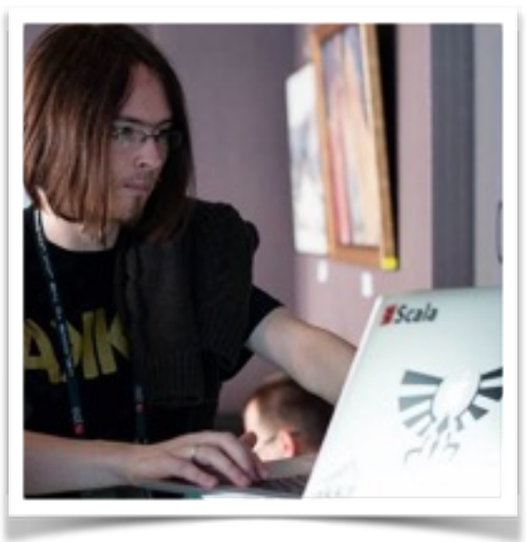

NEED FOR ASYNC
HOT PURSUIT
FOR SCALABLE APPS
Konrad `@ktosopl` Malawski

Typesafe

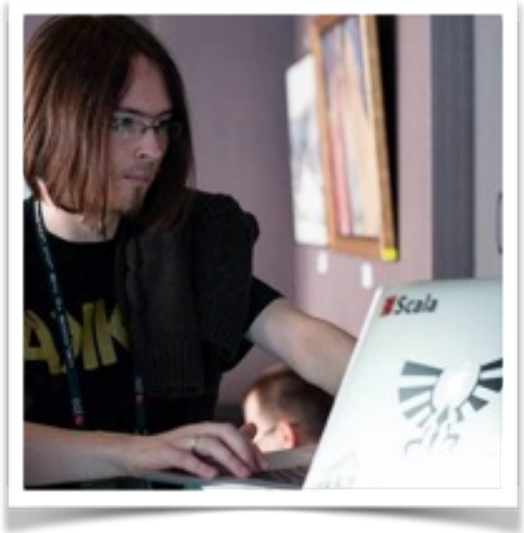
Typesafe



Konrad `ktoso` Malawski

Akka Team,
Reactive Streams TCK

Typesafe



polish java user group

geecon

Konrad `@ktosopl` Malawski

[sckrk]

KRAKOW SCALA

akka.io
typesafe.com
geecon.org
Java.pl / KrakowScala.pl
sckrk.com / meetup.com/Paper-Cup @ London
GDGKrakow.pl
lambdakrk.pl

Nice to meet you!
Who are you guys?

High Performance Software Development

For the majority of the time,
high performance software development
is not about compiler hacks and bit twiddling.

It is about fundamental design principles that are key to doing any effective software development.

Martin Thompson

practicalperformanceanalyst.com/2015/02/17/getting-to-know-martin-thompson-...

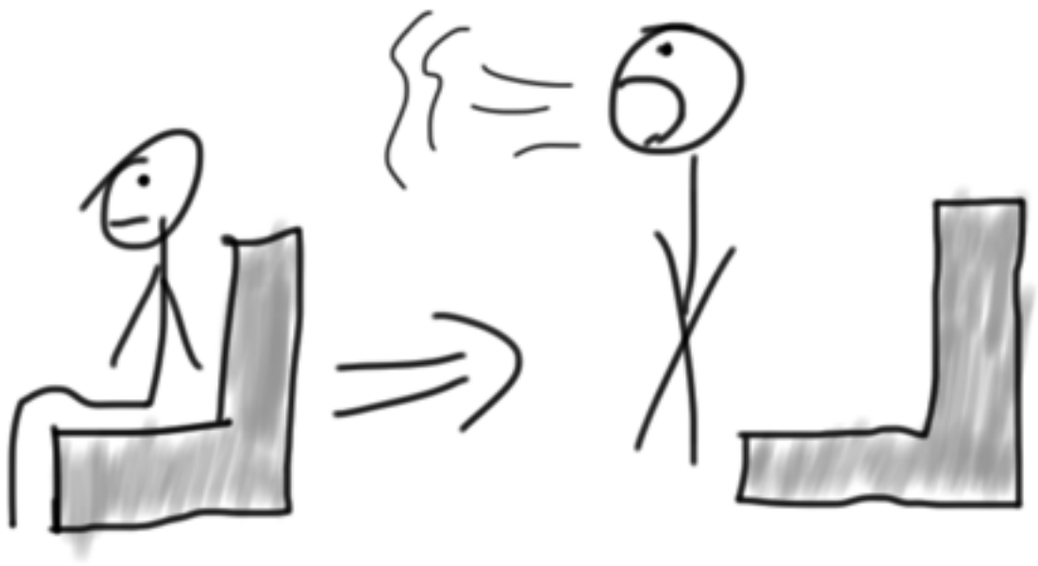
Agenda

- Async and Synch basics / definitions
- Async where it matters: Scheduling
- How NOT to measure Latency
- Concurrent < lock-free < wait-free
- I/O: IO, AIO, NIO, Zero
- C10K: select, poll, epoll / kqueue
- Distributed Systems: Where Async is at Home
- Wrapping up and Q/A

Init()

Brace yourself, heavy and low-level contents ahead!

Get a deep breath and stretch your bones!



Init()

! WARNING !

Shared Mutable State (!!!)

Concurrent Access to Shared Mutable State (!!!!!)

Side Effects (!!)

Locks (!!!)

No locks (!!!)

System calls

Schedulers

Big Oh

C code (!)

Assembly code (!!!)

Stick men!

Init()

! WARNING !

Shared Mutable State (!!!)

Concurrent Access to Shared Mutable State (!!!!!)

Side Effects (!!)

Locks (!!!)

No locks (!!!)

System calls

Schedulers

Big Oh

C code (!)

Assembly code (!!!)

Stick men!

All for the sake of our pure functional code.

For it to run. Run fast.

Sync / Async Basics

Sync / Async

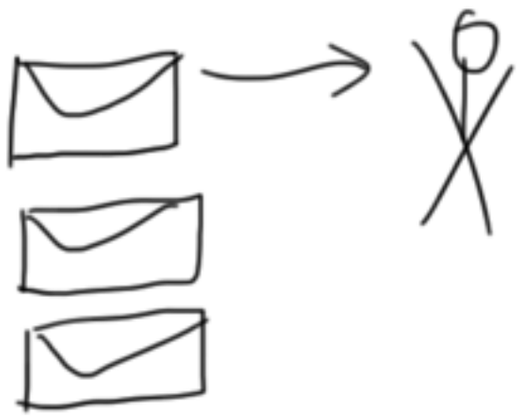
Sync / Async

Sync / Async

Sync / Async

Synchronous

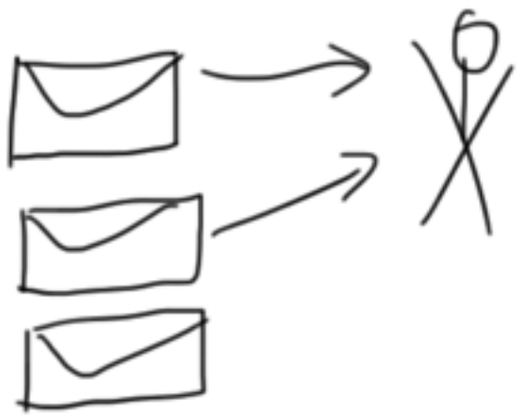
Asynchronous



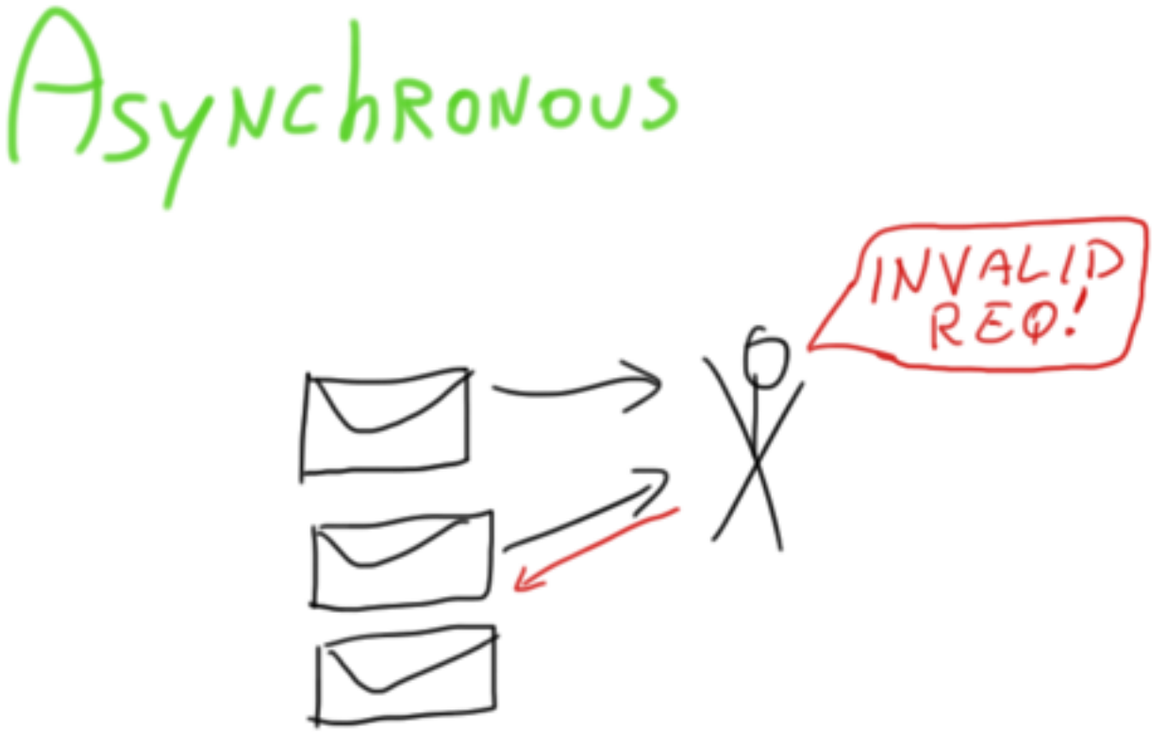
Sync / Async

Synchronous

Asynchronous



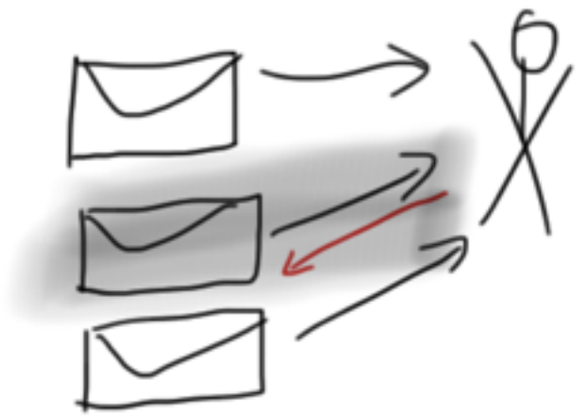
Sync / Async



Sync / Async

Synchronous

Asynchronous



Sync / Async

Synchronous

Asynchronous

Sync / Async

Synchronous

Asynchronous

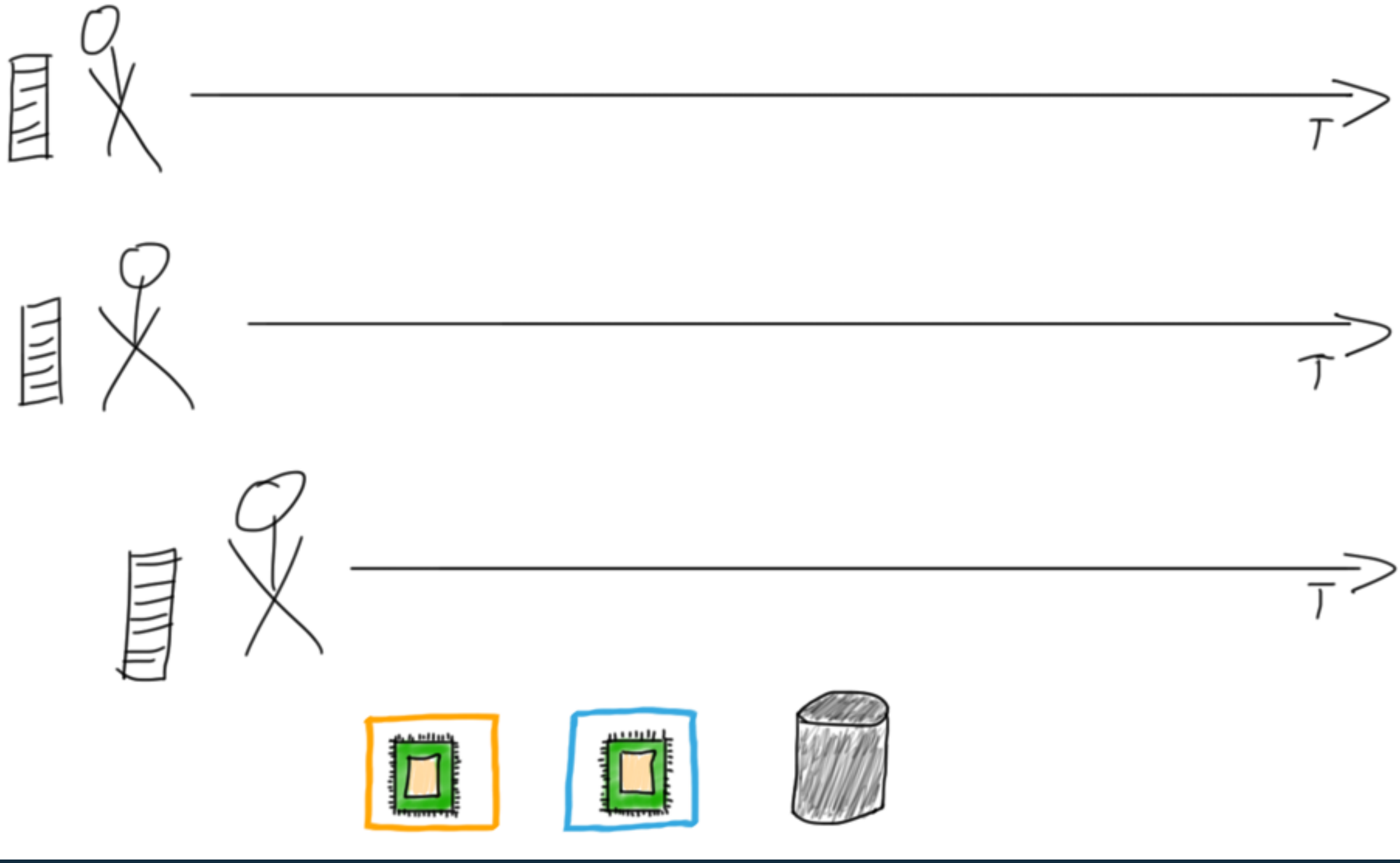
Async where it matters:

