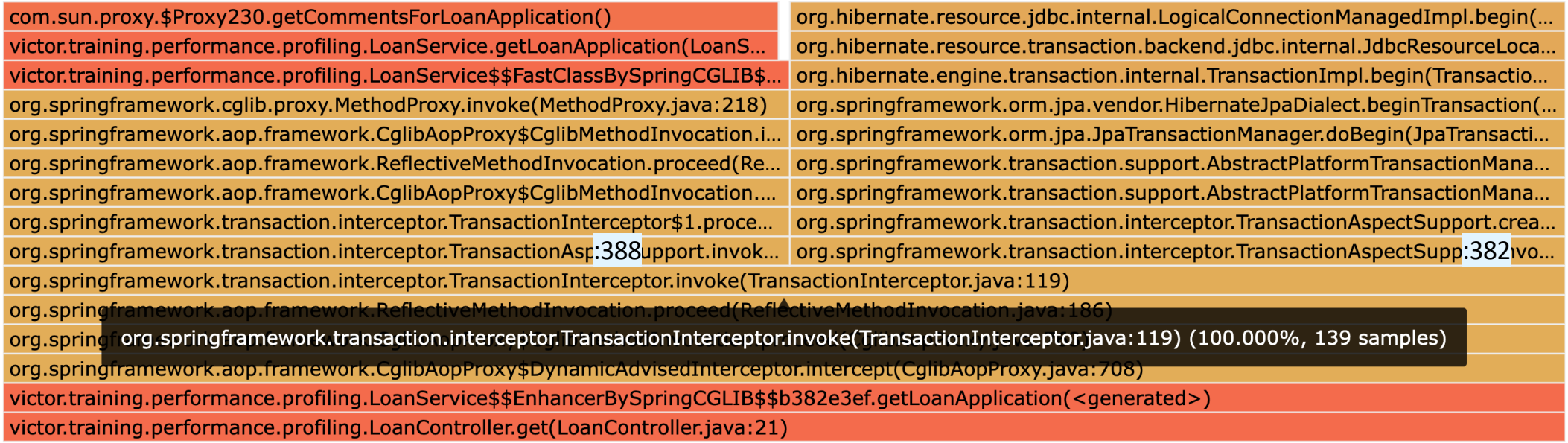

EXPERIMENT

Glowroot.org

- Simple, didactic profiler
  - Add VM arg: -javaagent:/path/to/glowroot.jar
  - Open http://localhost:4000
- Monitors
  - Methods Profiling, displayed as Flame Graphs
  - SQLs
  - API calls
  - Typical bottlenecks
  - Data per endpoint
- Free
  - Probably not production-ready ⚠