Highly parallel systems

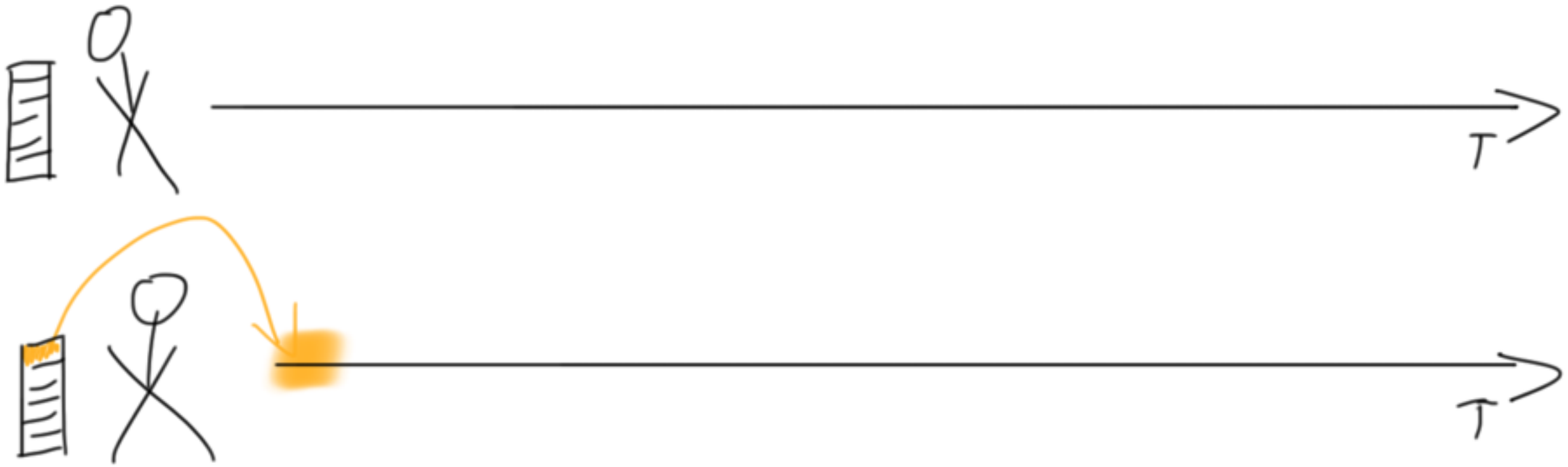
Event loops

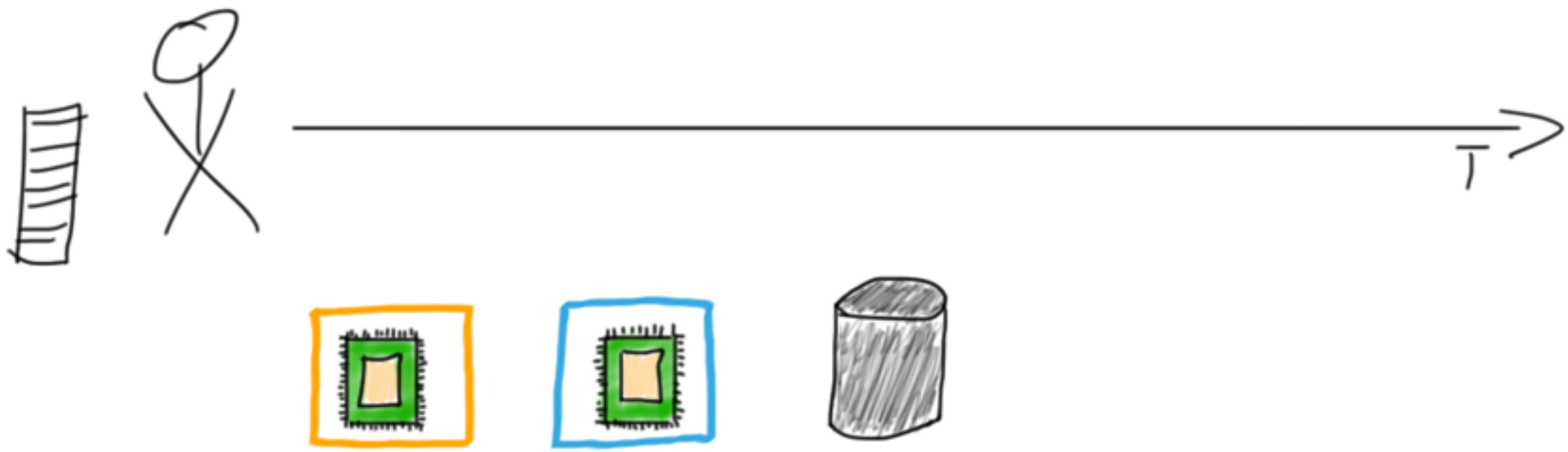
Actors

Async where it matters: Scheduling

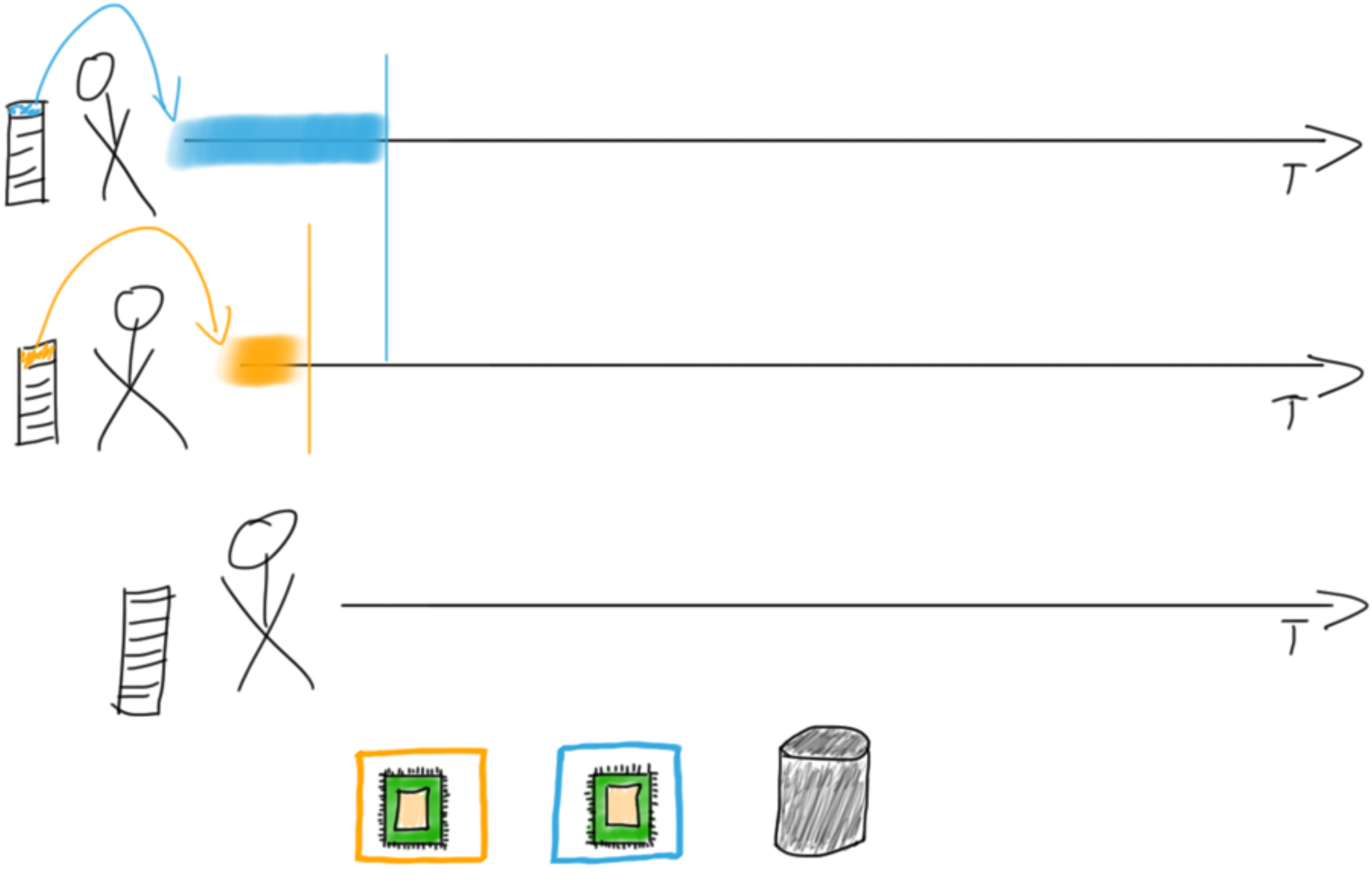


Async where it matters: Scheduling

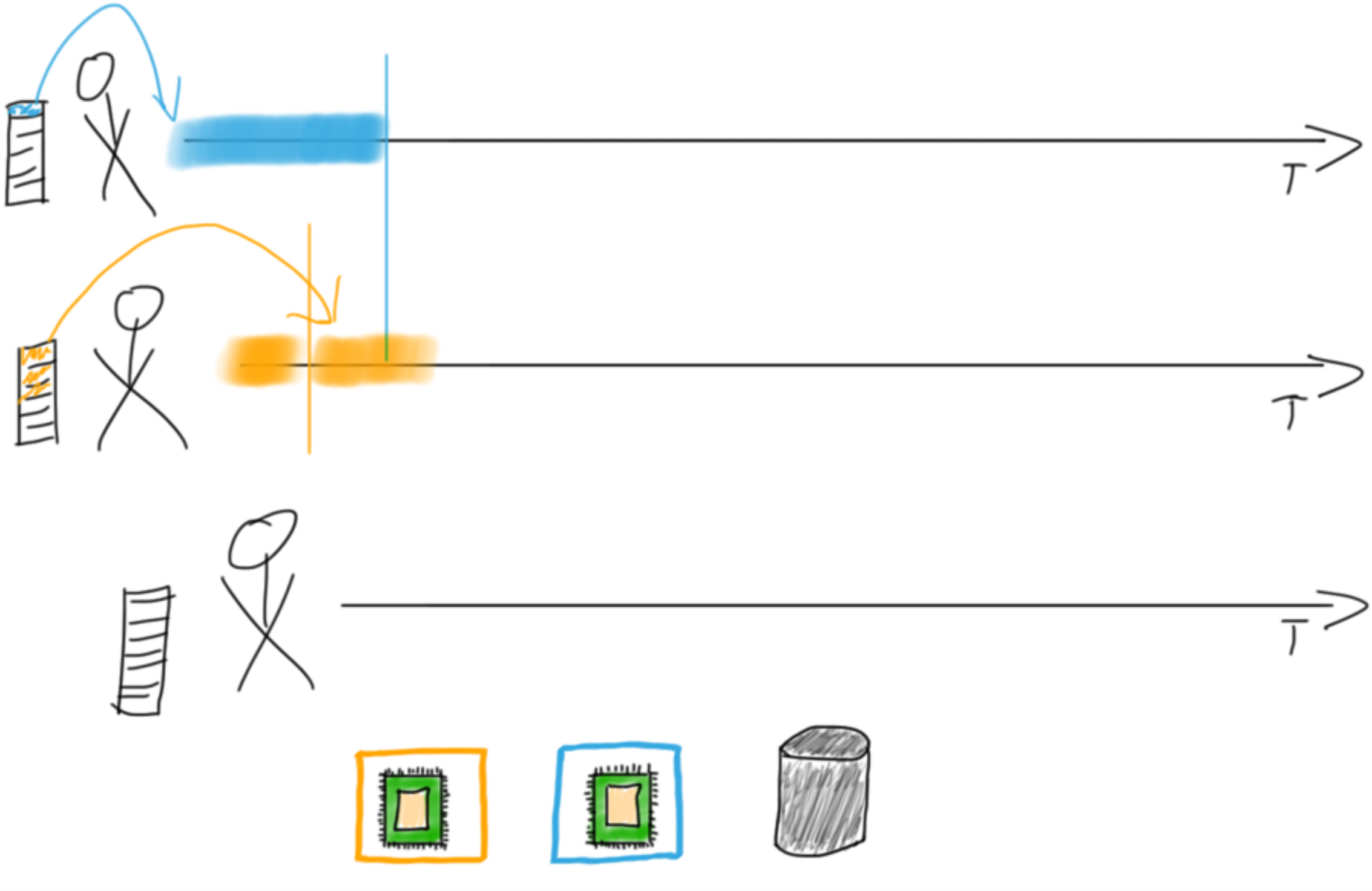




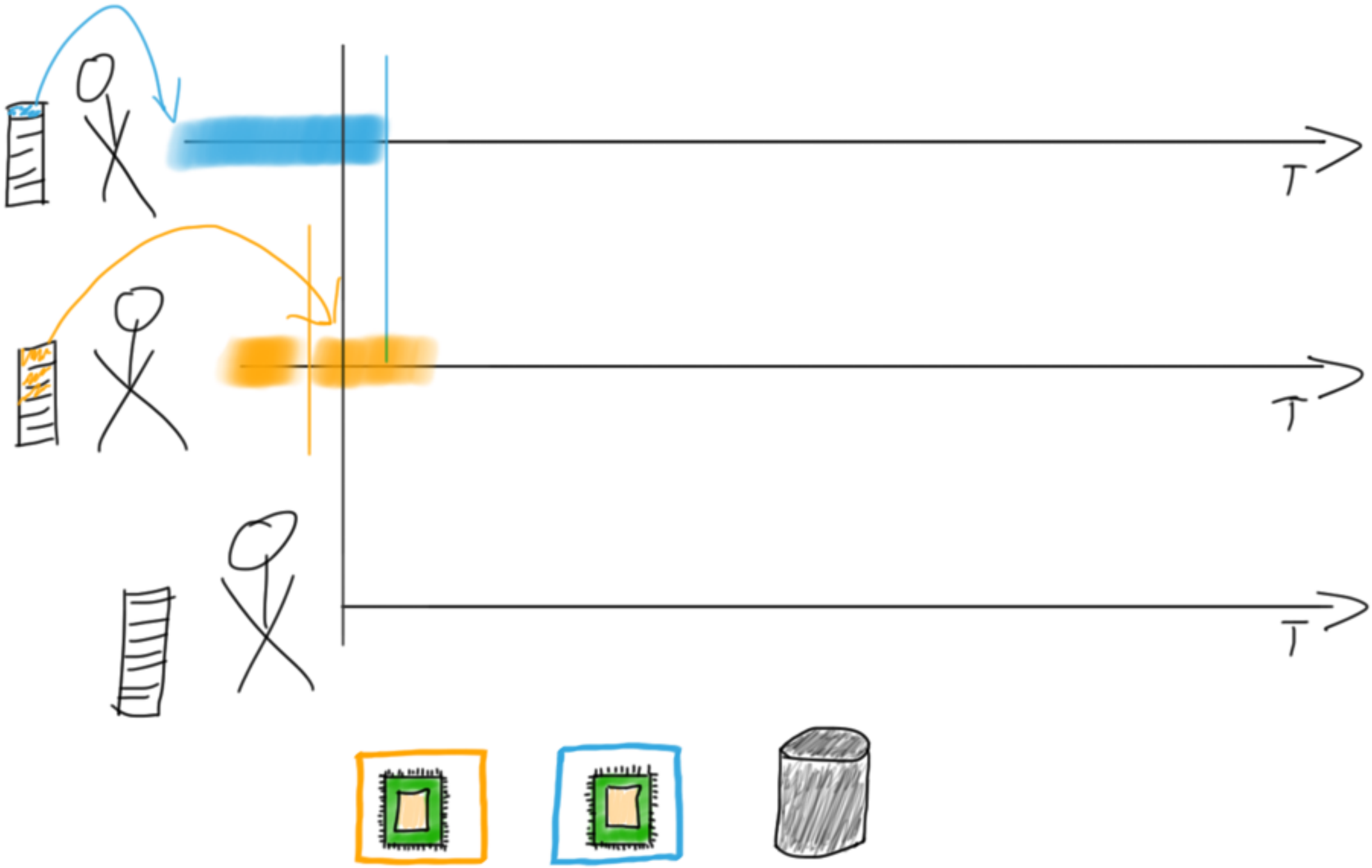
Async where it matters: Scheduling



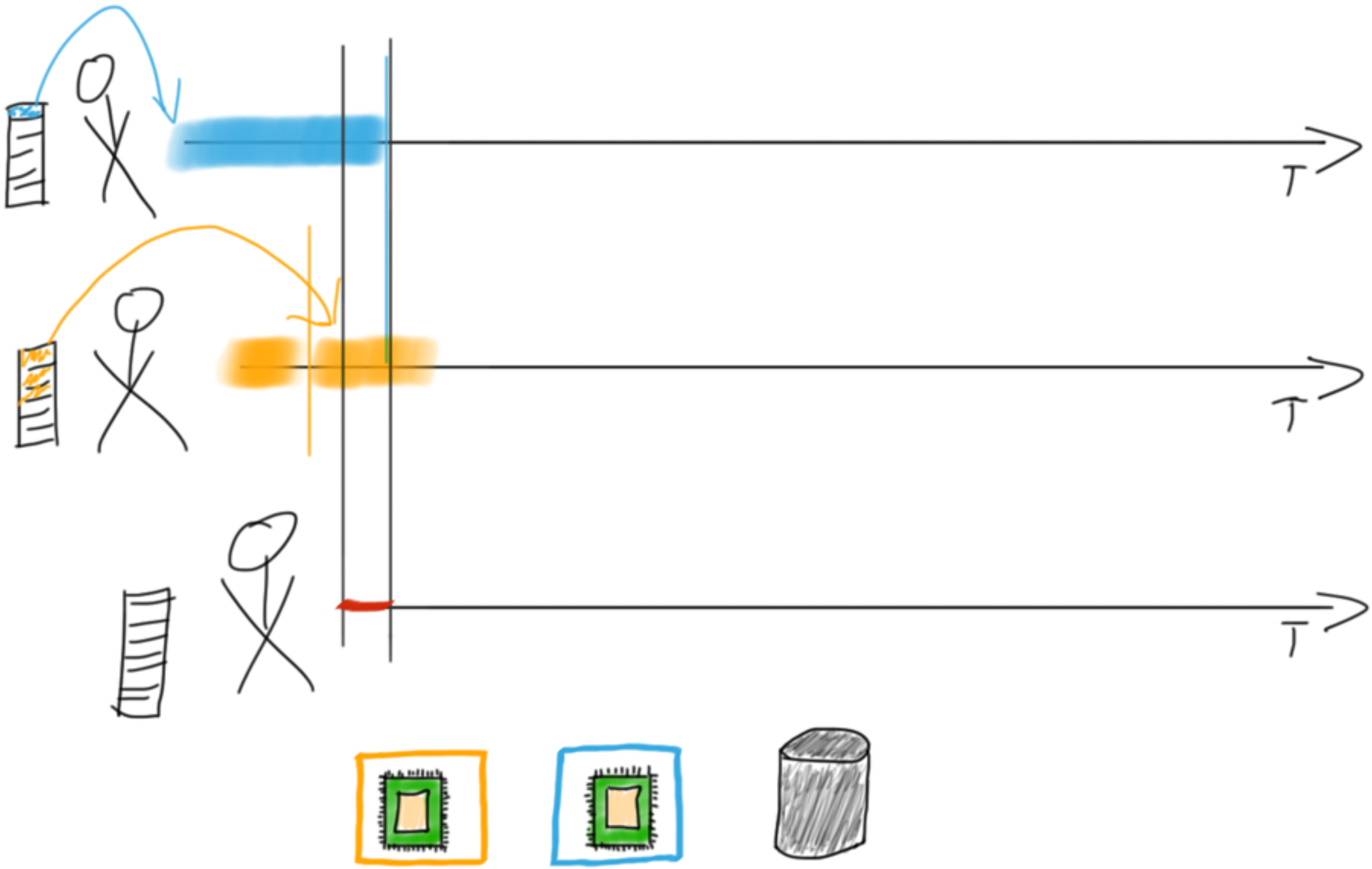
Async where it matters: Scheduling



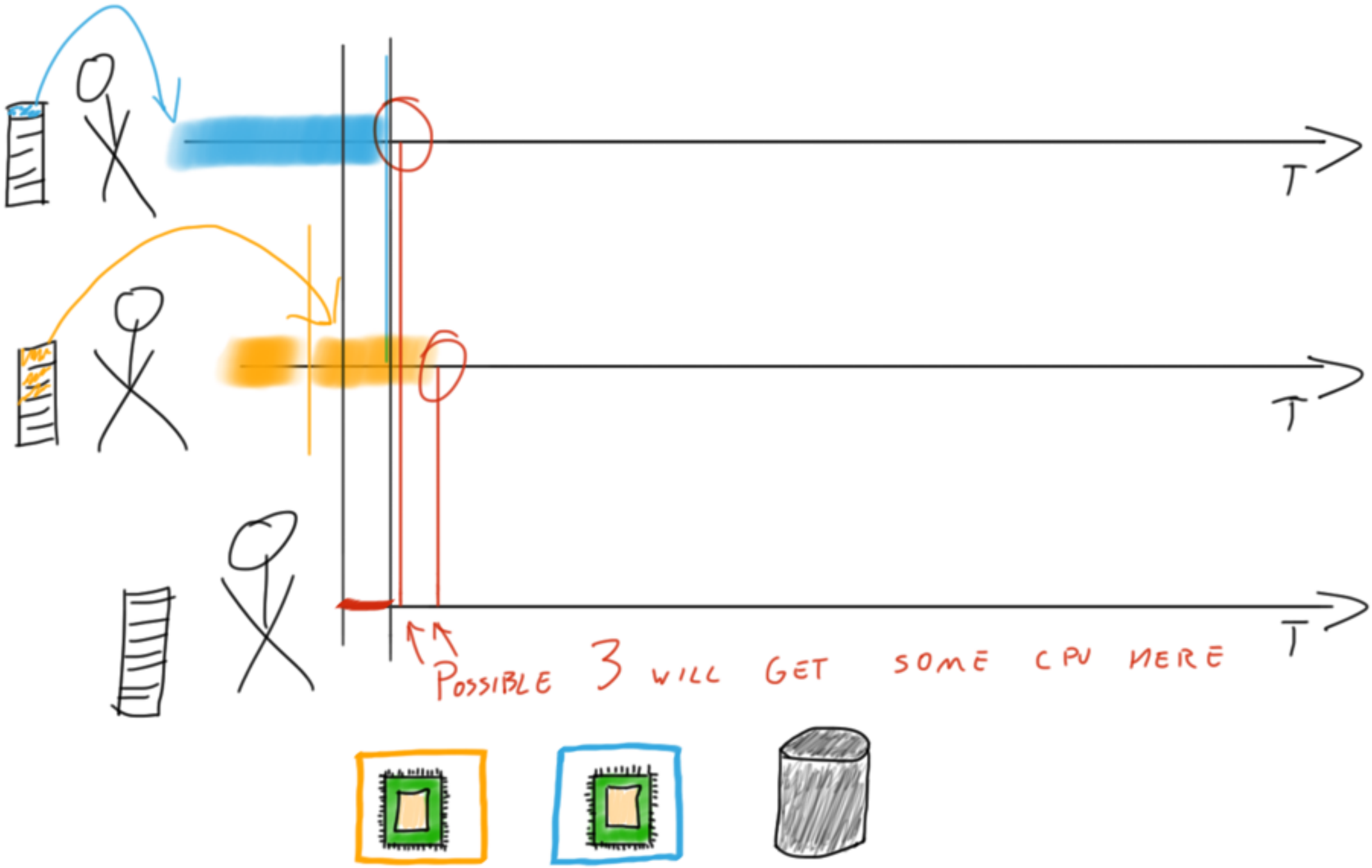
Async where it matters: Scheduling



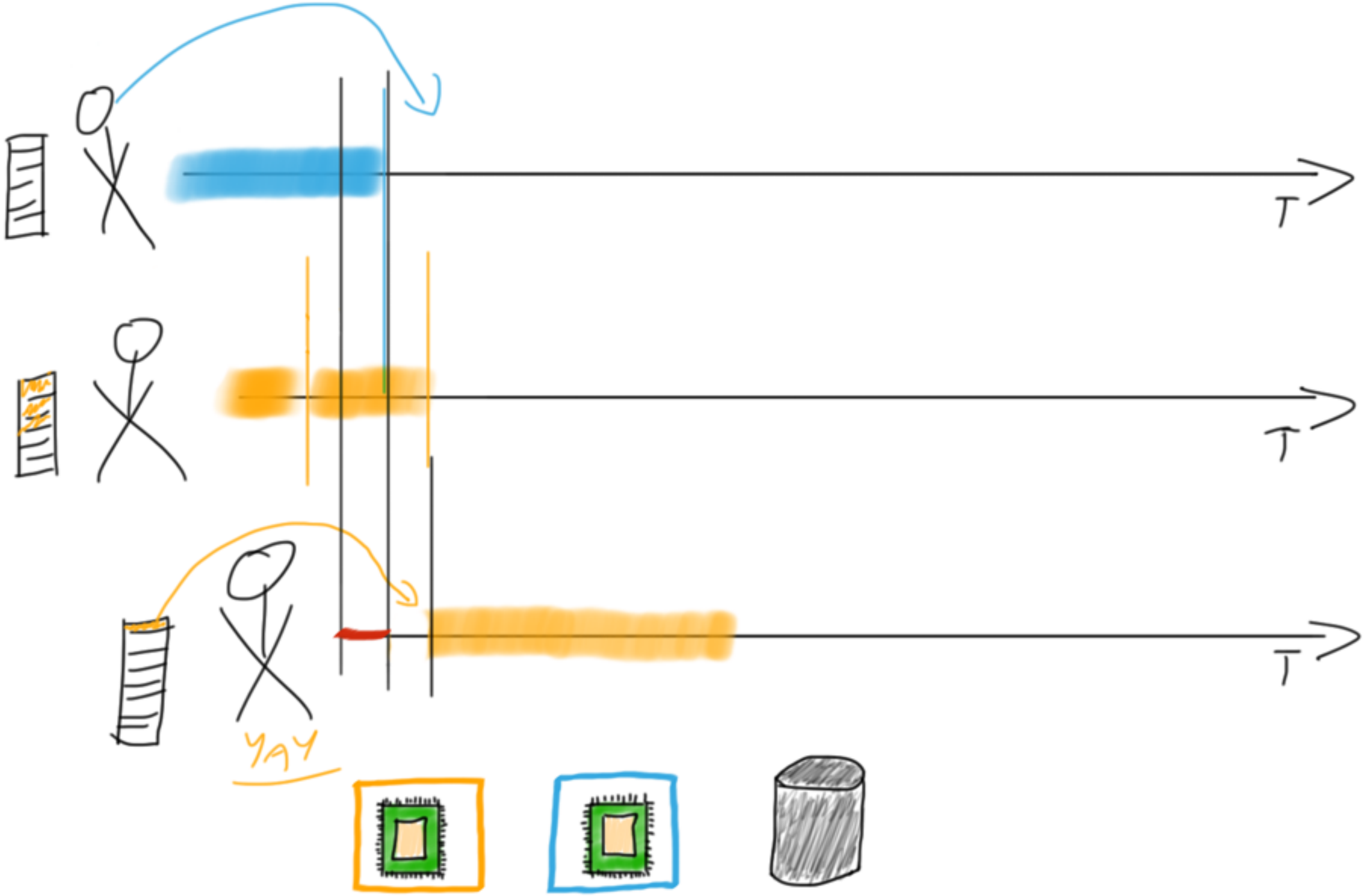
Async where it matters: Scheduling



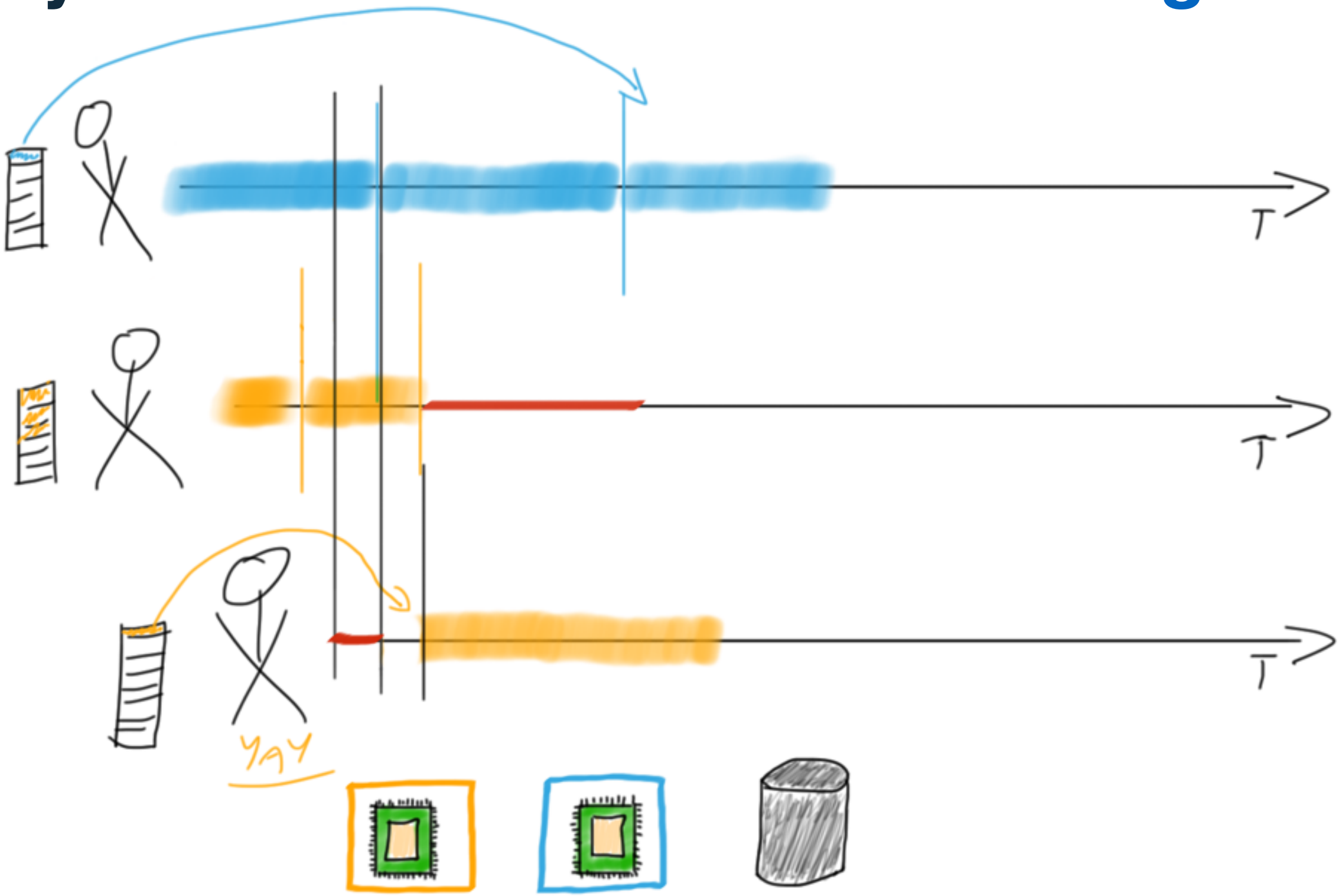
Async where it matters: Scheduling



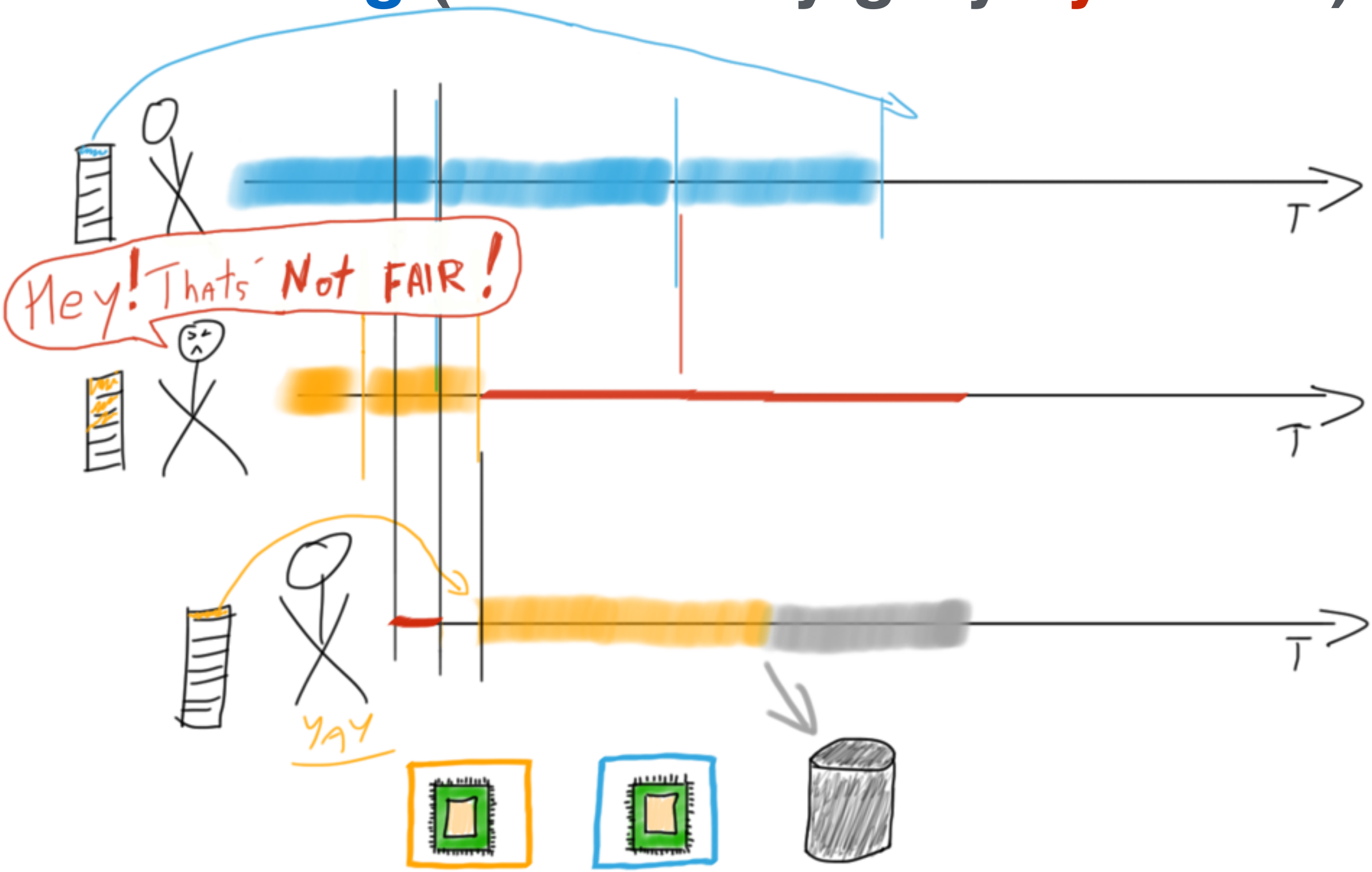
Async where it matters: Scheduling



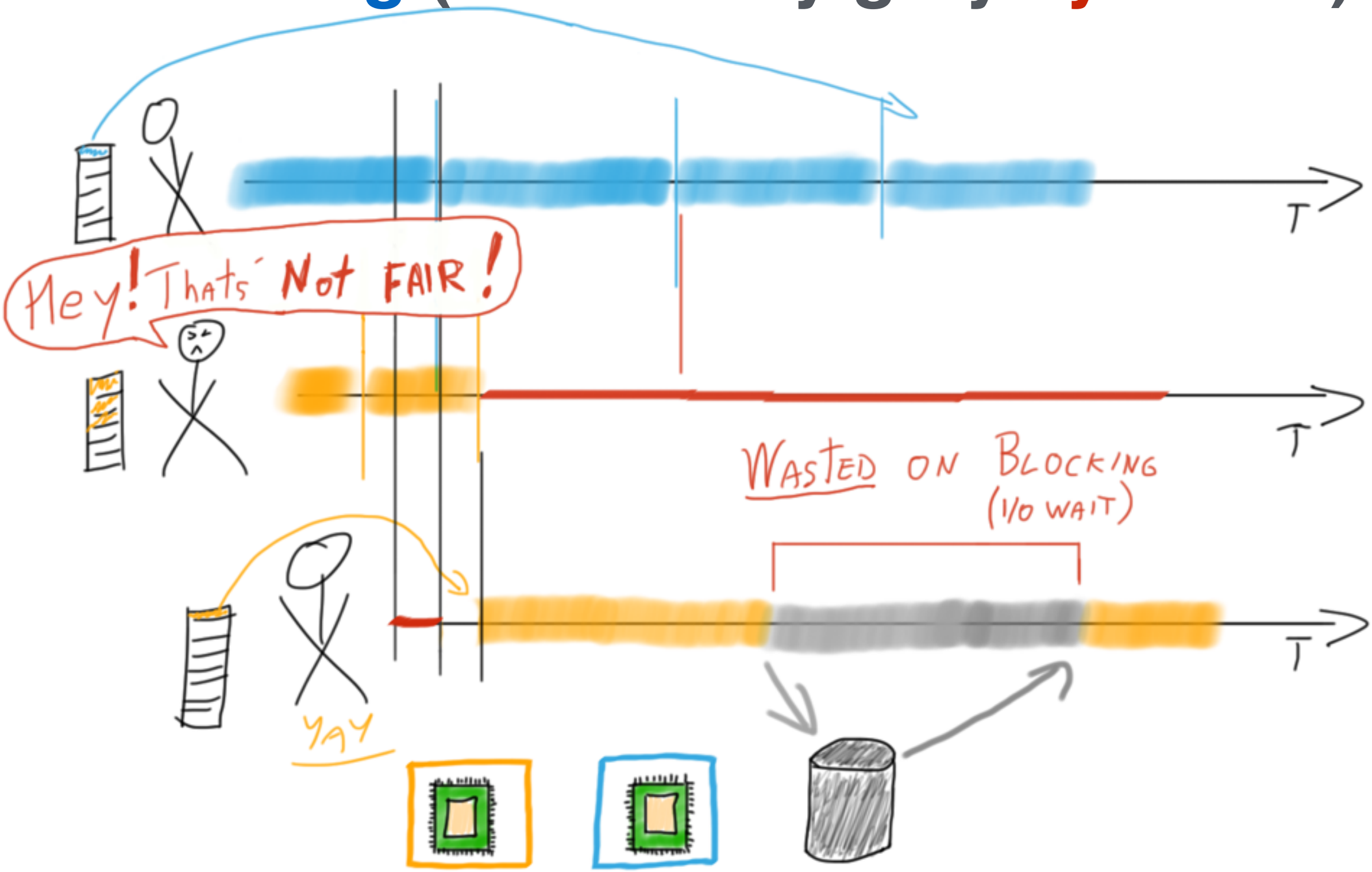
Async where it matters: Scheduling



Scheduling (notice they grey sync call)

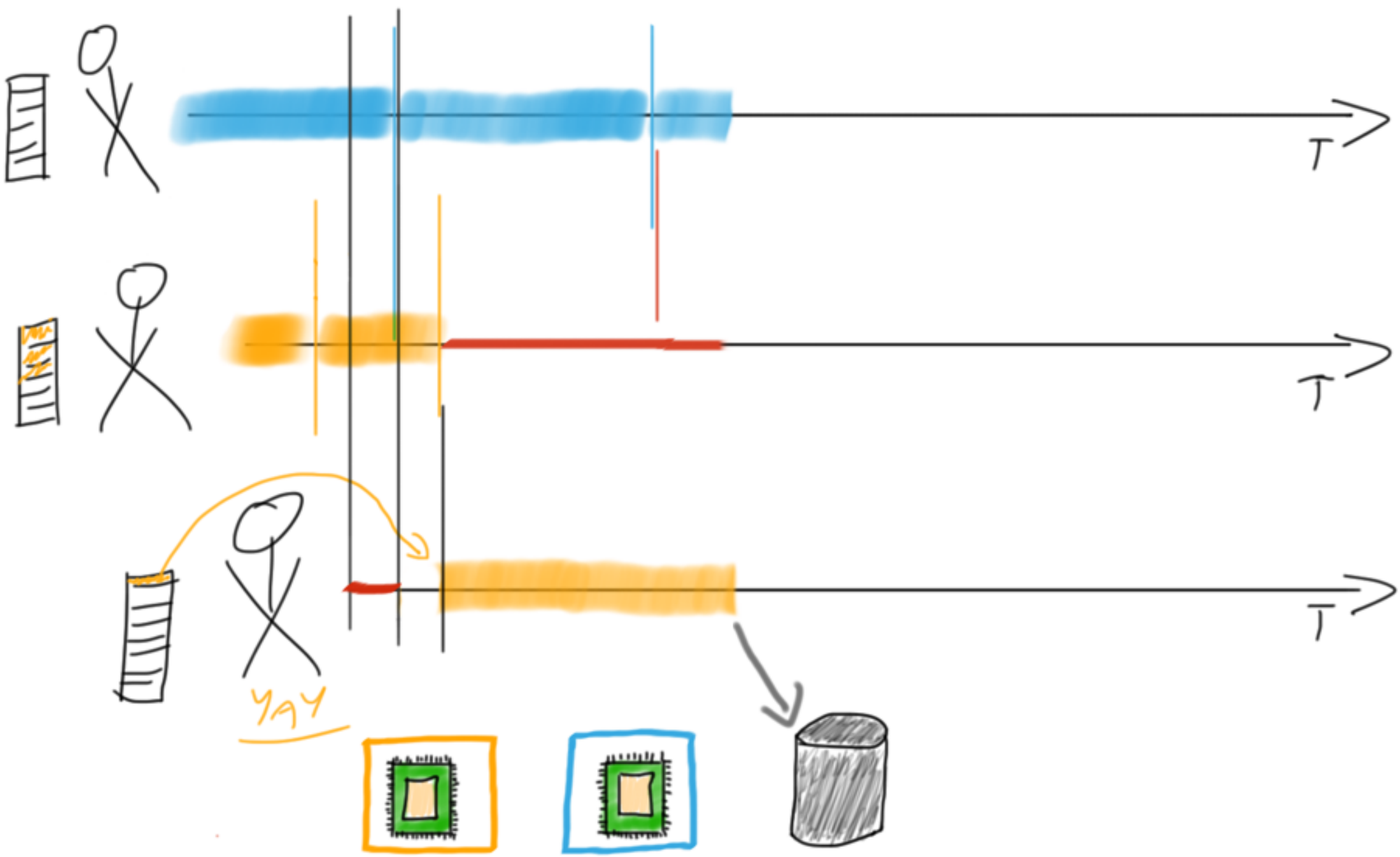


Scheduling (notice they grey sync call)

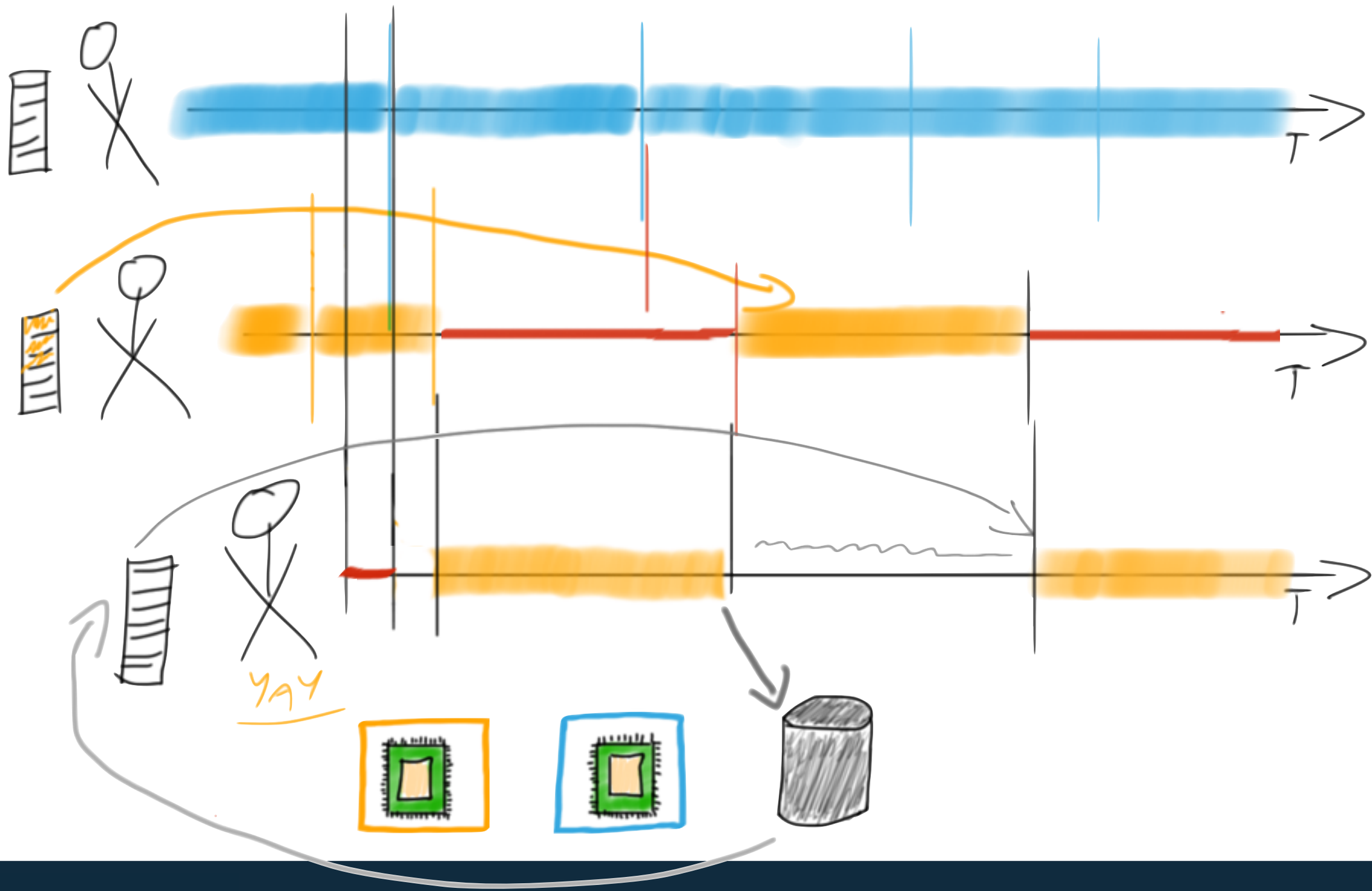


Scheduling (now with Async db call)

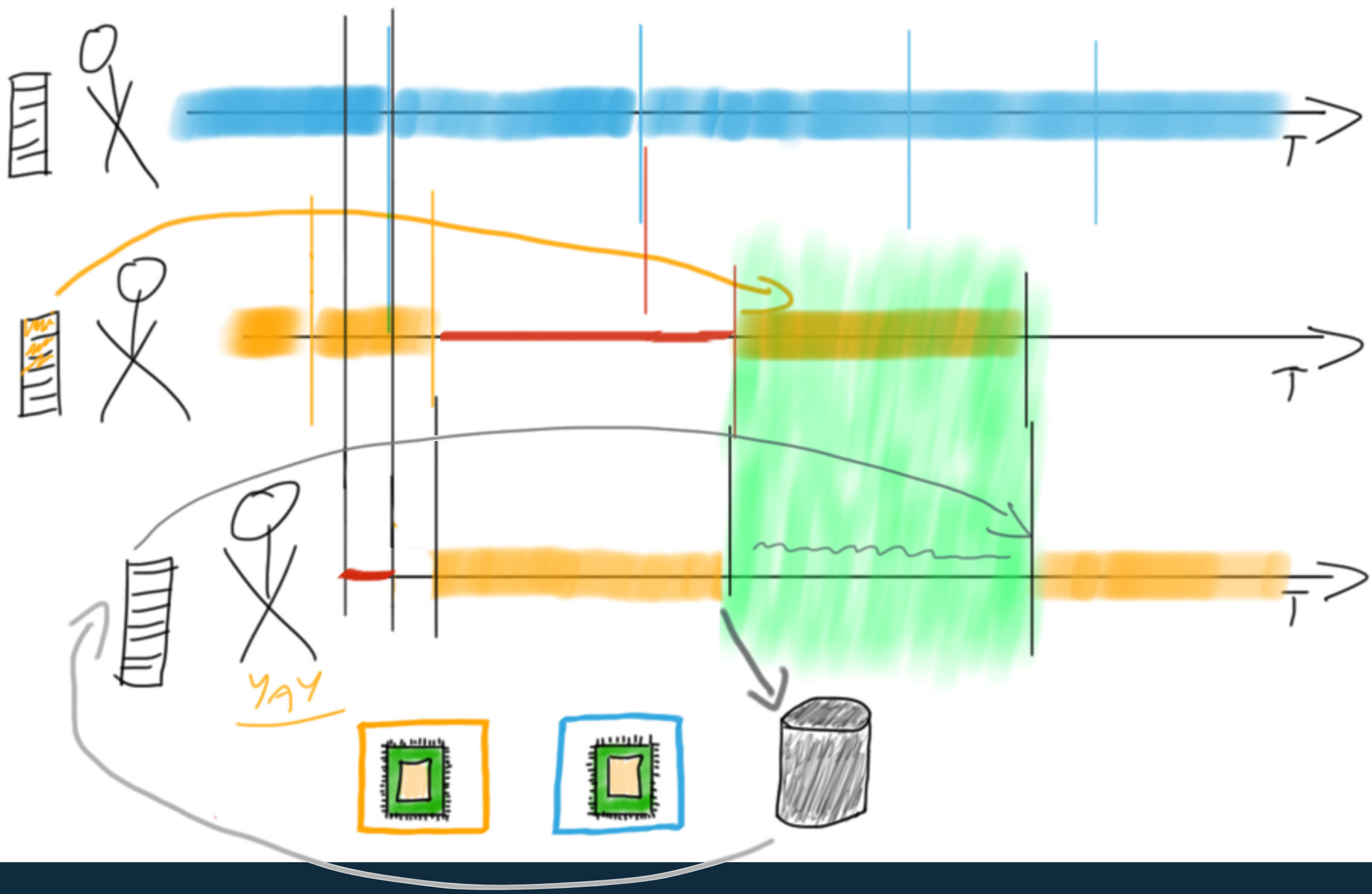
Scheduling (now with Async db call)



Scheduling (now with Async db call)



Scheduling (now with Async db call)



Latency

Latency

Time interval
between
the stimulation
and response.