Expensive Aspects

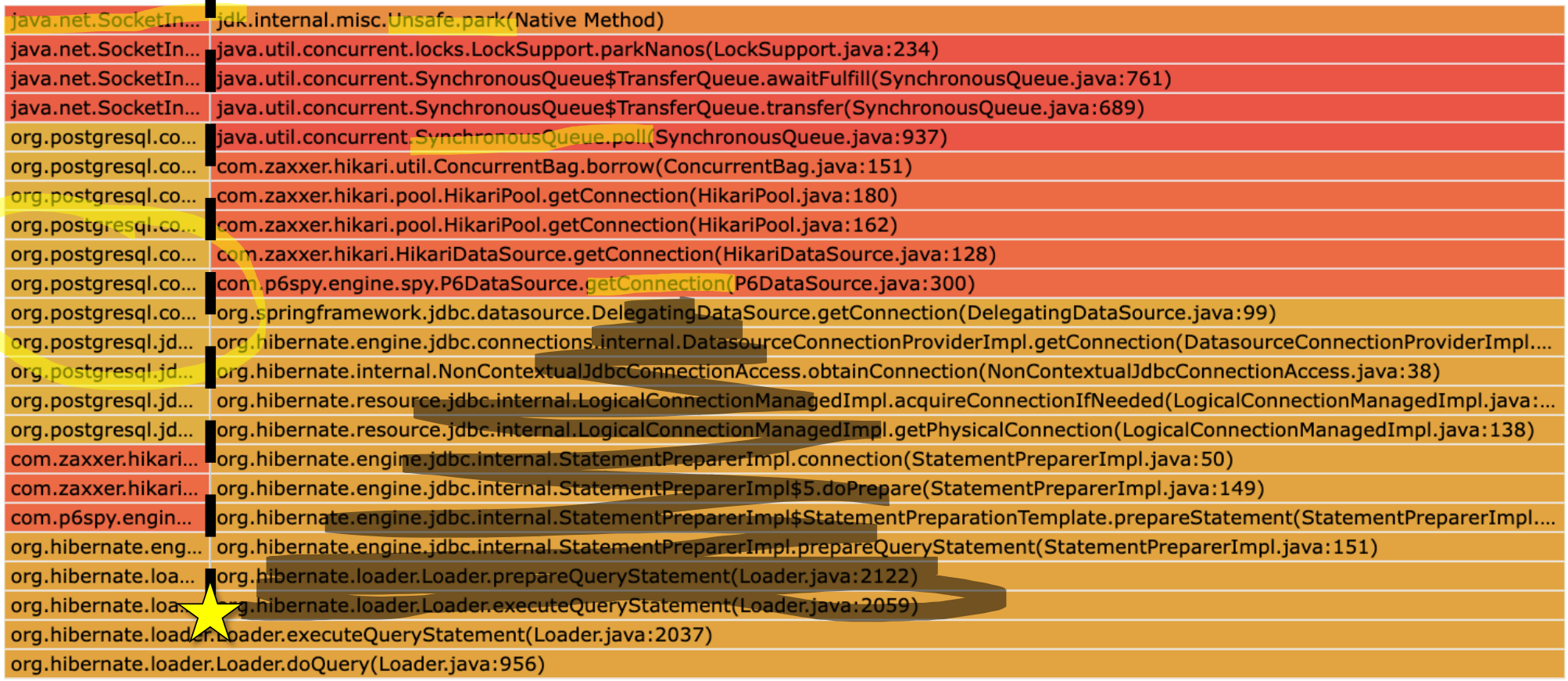
TransactionInterceptor
acquires a JDBC connection **before entering** the @Transactional method
and releases that connection **after the end** of the method



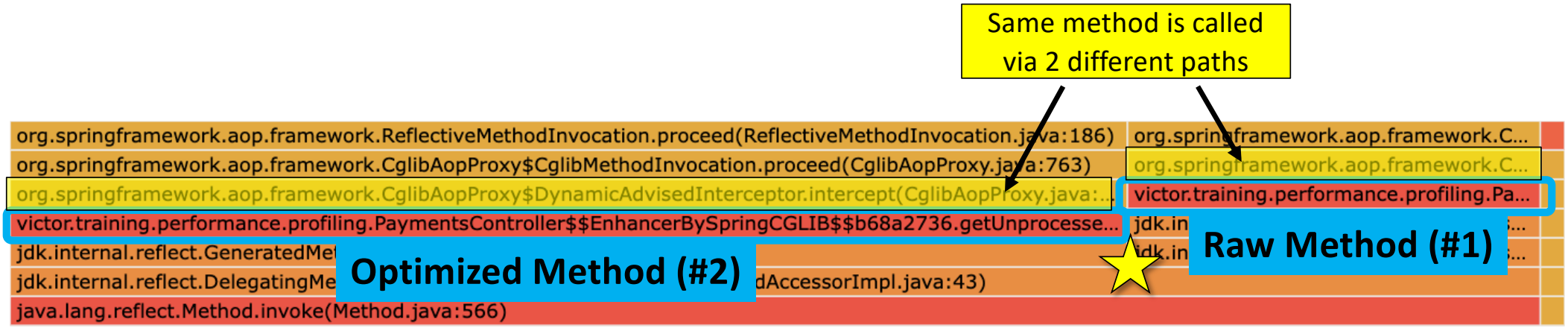
JDBC Connection Pool Starvation

Waiting for SQL server

← 85% of the query endpoint is spent acquiring a JDBC Connection →



False Split by JIT Optimization



Fix:

a) **Warmup JVM**: eg. repeat load test (already optimized)