Latency Quiz

Is **10s** latency acceptable in your app?
Is **200ms** latency acceptable?
How about **most** responses within **200ms**?
So mostly **20ms** and some ***1 minute*** latencies is OK?
Do people die when we go above **200ms**?

So 90% below **200ms**, 99% bellow **1s**, 99.99% below **2s**?

Gil Tene style, see: "How NOT to Measure Latency"

Latency in the "real world"

"Our response time is 200ms average,
stddev is around 60ms"

— a typical quote

Gil Tene style, see: "How NOT to Measure Latency"

Latency in the "real world"

"Our response time is 200ms average,
stddev is around 60ms"

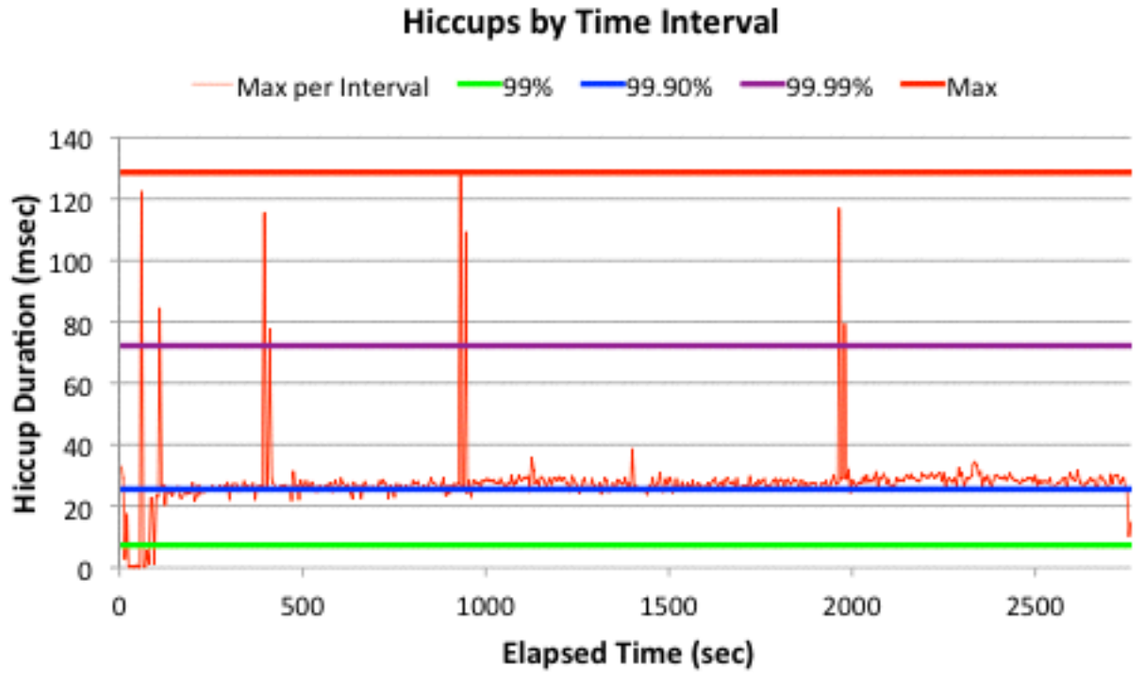
— a typical quote

"So yeah, our 99,99%' is…"

Latency does **NOT** behave like **normal distribution!**

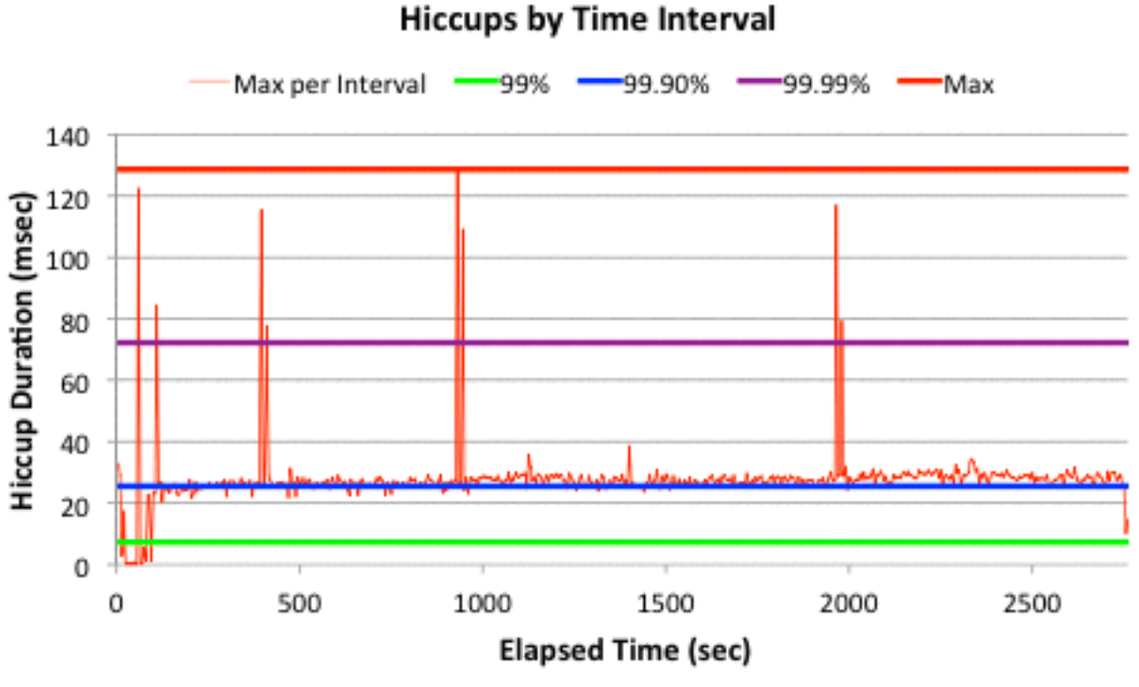
Gil Tene style, see: "How NOT to Measure Latency"

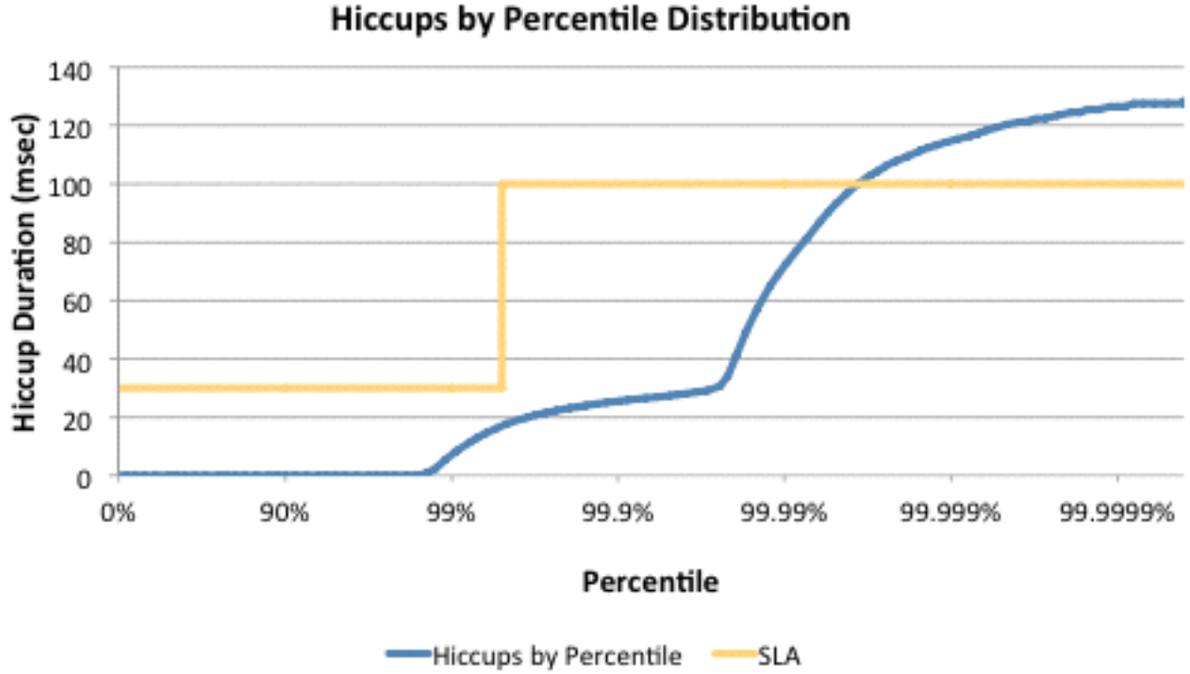
Hiccups



http://hdrhistogram.github.io/HdrHistogram/

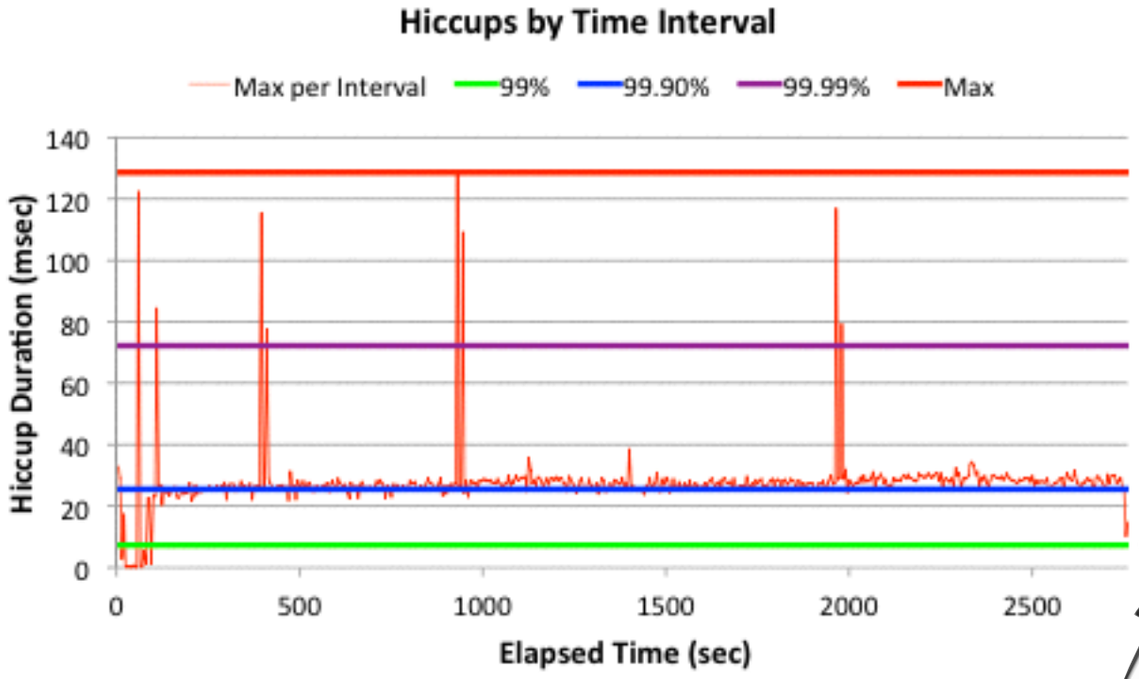
Hiccups

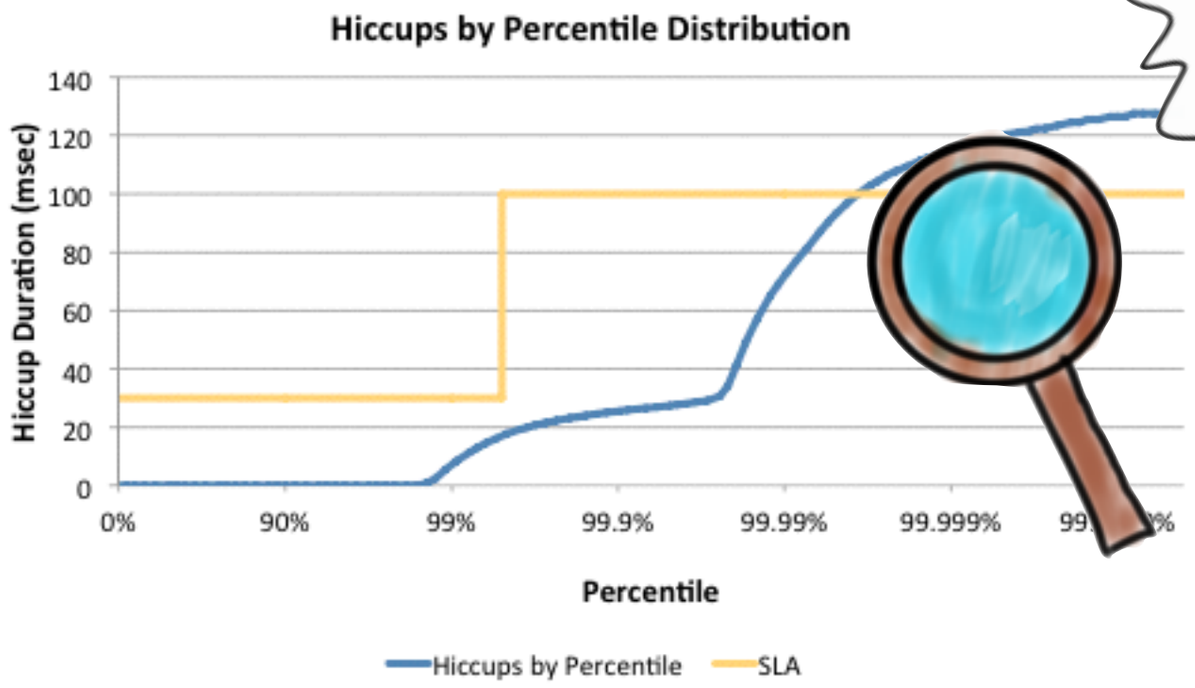




Gil Tene style, see: "How NOT to Measure Latency"

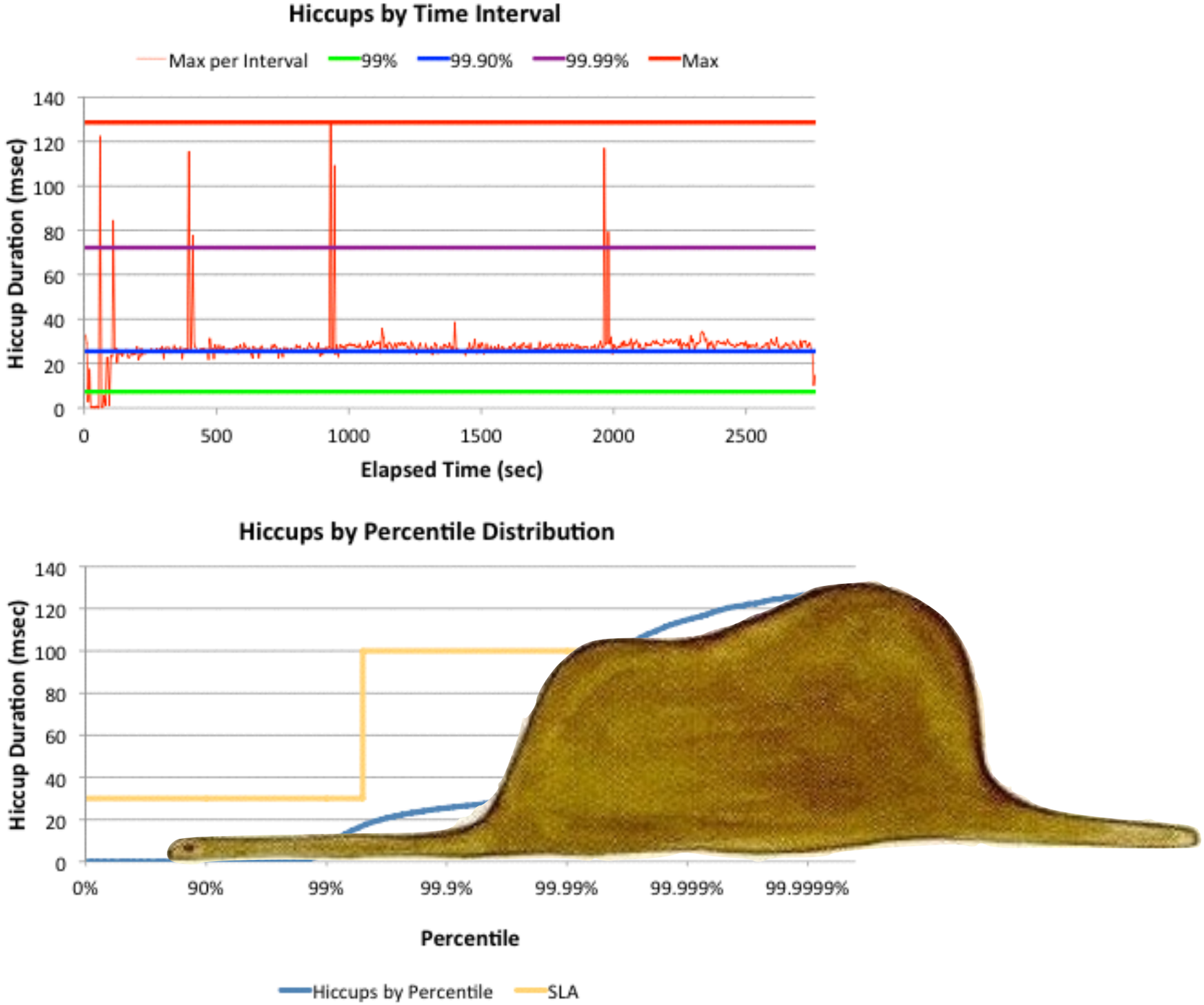
Hiccups





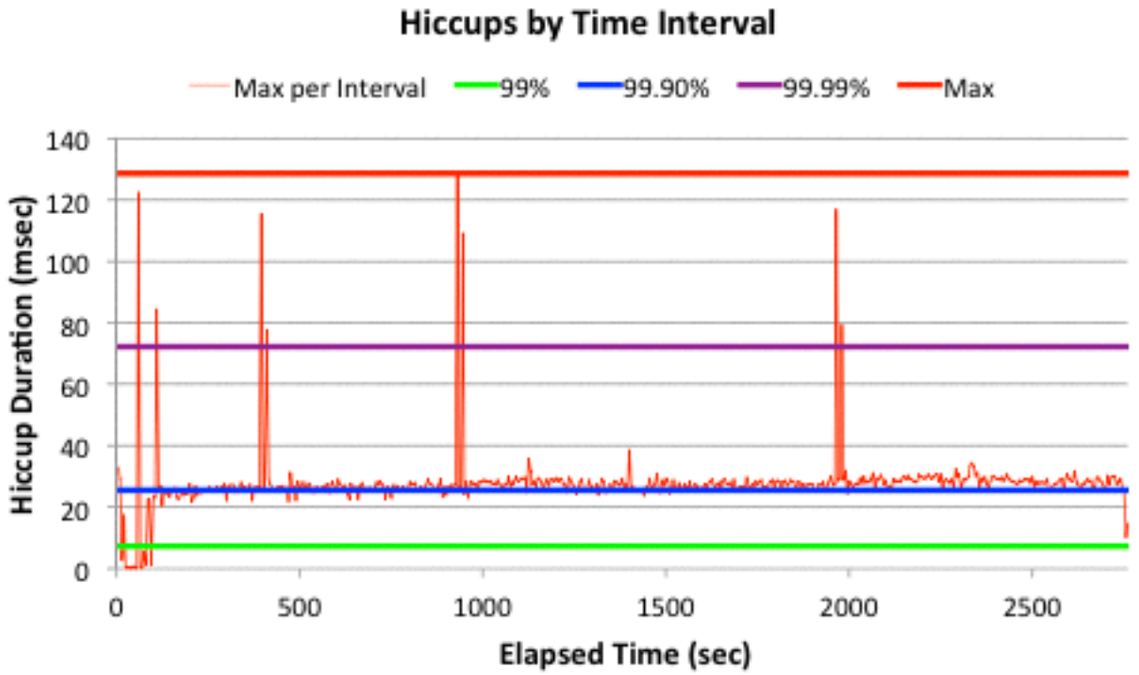
Gil Tene style, see: "How NOT to Measure Latency"