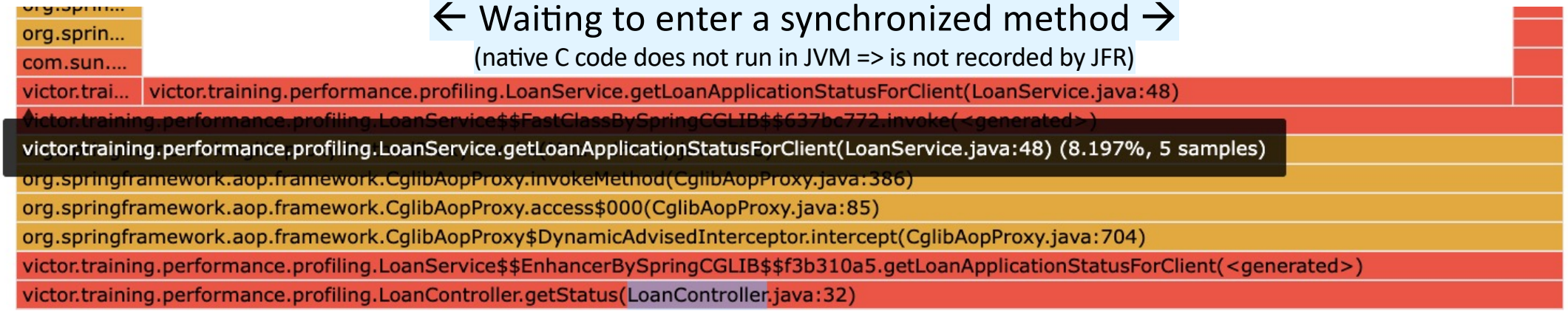
b) **Record more samples** (all future calls calls go to #2 Optimized)

Lock Contention

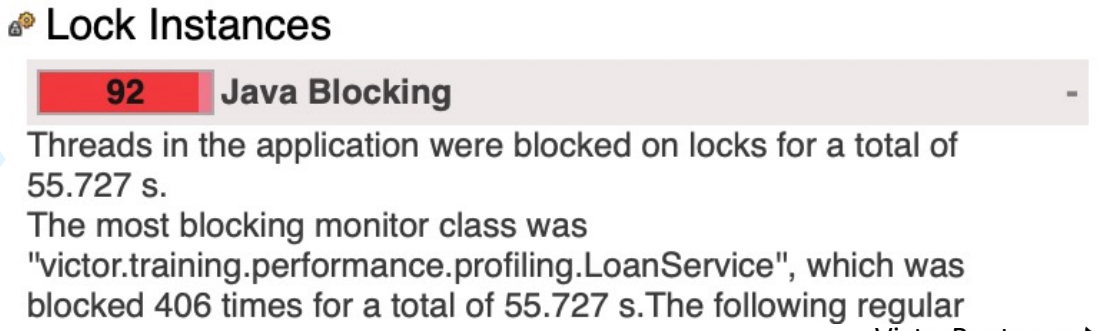
```
47    public synchronized Status getLoanApplicationStatusForClient(Long id) {
48      LoanApplication loanApplication = loanApplicationRepo.findById(id).orElseThrow();
```

← Waiting to enter a synchronized method →

(native C code does not run in JVM => is not recorded by JFR)



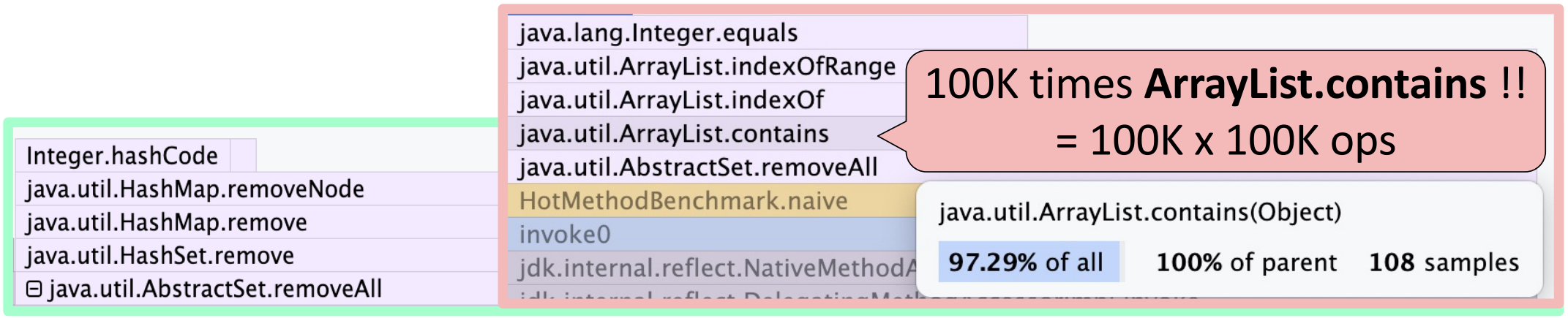
Automatic analysis of the .jfr output by **JDK Mission Control** (JMC)
https://wiki.openjdk.org/display/jmc/Releases



🔥Hot (CPU) Method

```
hashSet.removeAll(list);
```

set.size()	list.size()	Δ time
100.000	100	1 millis
100.000	99.999	50 millis
100.000	100.000	2.5 seconds (50x more) 😱



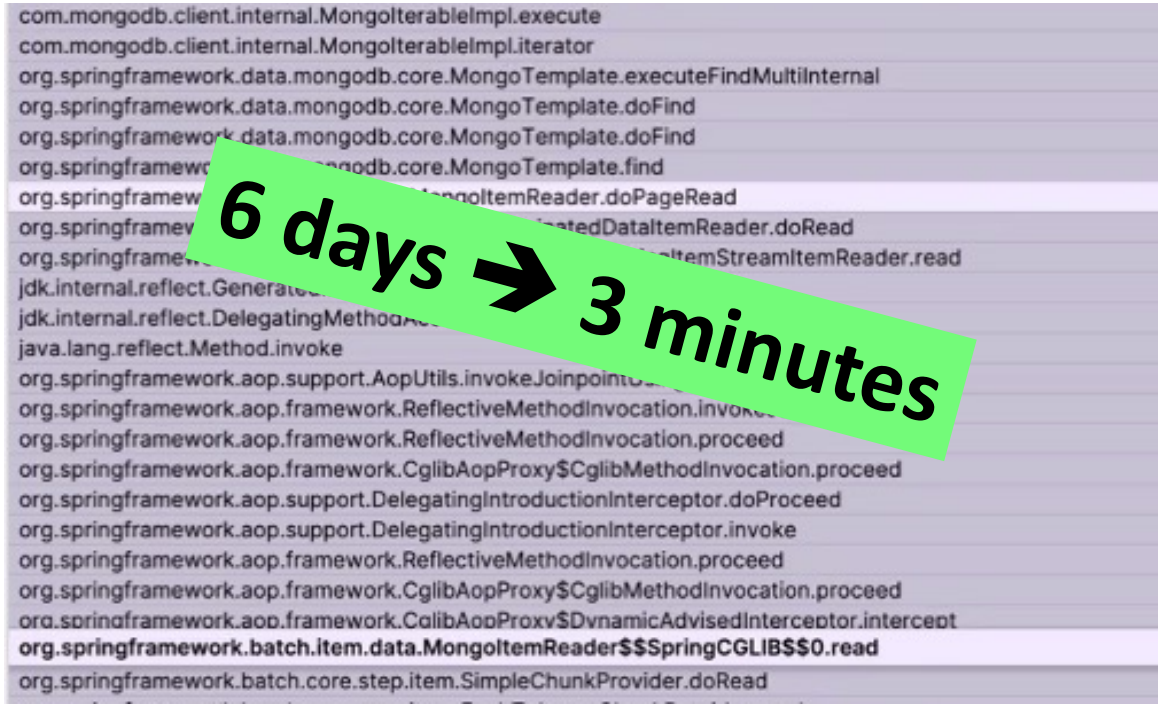
War Story: Exporting from Mongo

Export took 6 days!!?!!#$%$@

The team was concern about misusing an obscure library (jsonl)

JFR showed that MongoItemReader loaded data in pages, not streaming (link)

Switched to MongoCursorItemReader!