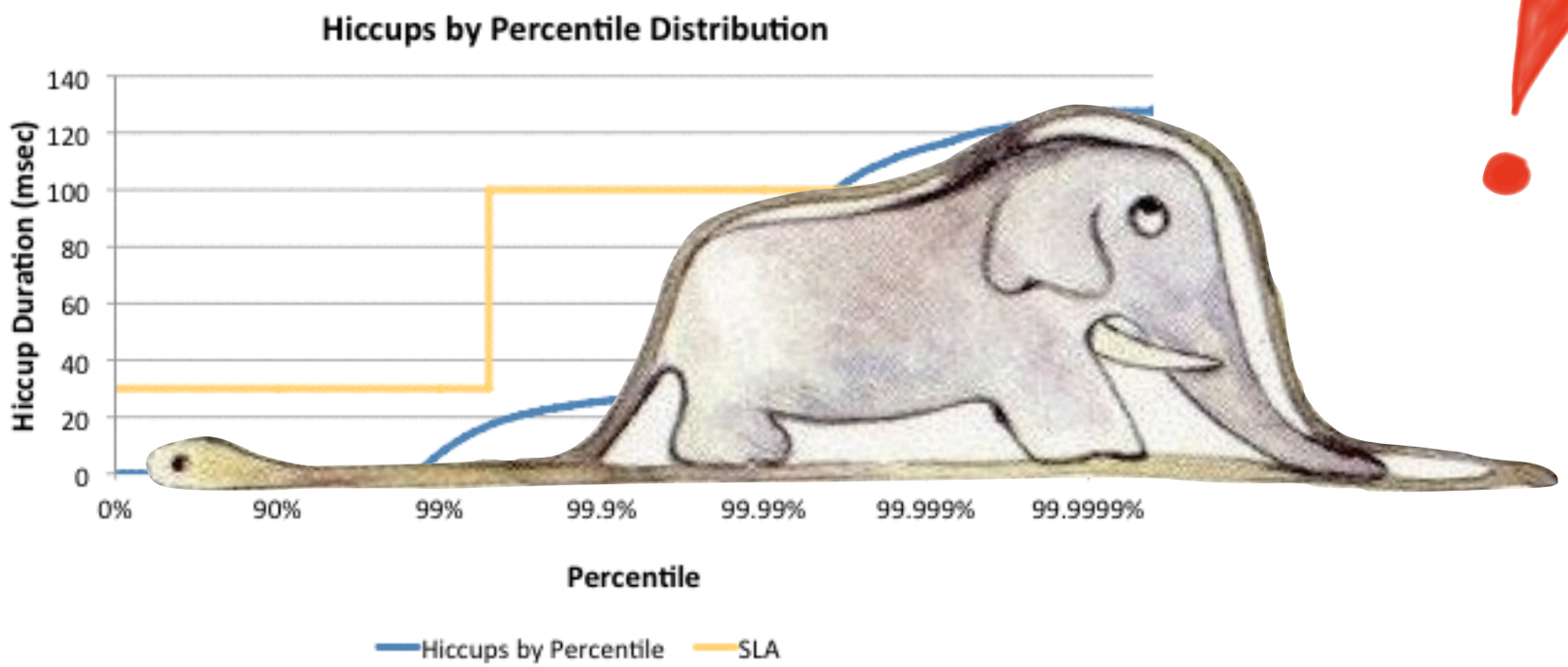
Hiccups



Gil Tene style, see: "How NOT to Measure Latency"

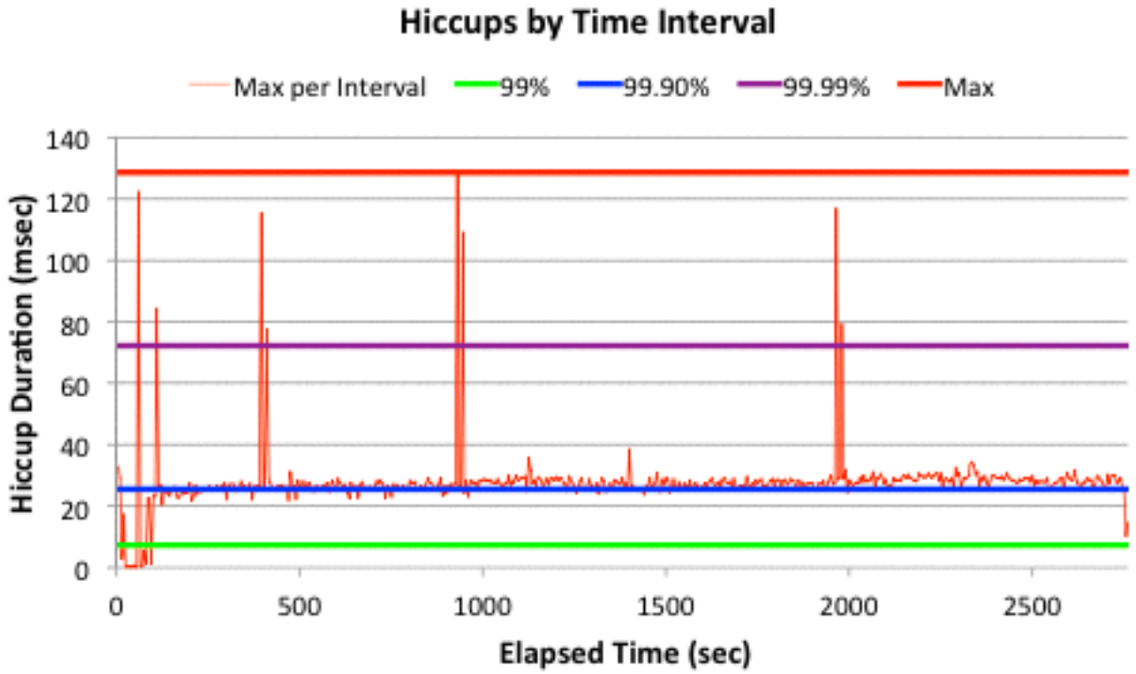
Hiccups

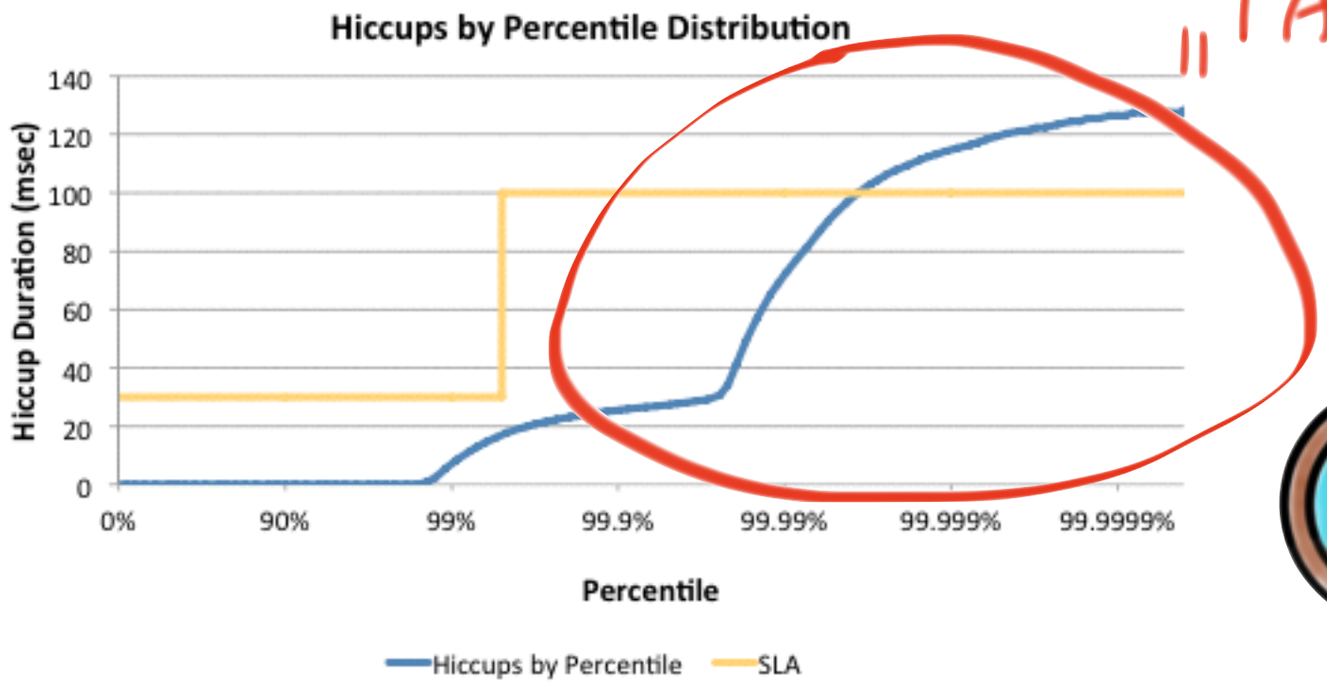




Gil Tene style, see: "How NOT to Measure Latency"

Hiccups





Gil Tene style, see: "How NOT to Measure Latency"

Concurrent

Concurrent < lock-free

Concurrent < lock-free < wait-free

Concurrent < lock-free < wait-free

"**concurrent**" data structure

Concurrent < lock-free < wait-free

What **can happen** in concurrent data structures:

```
A tries to write; B tries to write; B wins!
A tries to write; C tries to write; C wins!
A tries to write; D tries to write; D wins!
A tries to write; B tries to write; B wins!
A tries to write; E tries to write; E wins!
A tries to write; F tries to write; F wins!
```

…

Moral?

1) **Thread A** is a complete loser.
2) **Thread A may never make progress.**

Concurrent < lock-free < wait-free

What **can happen** in concurrent data structures:

```
A tries to write;  B tries to write;  B wins!
                   C tries to write;  C wins!
                   D tries to write;  D wins!
                   B tries to write;  B wins!
                   E tries to write;  E wins!
                   F tries to write;  F wins!
```

…

Moral?

1) **Thread A** is a complete loser.
2) **Thread A may never make progress.**

Concurrent < lock-free < wait-less

```
def offer(a: A): Boolean // returns on failure

def add(a: A): Unit      // throws on failure

def put(a: A): Boolean   // blocks until able to enqueue
```

Remember: **Concurrency** is NOT **Parallelism**.

Rob Pike - Concurrency is NOT Parallelism (video)

Concurrent < lock-free < wait-free

concurrent data structure

<

lock-free* data structure

* *lock-free a.k.a. lockless*

What lock-free programming looks like:

Concurrent < lock-free < wait-free

An algorithm is **lock-free** if it satisfies that:
When the program **threads** are **run sufficiently long**,
at **least one** of the threads **makes progress**.

Concurrent < lock-free < wait-free

```
class CASBackedQueue[A] {
  val _queue = new AtomicReference(Vector[A]())

  // does not block, may spin though
  @tailrec final def put(a: A): Unit = {
    val queue = _queue.get
    val appended = queue :+ a
    if (!_queue.compareAndSet(queue, appended))
      put(a)
  }
}
```

** Both versions are used: lock-free / lockless*

Concurrent < lock-free < wait-free

```
class CASBackedQueue[A] {
  val _queue = new AtomicReference(Vector[A]())

  // does not block, may spin though
  @tailrec final def put(a: A): Unit = {
    val queue = _queue.get
    val appended = queue :+ a
    if (!_queue.compareAndSet(queue, appended))
      put(a)
  }
}
```

CMPXCHG

Compare and Exchange

Opcode	Mnemonic	Description
0F B0 /r	CMPXCHG r/m8,r8	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
0F B1 /r	CMPXCHG r/m16,r16	Compare AX with r/m16. If equal, ZF is set and r16 is loaded into r/m16. Else, clear ZF and load r/m16 into AX
0F B1 /r	CMPXCHG r/m32,r32	Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into EAX

Concurrent < lock-free < wait-free

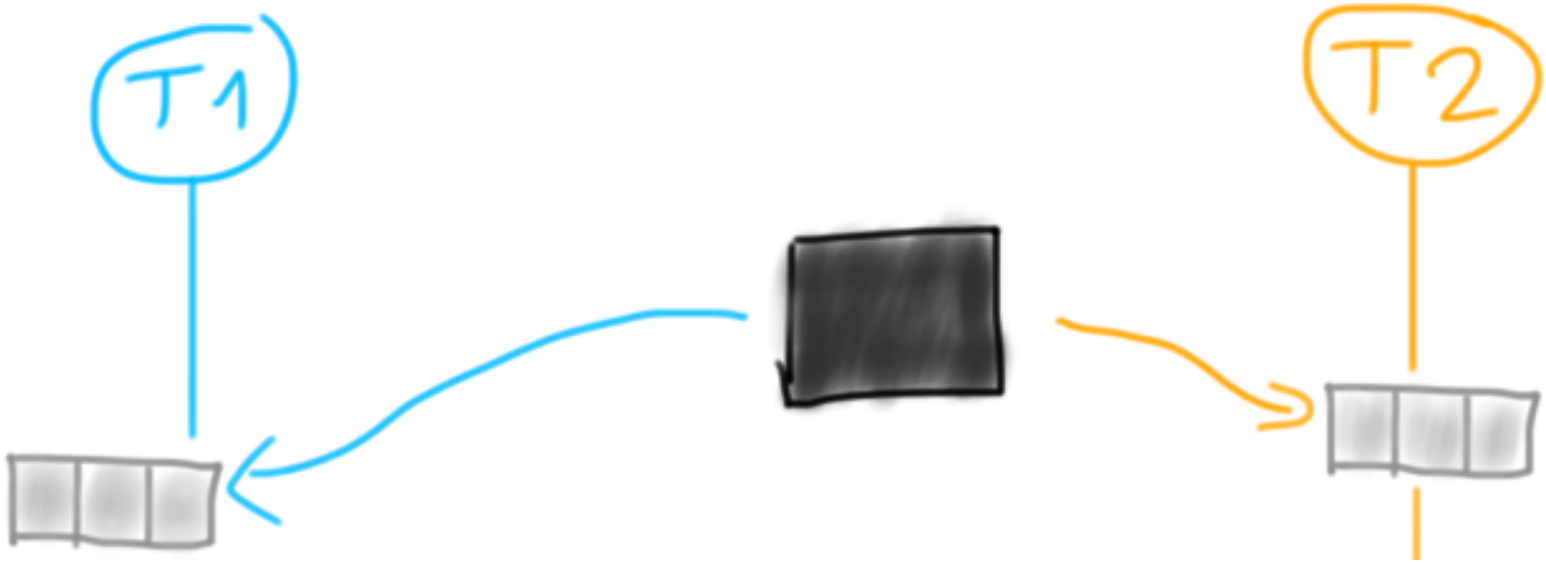
```
class CASBackedQueue[A] {
  val _queue = new AtomicReference(Vector[A]())

  // does not block, may spin though
  @tailrec final def put(a: A): Unit = {
    val queue = _queue.get
    val appended = queue :+ a

    if (!_queue.compareAndSet(queue, appended))
      put(a)
  }
}
```

CMPXCHG

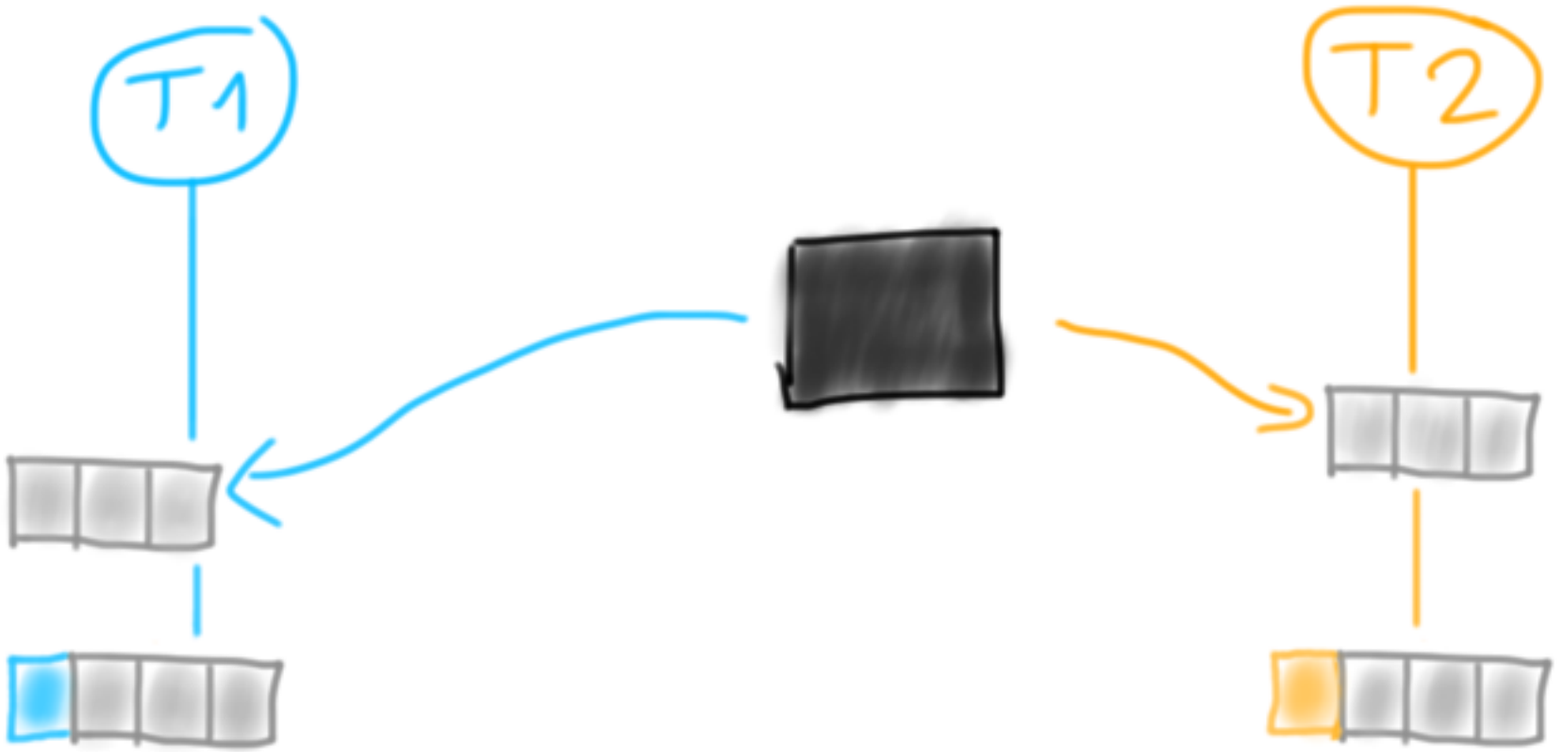
Compare and Exchange

Opcode	Mnemonic	Description
0F B0 /r	CMPXCHG r/m8,r8	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
0F B1 /r	CMPXCHG r/m16,r16	Compare AX with r/m16. If equal, ZF is set and r16 is loaded into r/m16. Else, clear ZF and load r/m16 into AX
0F B1 /r	CMPXCHG r/m32,r32	Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into EAX