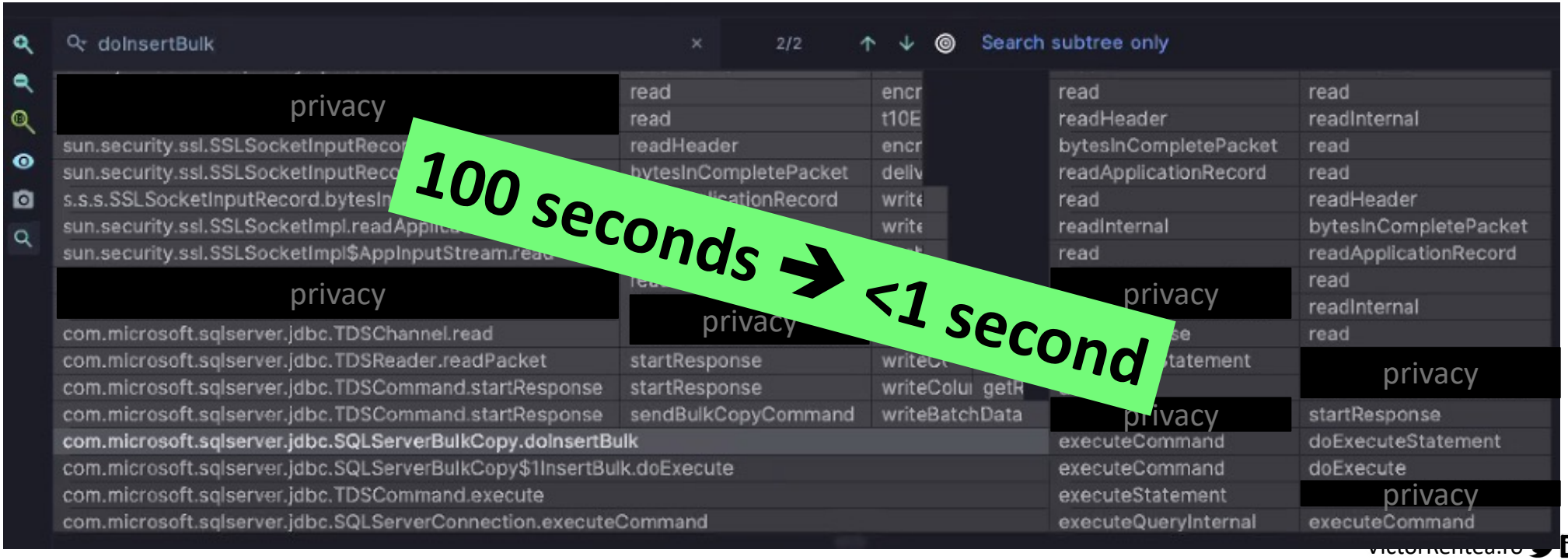


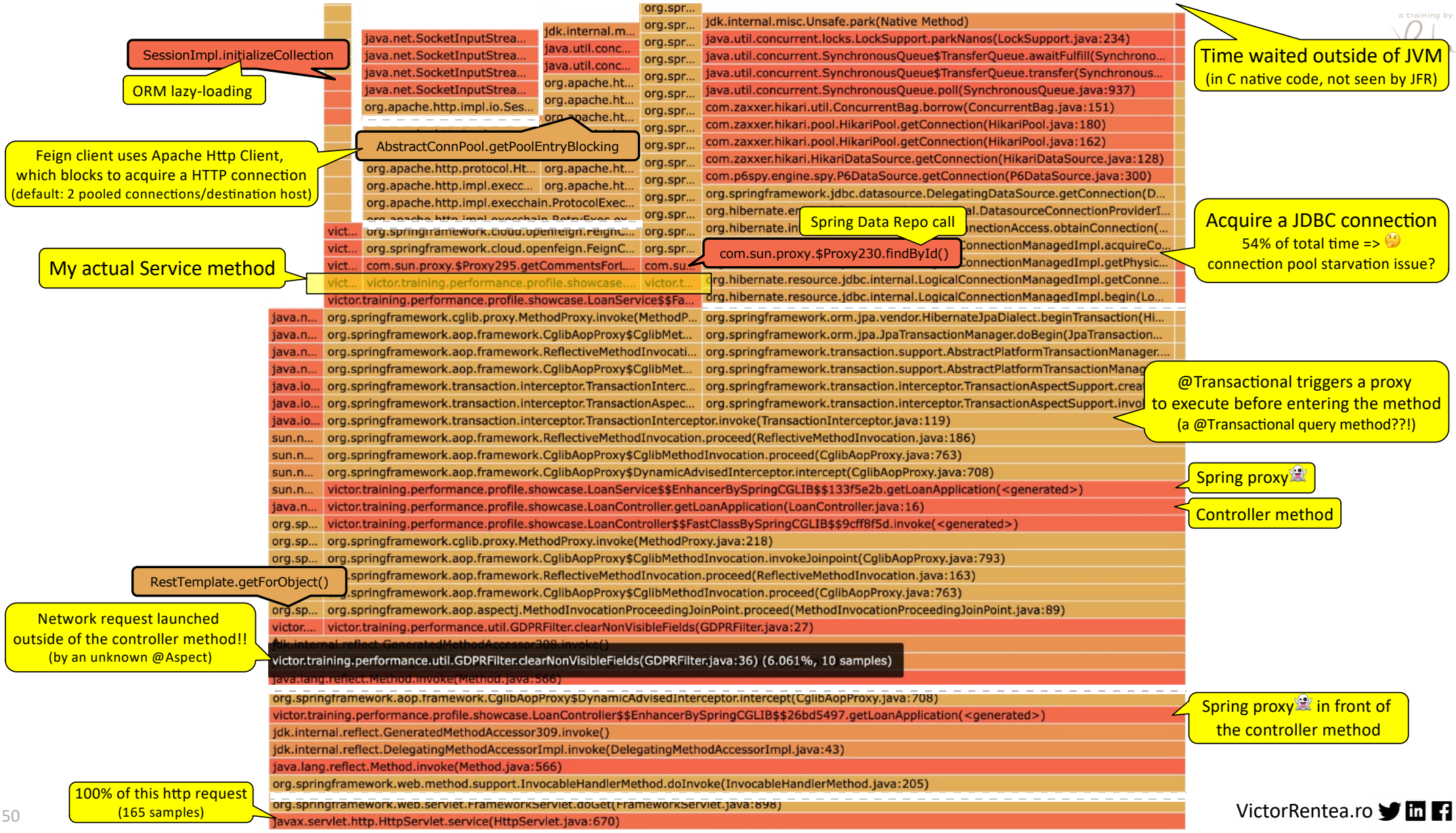
War Story: Importing in MsSQL

INSERT #1: 11.000 rows ... 1 second – OK.

INSERT #2: 5.000 rows ... 100 seconds !!?😬

Profiler showed that #2 was NOT using "doInsertBulk" as #1
StackOverflow revealed an [issue inserting in DATE columns](#).