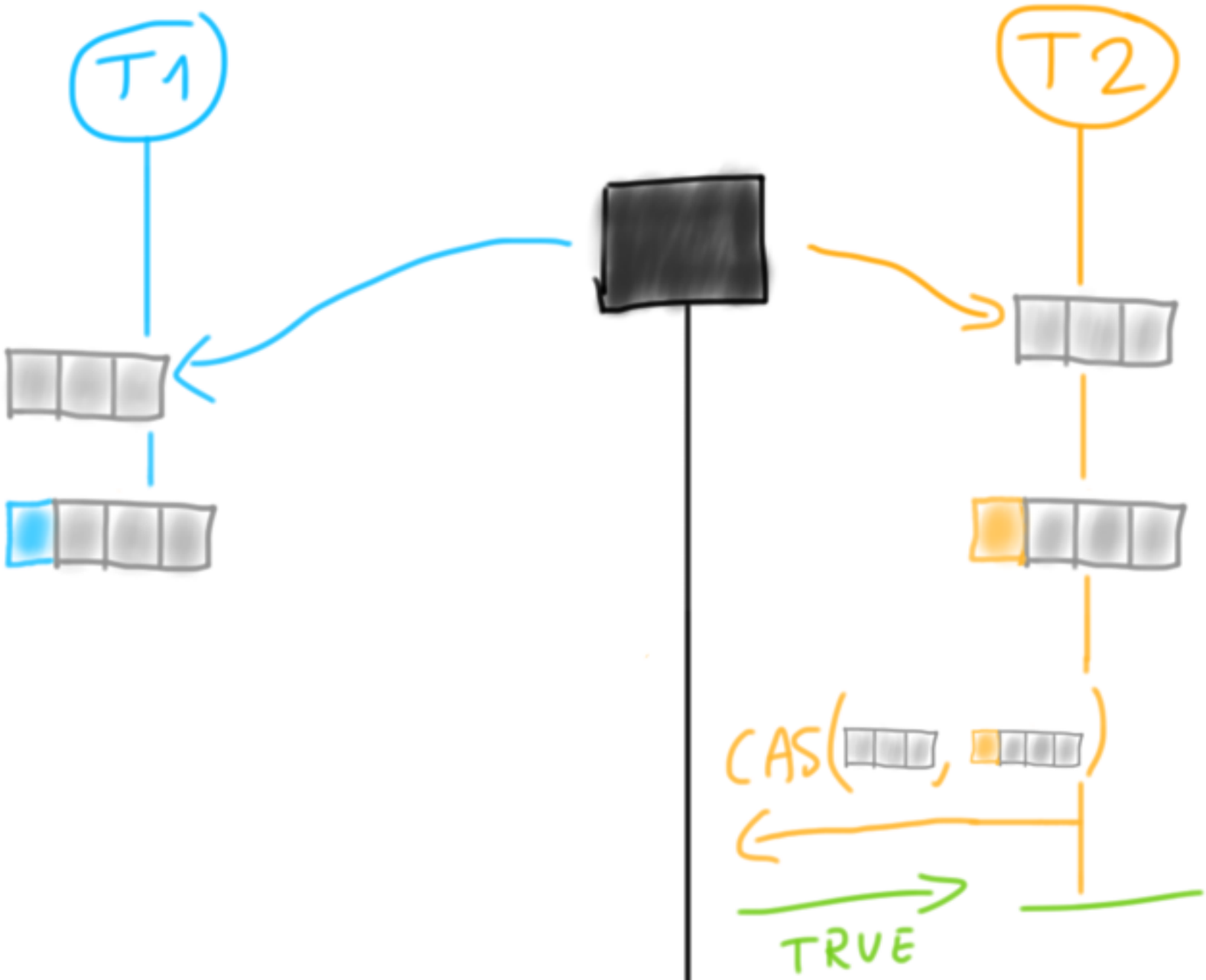
Concurrent < lock-free < wait-free



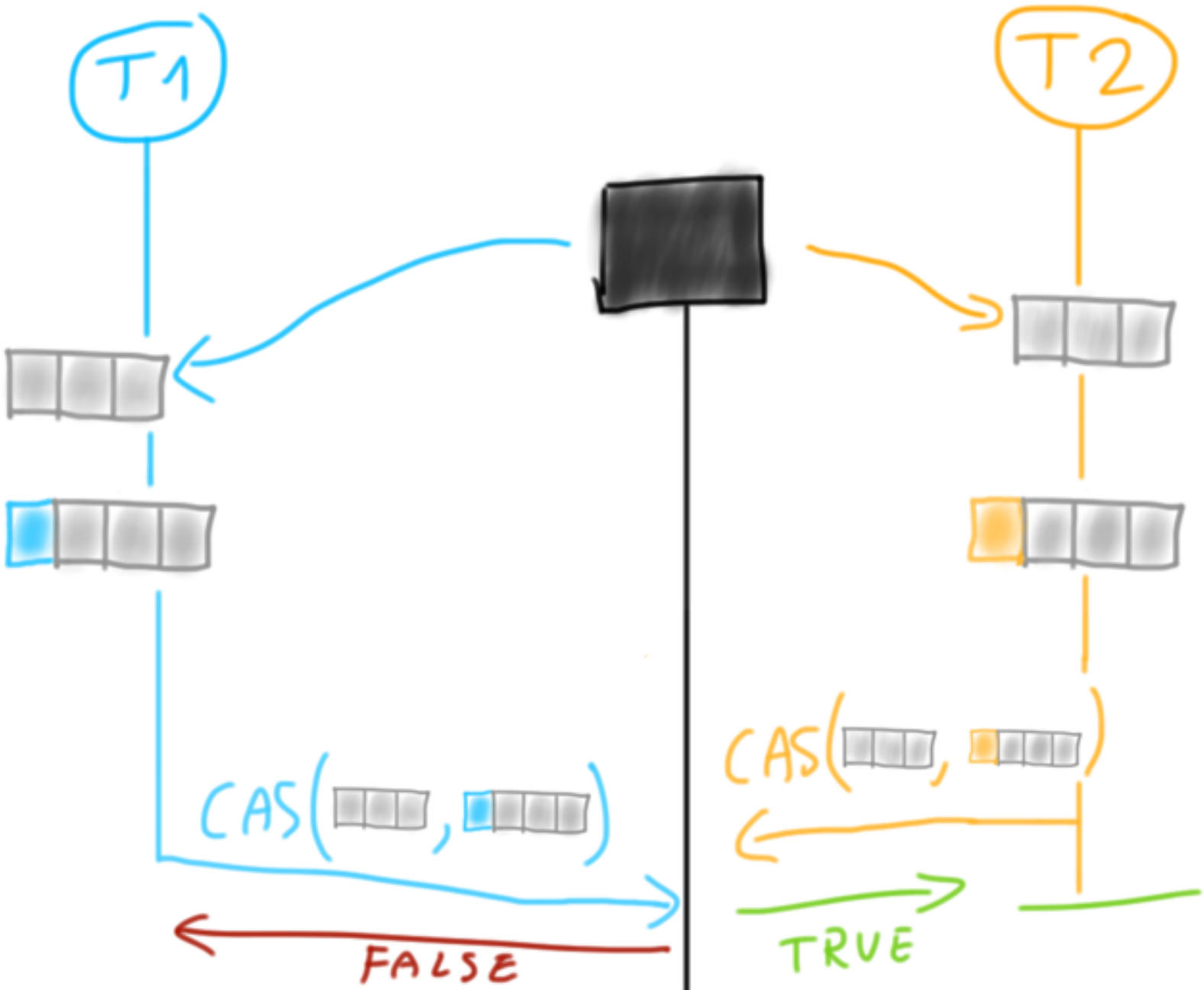
Concurrent < lock-free < wait-free



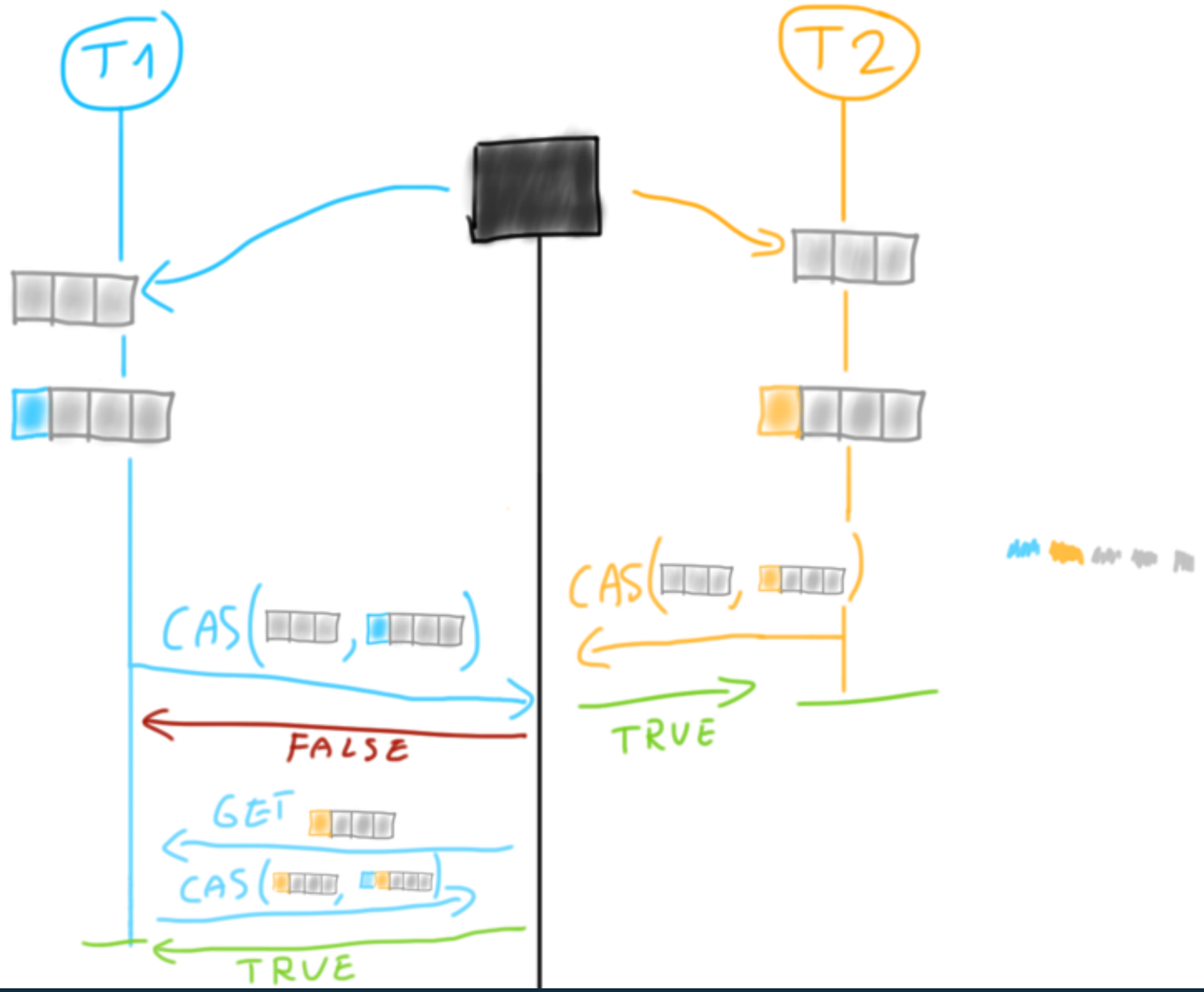
Concurrent < lock-free < wait-free



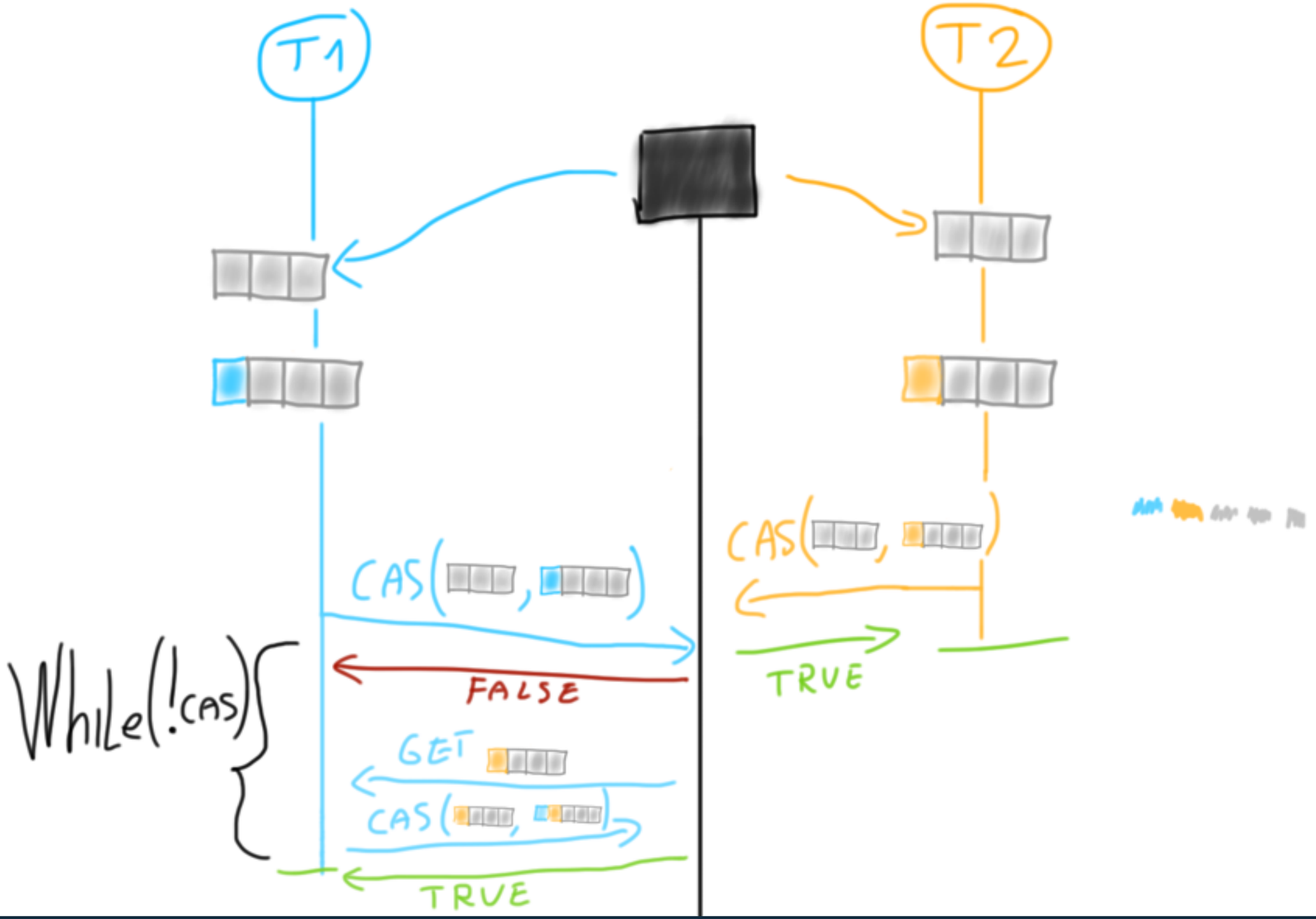
Concurrent < lock-free < wait-free



Concurrent < lock-free < wait-free



Concurrent < lock-free < wait-free



Concurrent < lock-free < wait-free

"**concurrent**" data structure

<

lock-free* data structure

<

wait-free data structure

** Both versions are used: lock-free / lockless*

Concurrent < lock-free < wait-free

An algorithm is **wait-free** if **every** operation has a **bound** on the **number of steps** the algorithm will take before the operation completes.

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
Maged M. Michael Michael L. Scott

wait-free: j.u.c.ConcurrentLinkedQueue

This is a modification of the Michael & Scott algorithm, adapted for a garbage-collected environment, with support for interior node deletion (to support remove(Object)).

For explanation, read the paper.

```
public boolean offer(E e) {
  checkNotNull(e);
  final Node<E> newNode = new Node

  for (Node<E> t = tail, p = t;;)
    Node<E> q = p.next;
    if (q == null) {
      // p is last node
      if (p.casNext(null, newNode)) {
        // Successful CAS is the linearization point
        // for e to become an element of this queue,
        // and for newNode to become "live".
        if (p != t) // hop two nodes at a time
          casTail(t, newNode);  // Failure is OK.
          return true;
        }
        // Lost CAS race to another thread; re-read next
      }
      else if (p == q)
        // We have fallen off list.  If tail is unchanged, it
        // will also be off-list, in which case we need to
        // jump to head, from which all live nodes are always
        // reachable.  Else the new tail is a better bet.
        p = (t != (t = tail)) ? t : head;
      else
        // Check for tail updates after two hops.
        p = (p != t && t != (t = tail)) ? t : q;
  }
}
```

IO / AIO

IO / AIO / NIO

IO / AIO / NIO / Zero

Synchronous I / O

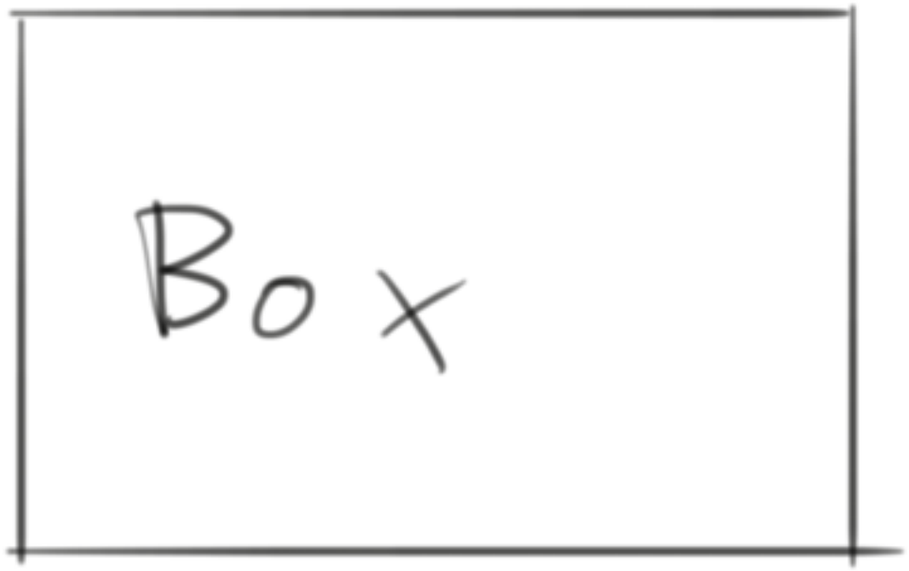
When I learned J2EE about 2008 with some of my desktop colleagues our reactions included something like:

"wtf is this sync IO crap, where is the main loop?!" :-)

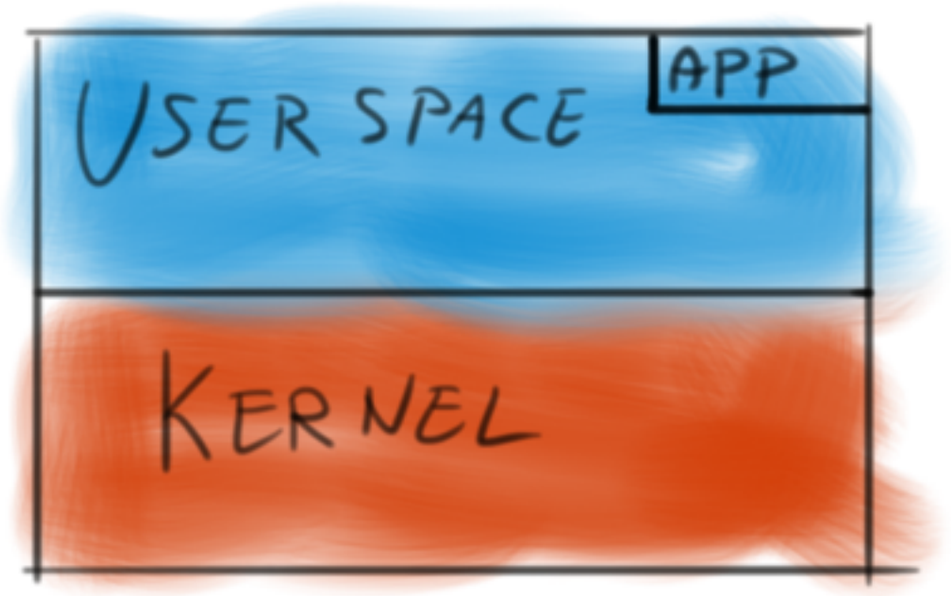
— Havoc Pennington
(HAL, GNOME, GConf, D-BUS, now Typesafe)