SessionImpl.initializeCollection
ORM lazy-loading
AbstractConnPool.getPoolEntryBlocking
Feign client uses Apache Http Client, which blocks to acquire a HTTP connection (default: 2 pooled connections/destination host)
My actual Service method
Spring Data Repo call
com.sun.proxy.$Proxy230.findById()
Time waited outside of JVM
(in C native code, not seen by JFR)
Acquire a JDBC connection
54% of total time =>
connection pool starvation issue?
@Transactional triggers a proxy to execute before entering the method
(a @Transactional query method??!)
Spring proxy
Controller method
RestTemplate.getForObject()
Network request launched outside of the controller method!!
(by an unknown @Aspect)
Spring proxy in front of the controller method
100% of this http request
(165 samples)
jdk.internal.misc.Unsafe.park(Native Method)
java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:234)
java.util.concurrent.SynchronousQueue$TransferQueue.awaitFulfill(Synchrono...
java.util.concurrent.SynchronousQueue$TransferQueue.transfer(Synchronous...
java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:937)
com.zaxxer.hikari.util.ConcurrentBag.borrow(ConcurrentBag.java:151)
com.zaxxer.hikari.pool.HikariPool.getConnection(HikariPool.java:180)
com.zaxxer.hikari.pool.HikariPool.getConnection(HikariPool.java:162)
com.zaxxer.hikari.HikariDataSource.getConnection(HikariDataSource.java:128)
com.p6spy.engine.spy.P6DataSource.getConnection(P6DataSource.java:300)
org.springframework.jdbc.datasource.DelegatingDataSource.getConnection(D...
victor.training.performance.profile.showcase.LoanService$$Fa...
org.springframework.cglib.proxy.MethodProxy.invoke(MethodP...
org.springframework.aop.framework.CglibAopProxy$CglibMet...
org.springframework.aop.framework.ReflectiveMethodInvocati...
org.springframework.aop.framework.CglibAopProxy$CglibMet...
org.springframework.transaction.interceptor.TransactionInterc...
org.springframework.transaction.interceptor.TransactionAspec...
org.springframework.transaction.interceptor.TransactionInterceptor.invoke(TransactionInterceptor.java:119)
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.proceed(CglibAopProxy.java:763)
org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:708)
victor.training.performance.profile.showcase.LoanService$$EnhancerBySpringCGLIB$$133f5e2b.getLoanApplication(<generated>)
victor.training.performance.profile.showcase.LoanController.getLoanApplication(LoanController.java:16)
victor.training.performance.profile.showcase.LoanController$$FastClassBySpringCGLIB$$9cff8f5d.invoke(<generated>)
org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:218)
org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.invokeJoinpoint(CglibAopProxy.java:793)
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:163)
org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.proceed(CglibAopProxy.java:763)
org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint.proceed(MethodInvocationProceedingJoinPoint.java:89)
victor.training.performance.util.GDPRFilter.clearNonVisibleFields(GDPRFilter.java:27)
victor.training.performance.util.GDPRFilter.clearNonVisibleFields(GDPRFilter.java:36) (6.061%, 10 samples)
org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:708)
victor.training.performance.profile.showcase.LoanController$$EnhancerBySpringCGLIB$$26bd5497.getLoanApplication(<generated>)
jdk.internal.reflect.GeneratedMethodAccessor309.invoke()
jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
java.lang.reflect.Method.invoke(Method.java:566)
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:205)
javax.servlet.http.HttpServlet.service(HttpServlet.java:670)

Flagship Feature of Java 21?

🎉 Released on 19 Sep 2023 🎉

Virtual Threads

(Project Loom)

JVM can reuse the 1MB OS **Platform Thread**
for other tasks **when you block** in a call to an API/DB