Interruption!
CPU: User Mode / Kernel Mode

Kernels and CPUs

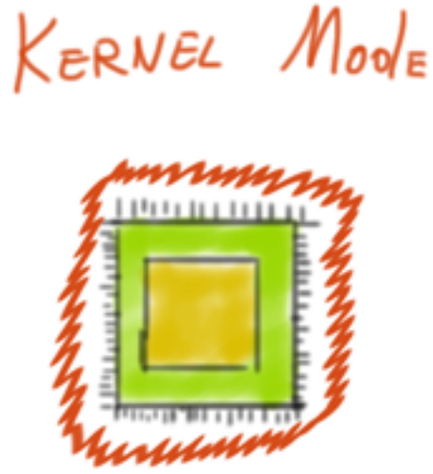


Kernels and CPUs

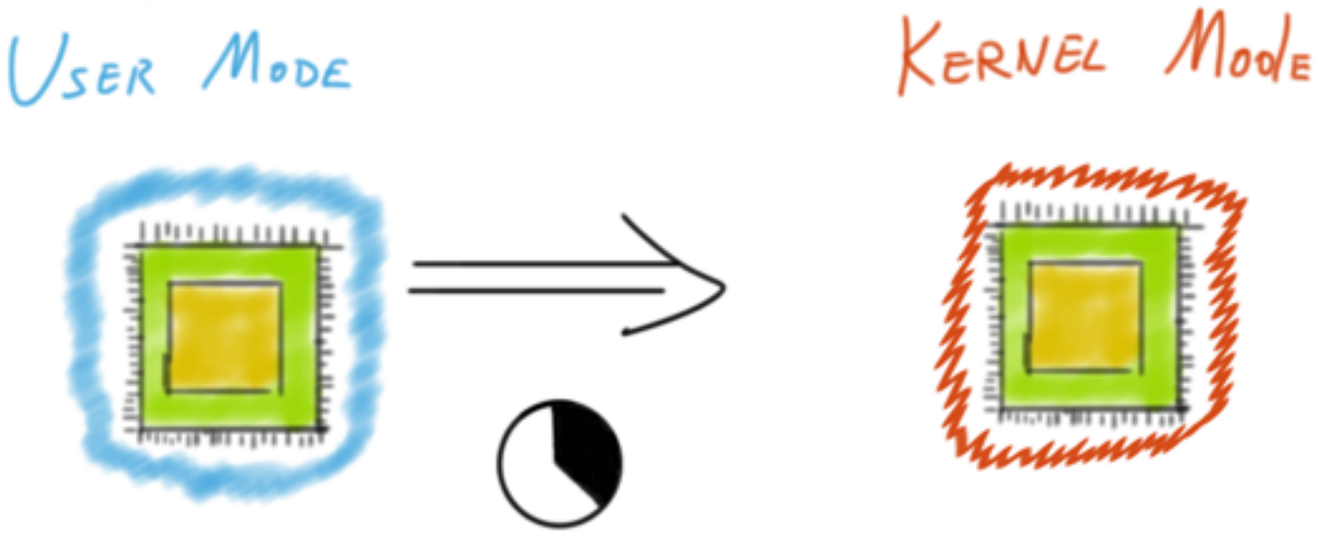


Kernels and CPUs

Kernels and CPUs

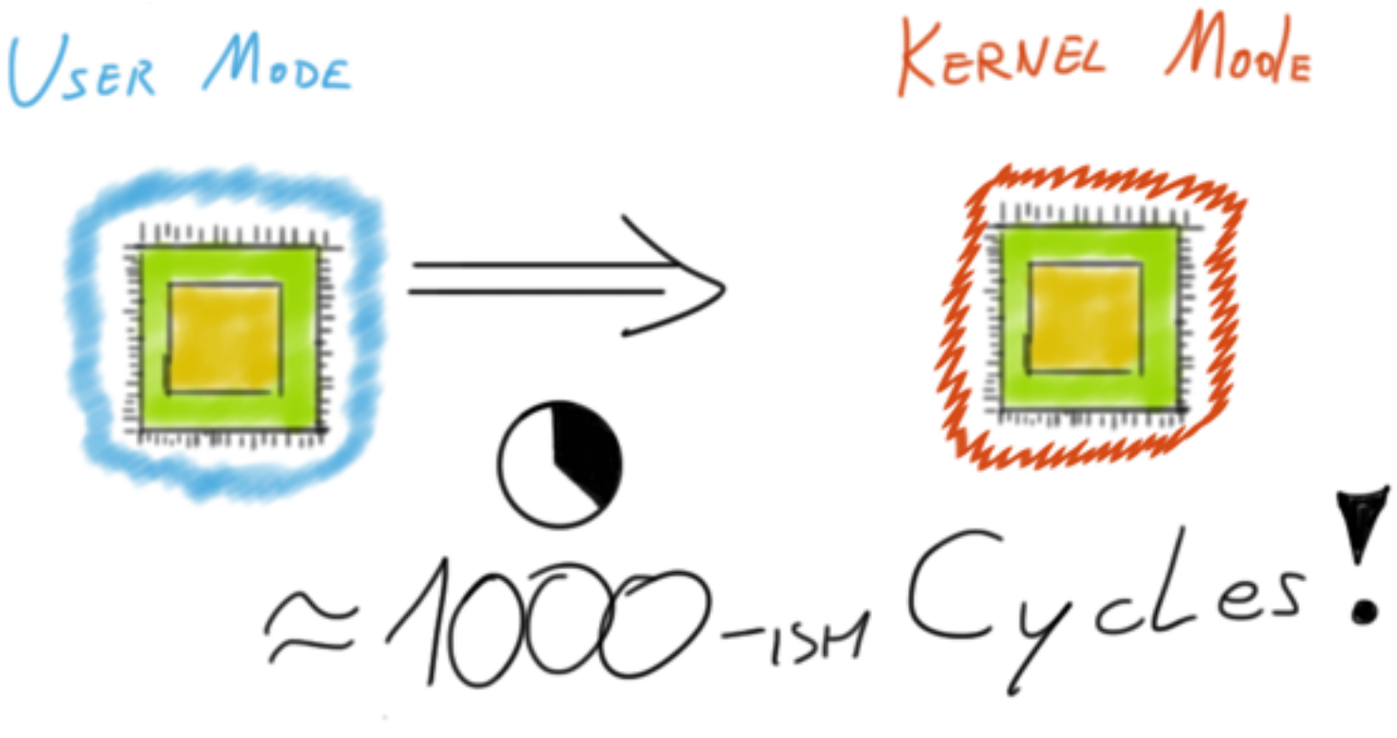


Kernels and CPUs



http://wiki.osdev.org/Context_Switching

Kernels and CPUs



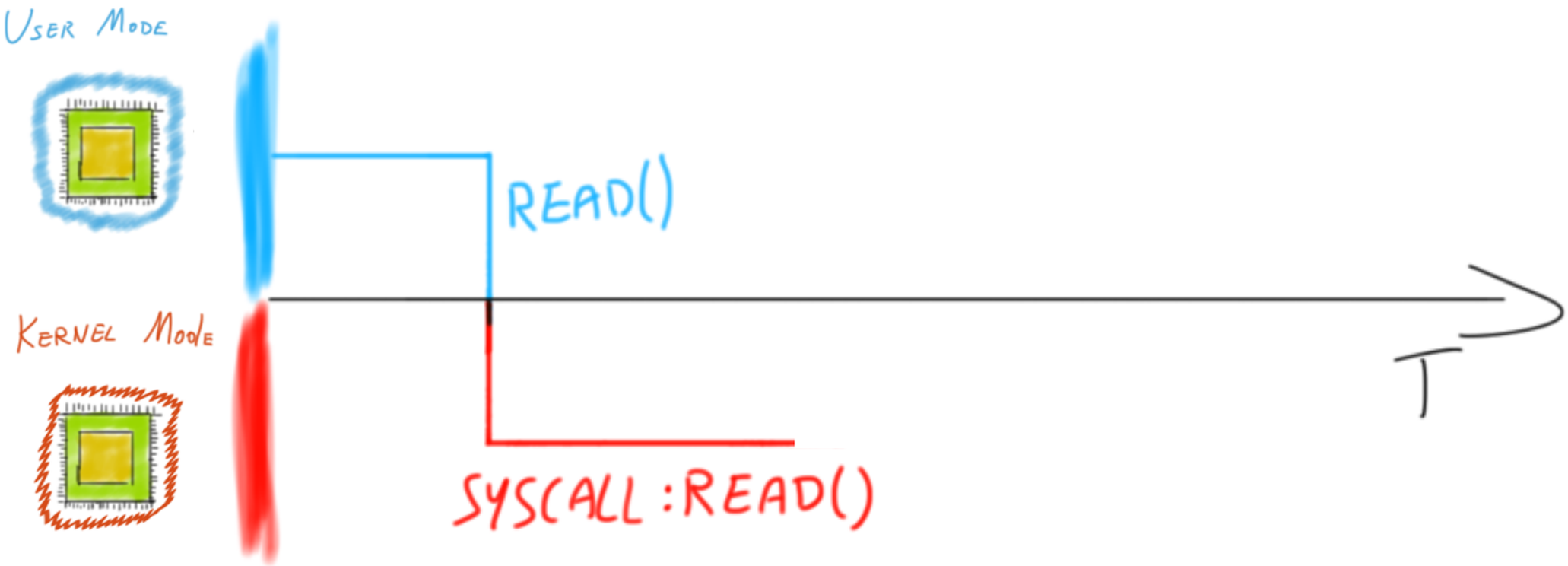
[…] switching from **user-level** to **kernel-level**
on a (2.8 GHz) **P4** is **1348 cycles**.

[…] Counting **actual time**, the **P4** takes **481ns** […]

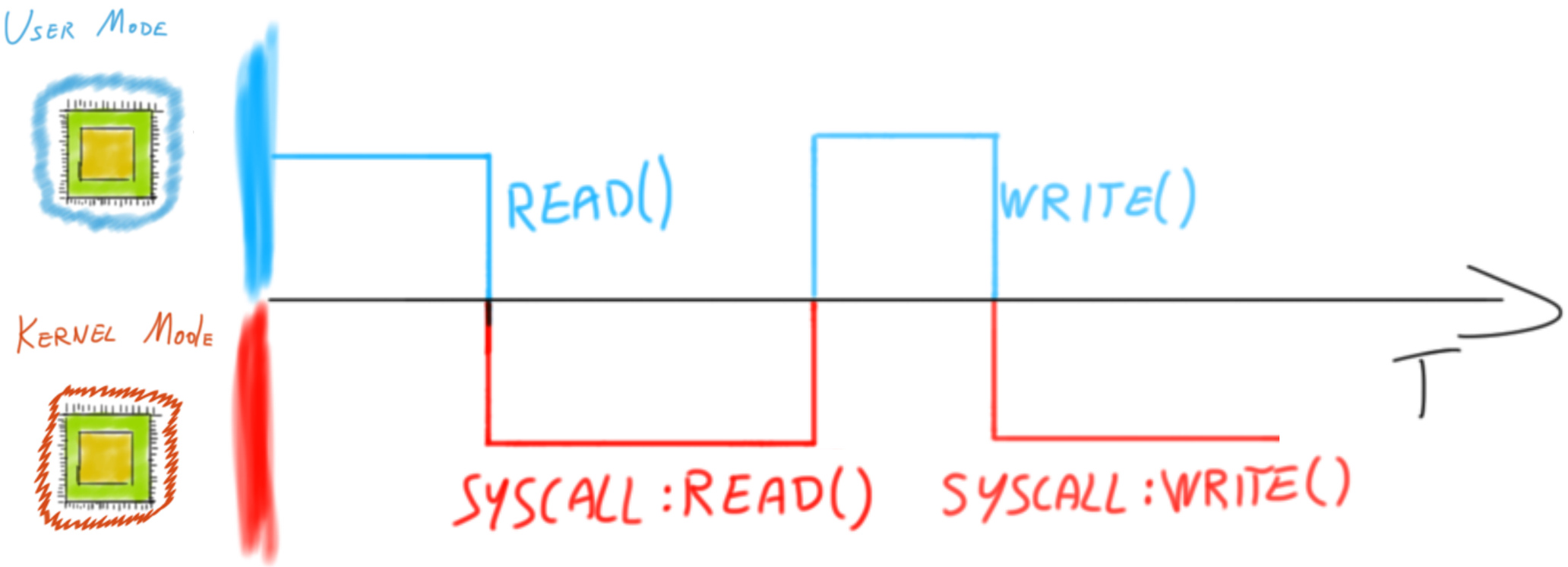
http://wiki.osdev.org/Context_Switching

I / O

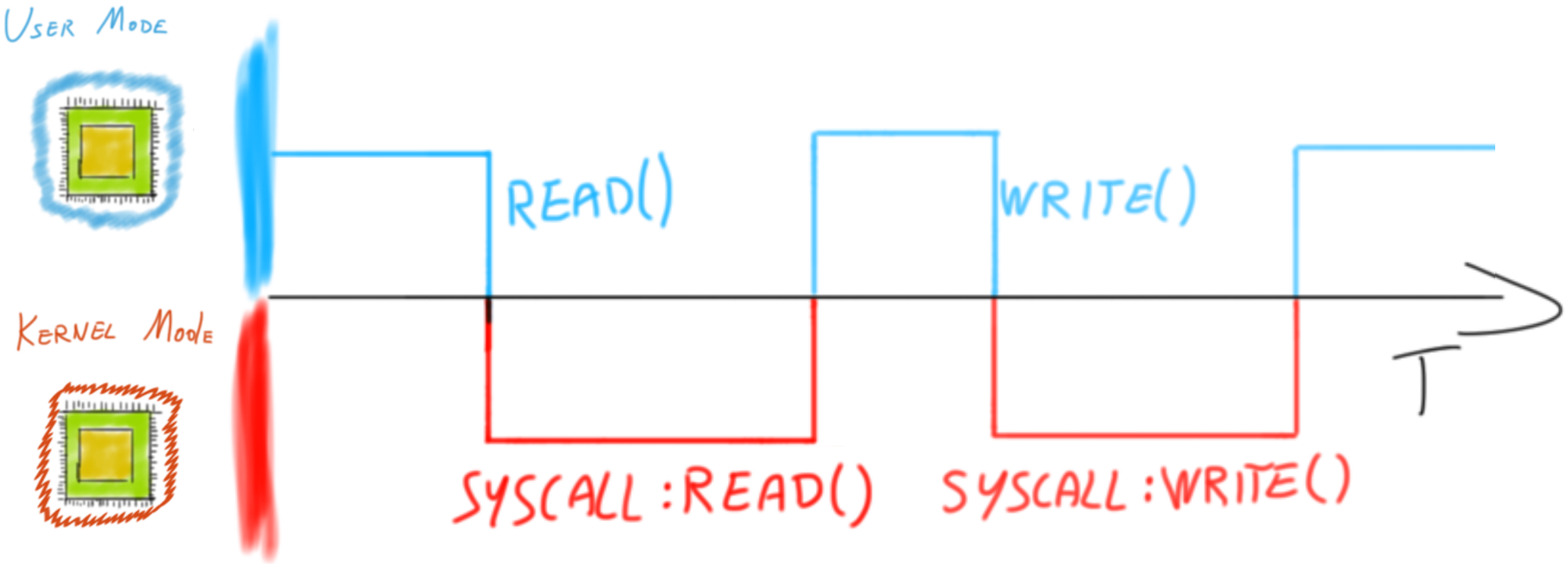
I / O



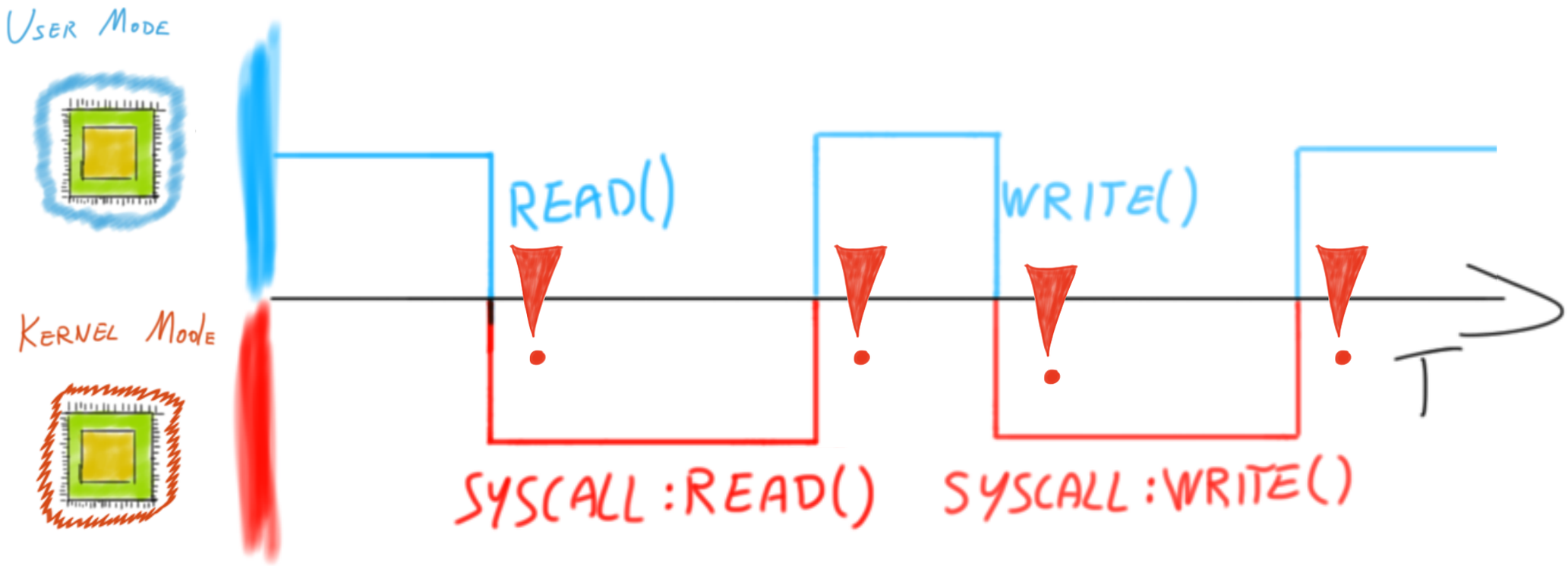
I / O


User Mode
Kernel Mode
READ()
WRITE()
T
SYSCALL:READ()
SYSCALL:WRITE()

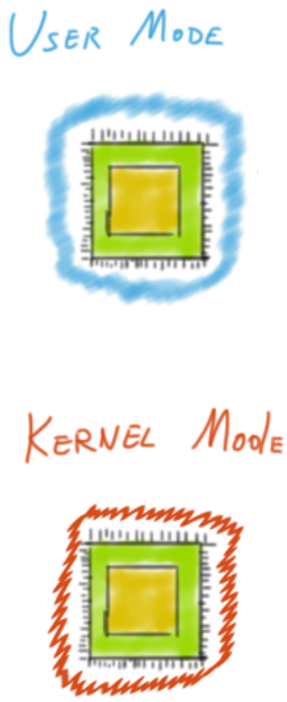
I / O


User Mode
Kernel Mode
READ()
WRITE()
T
SYSCALL:READ()
SYSCALL:WRITE()

I / O


User Mode
Kernel Mode
READ()
WRITE()
SYSCALL:READ()
SYSCALL:WRITE()
T

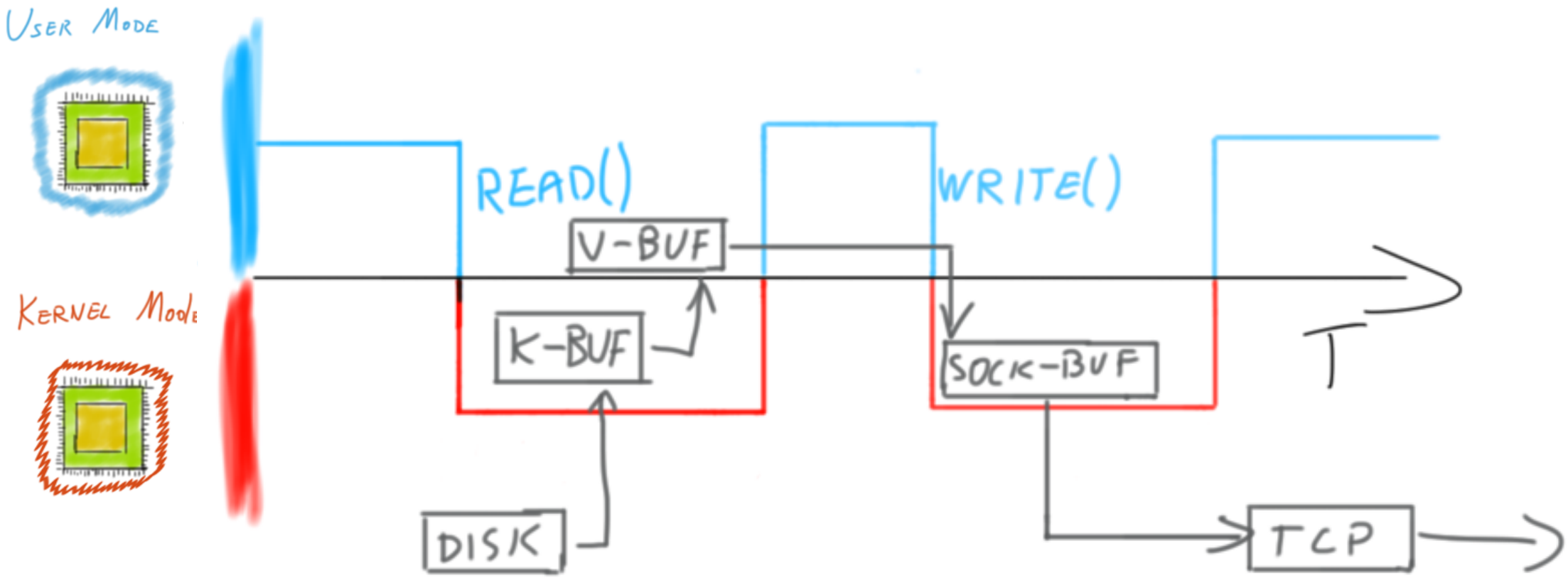
I/O



"Don't worry.
It only gets worse!"

I / O

"Don't worry.
It only gets worse!"



4 mode switches!

Same data in 3 buffers!

Asynchronous I / O [Linux]

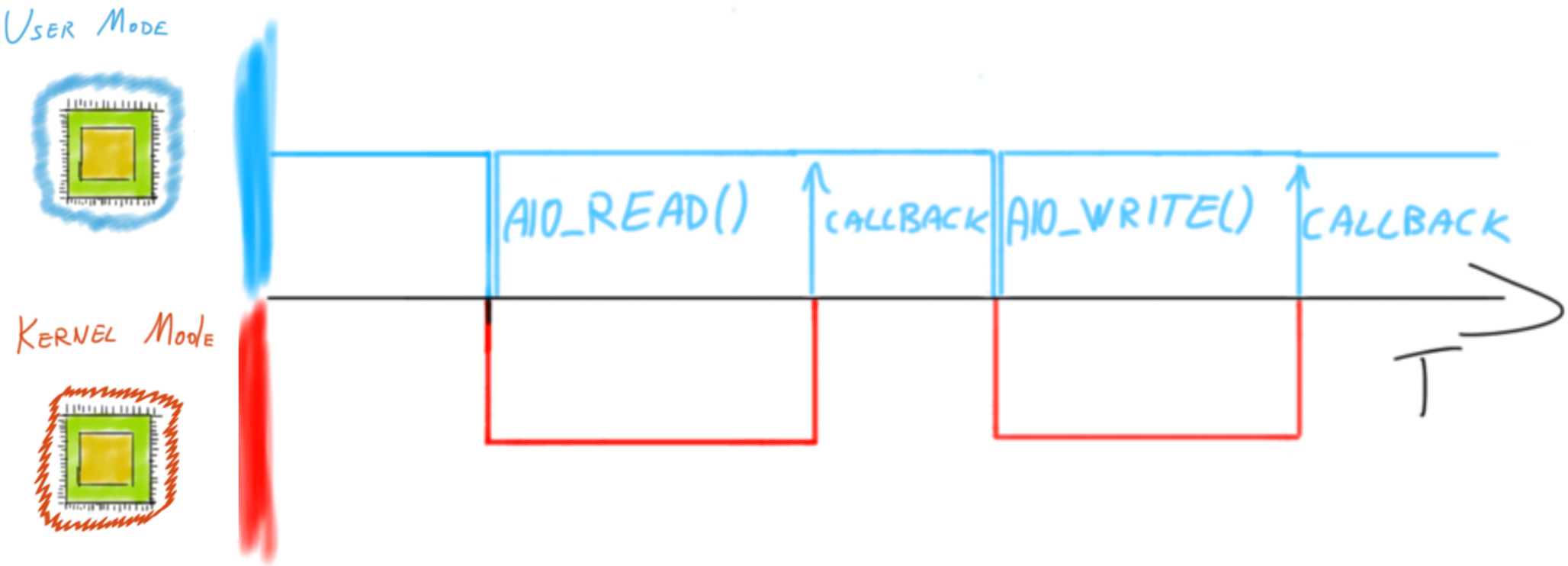
Linux AIO = JVM NIO

Asynchronous I / O [Linux]

Linux AIO = JVM NIO

NewIO… since 2004!
(No-one calls it “new” any more)

Asynchronous I / O [Linux]



Less time wasted waiting.