Blocking is free!! 🥳

Continuation....	Continuation.yield0(ContinuationScope, Continuation)
Continuation....	Continuation.yield(ContinuationScope)
VirtualThread...	VirtualThread.yieldContinuation()
VirtualThread...	VirtualThread.park()
System$2.pa...	System$2.parkVirtualThread()
VirtualThread...	VirtualThreads.park()
LockSupport....	LockSupport.park()
Poller.poll2(in...	Poller.poll2(int, long, BooleanSupplier)
Poller.poll(int,...	Poller.poll(int, long, BooleanSupplier)
Poller.poll(int,...	Poller.poll(int, int, long, BooleanSupplier)
NioSocketIm...	NioSocketImpl.park(FileDescriptor, int, long)
NioSocketIm...	NioSocketImpl.park(FileDescriptor, int)
NioSocketIm...	NioSocketImpl.implRead(byte[], int, int)
Socket.conne...	NioSocketImpl.read(byte[], int, int)
Socket.conne...	NioSocketImpl$1.read(byte[], int, int)
NetworkClien...	Socket$SocketInputStream.read(byte[], int, int)
HttpClient.op...	BufferedInputStream.fill()
HttpClient.op...	BufferedInputStream.read1(byte[], int, int)
HttpClient.<i...	BufferedInputStream.implRead(byte[], int, int)
HttpClient.Ne...	BufferedInputStream.read(byte[], int, int)
HttpClient.Ne...	HttpClient.parseHTTPHeader(MessageHeader, ProgressSource, HttpURLConnection)
HttpURLConn...	HttpClient.parseHTTP(MessageHeader, ProgressSource, HttpURLConnection)
HttpURLConn...	HttpURLConnection.getInputStream0()
HttpURLConn...	HttpURLConnection.getInputStream()
HttpURLConn...	HttpURLConnection.getResponseCode()
SimpleBufferingClientHttpRequest.executeInternal(HttpHeaders, byte[])	
AbstractBufferingClientHttpRequest.executeInternal(HttpHeaders)	
AbstractClientHttpRequest.execute()	
RestTemplate.doExecute(URI, String, HttpMethod, RequestCallback, ResponseExtractor)	
RestTemplate.execute(String, HttpMethod, RequestCallback, ResponseExtractor, Object[])	
RestTemplate.getForObject(String, Class, Object[])	
NonBlockingNetworkCalls.seq()	

JFR can record ❤️Virtual Threads❤️

(a Spring Boot 3 app running on Java 21)

Structured Concurrency

(Java 25 LTS 🤞)

Thread Dump of code using Virtual Threads + **Structured Concurrency (Java 25)** captures the parent thread

```
{
  "container": "jdk.internal.misc.ThreadFlock@6373f7e2",
  "parent": "java.util.concurrent.ThreadPerTaskExecutor@634c5b98",
  "owner": "83",
  "threads": [
   {
     "tid": "84",
     "name": "",
     "stack": [
        "java.base\/jdk.internal.vm.Continuation.yield(Continuation.java:357)",
        "java.base\/java.lang.VirtualThread.yieldContinuation(VirtualThread.java:370)",
        "java.base\/java.lang.VirtualThread.park(VirtualThread.java:499)",
....
        "org.springframework.web.client.RestTemplate.getForObject(RestTemplate.java:378)",
        "victor.training.java.loom.StructuredConcurrency.fetchBeer(StructuredConcurrency.java:
....
        "java.base\/jdk.internal.vm.Continuation.enter(Continuation.java:320)"
     ]
```

JFR Key Points

- ✅ Sees inside unknown code / libraries
- ❌ **CANNOT** see into native C++ code (like **`synchronized`** keyword)
- ⚠️ Sample-based: requires load tests or record in production
- ✅ Can be used in production thanks to extra-low overhead <2%
- ⚠️ Tuning requires patience, expectations, and library awareness
- ⚠️ JIT optimizations can bias profile of CPU-intensive code (link)
- ❌ **CANNOT** follow thread hops (eg. thread pools) or reactive chains
- ✅ Records Virtual Threads and Structured Concurrency

Finding the Bottleneck

- Identify slow system (eg w/ Zipkin)
- Study its metrics + expose more (eg w/ Grafana)
- Profile the execution in production / load-tests
  - Load the `.jfr` in IntelliJ (code navigation) and in JMC (auto-analysis)
- Optimize code/config, balancing gain vs. risk vs. code mess

Tracing Java's Hidden

Performance Traps

Network Calls	Aspects, JPA, Lombok ..
Resource Starvation	Naive **`@Transactional`**
Concurrency Control	Large **`synchronized`**
CPU work	Library misuse/misconfig

Git: https://github.com/victorrentea/performance-profiling.git
Branch: devoxx-uk-24

Thank You!

Meet me online at:

{EUROPEAN SOFTWARE CRAFTERS}

I was Victor Rentea,
trainer & coach for experienced teams

victor.rentea@gmail.com ♦ @victorrentea ♦ VictorRentea.ro