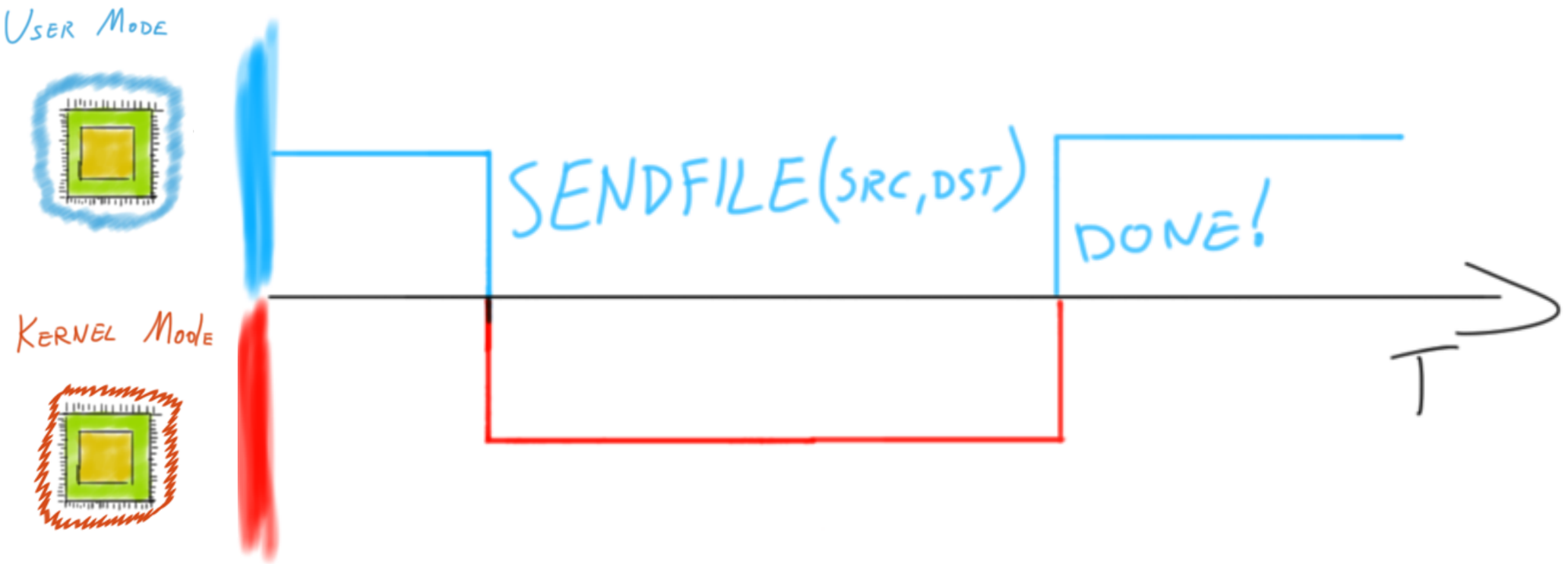
Same amount of buffer copies.

ZeroCopy = `sendfile` [Linux]

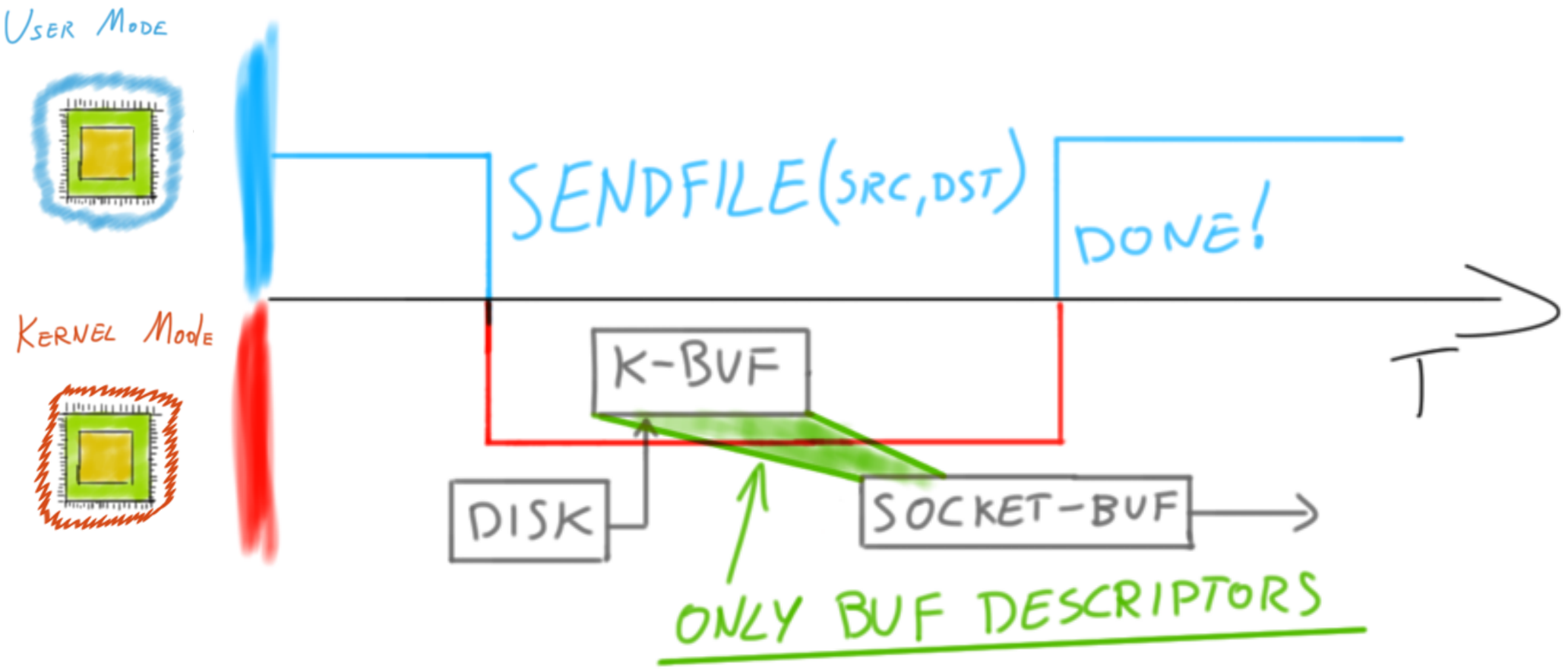
"Work smarter.
Not harder."

http://fourhourworkweek.com/

ZeroCopy = `sendfile` [Linux]

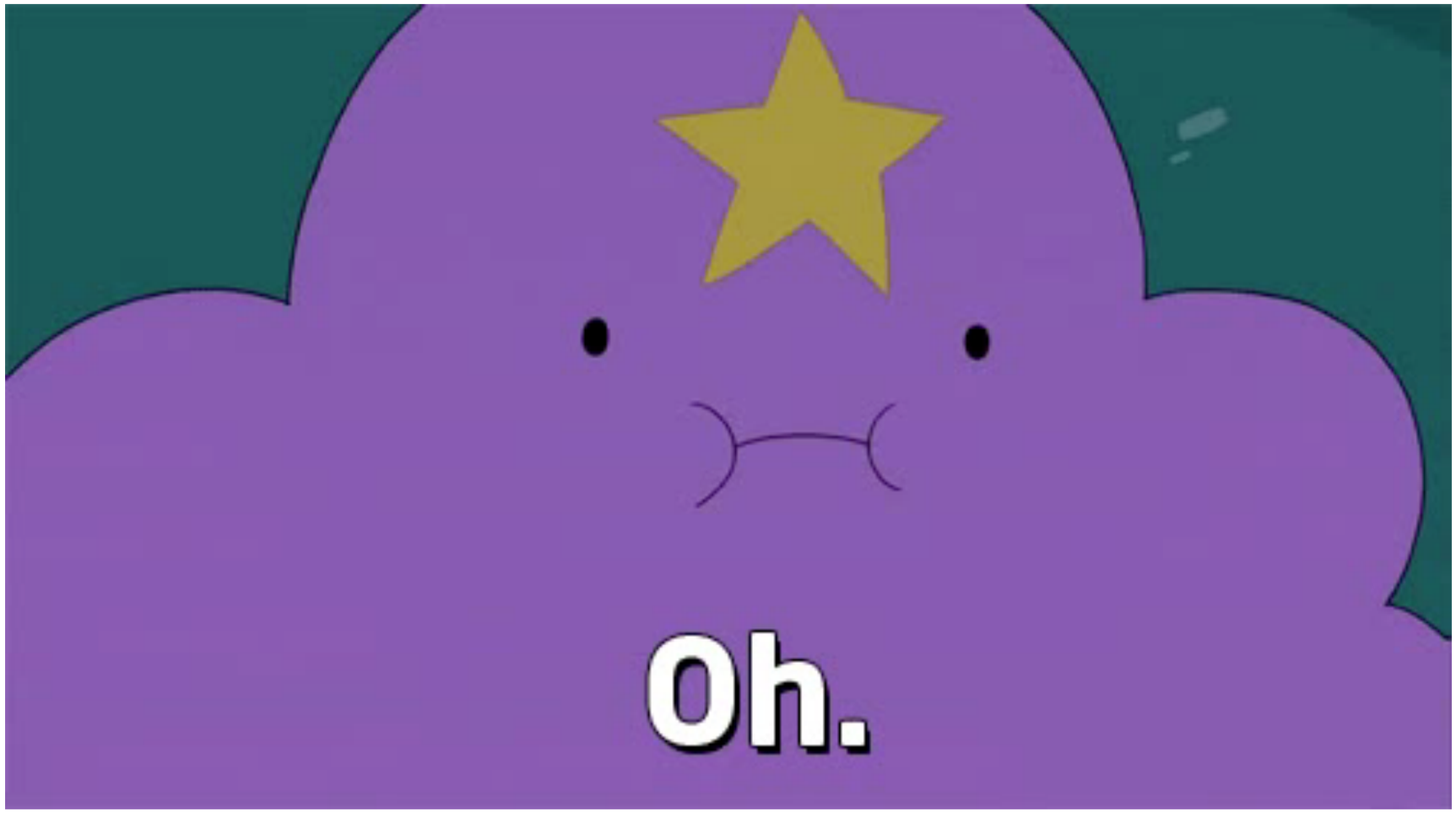


ZeroCopy = `sendfile` [Linux]



Data never leaves kernel mode!

ZeroCopy…



C10K and beyond

C10K and beyond

"10.000 concurrent connections"

Not a new problem, pretty old actually: ~12 years old.

http://www.kegel.com/c10k.html

C10K and beyond

It's **not** about **performance**.
It's about **scalability**.

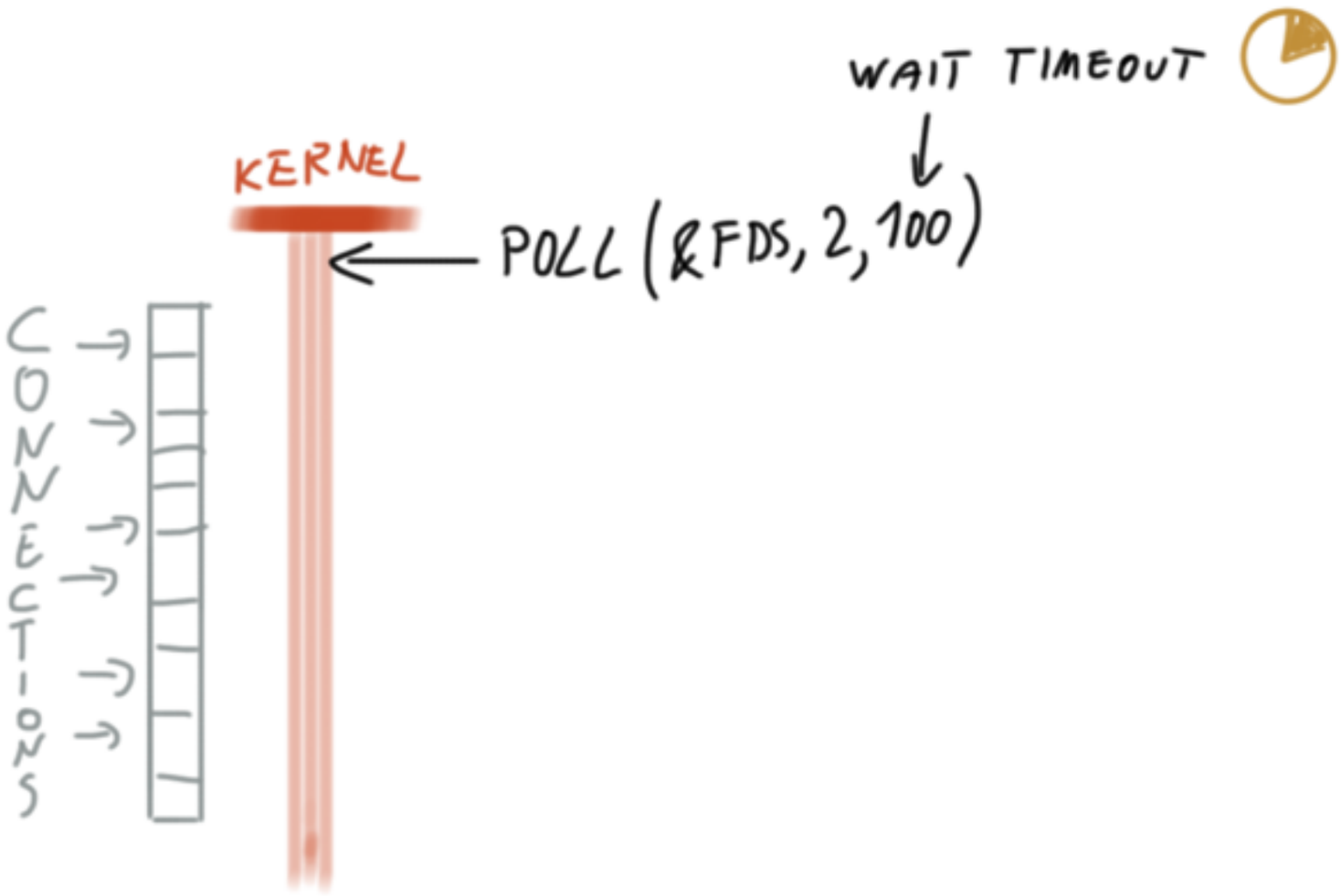
These are **orthogonal** things.

Threading differences: apache / nginx

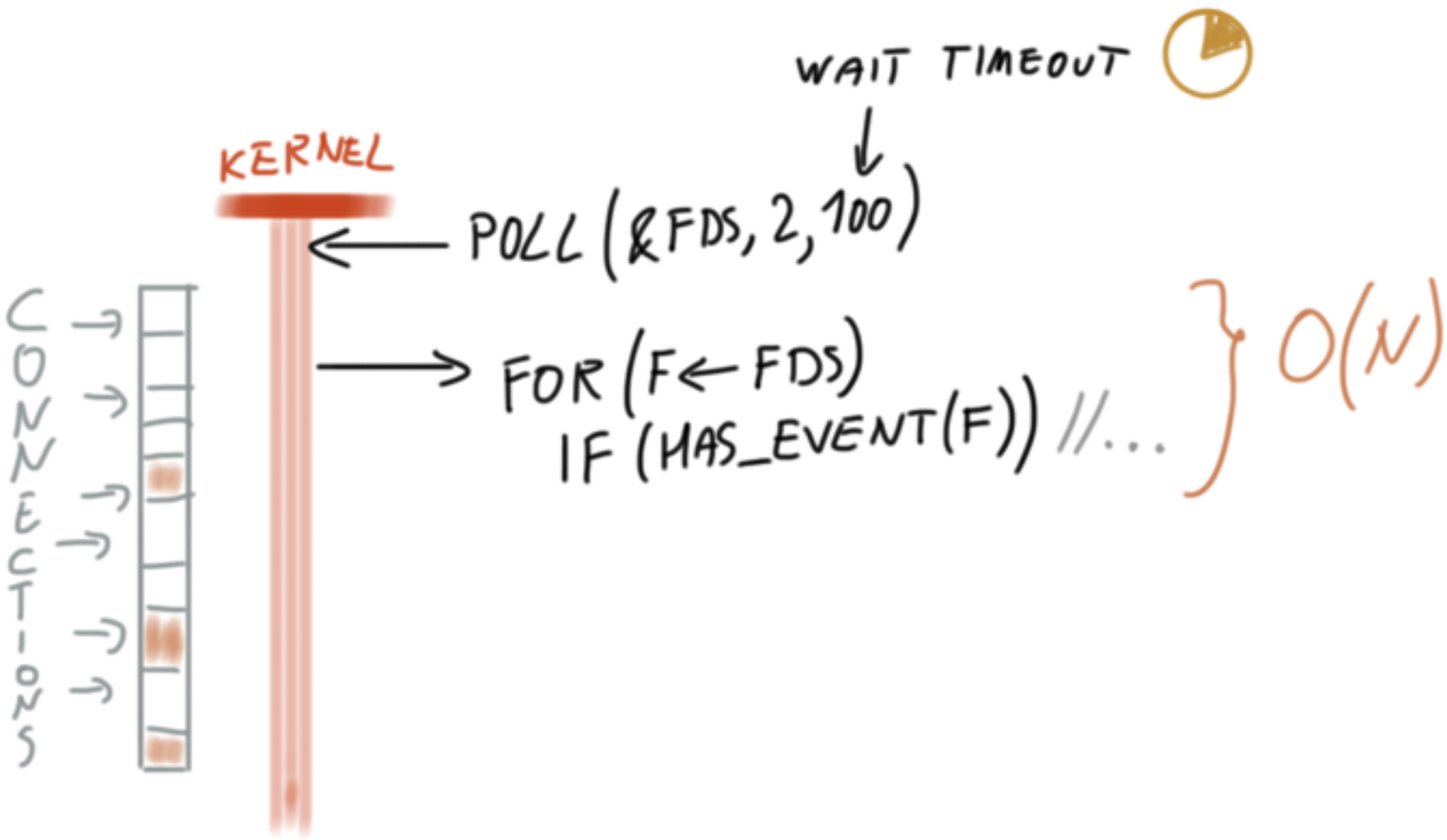
select/poll

C10K – poll

C10K – poll

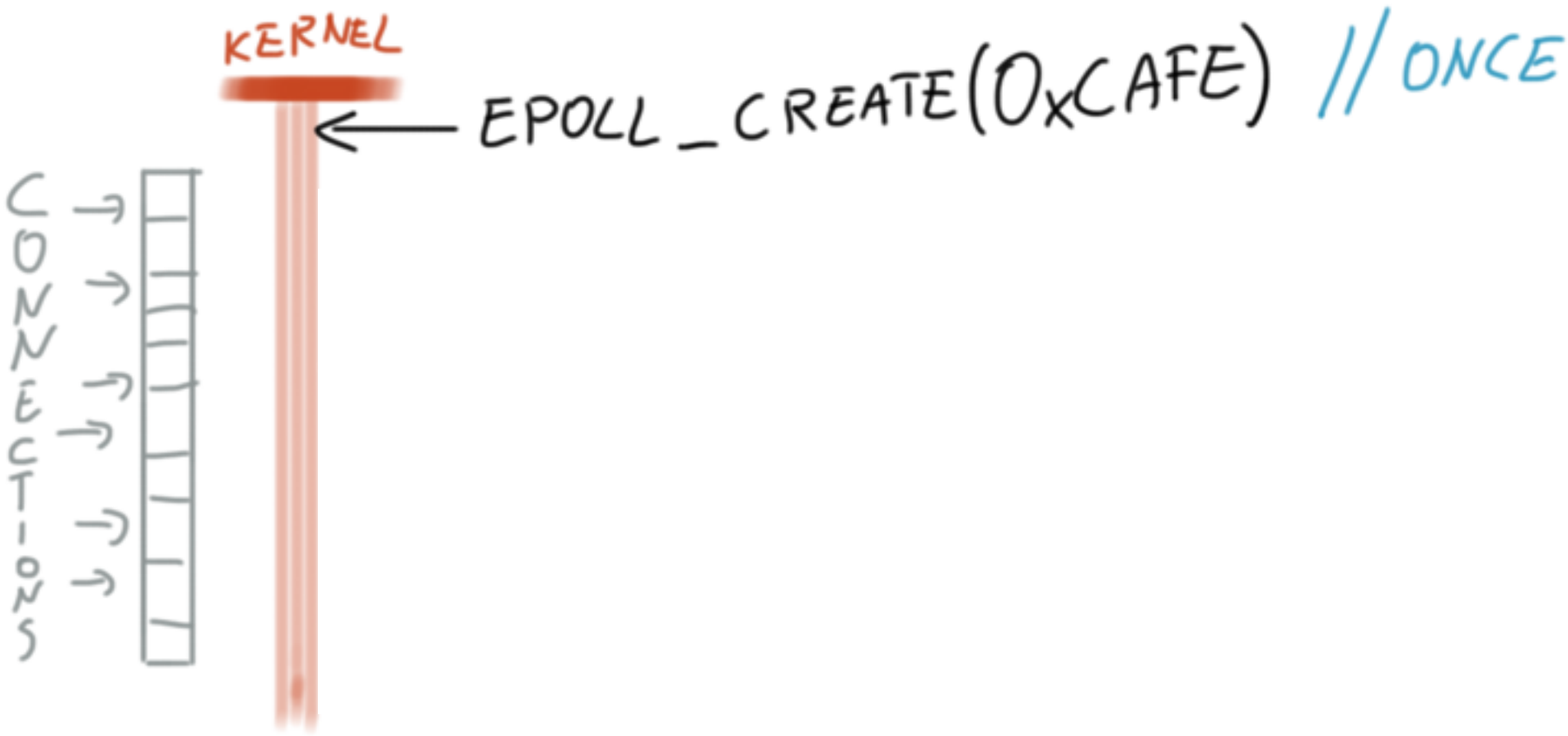


C10K – poll

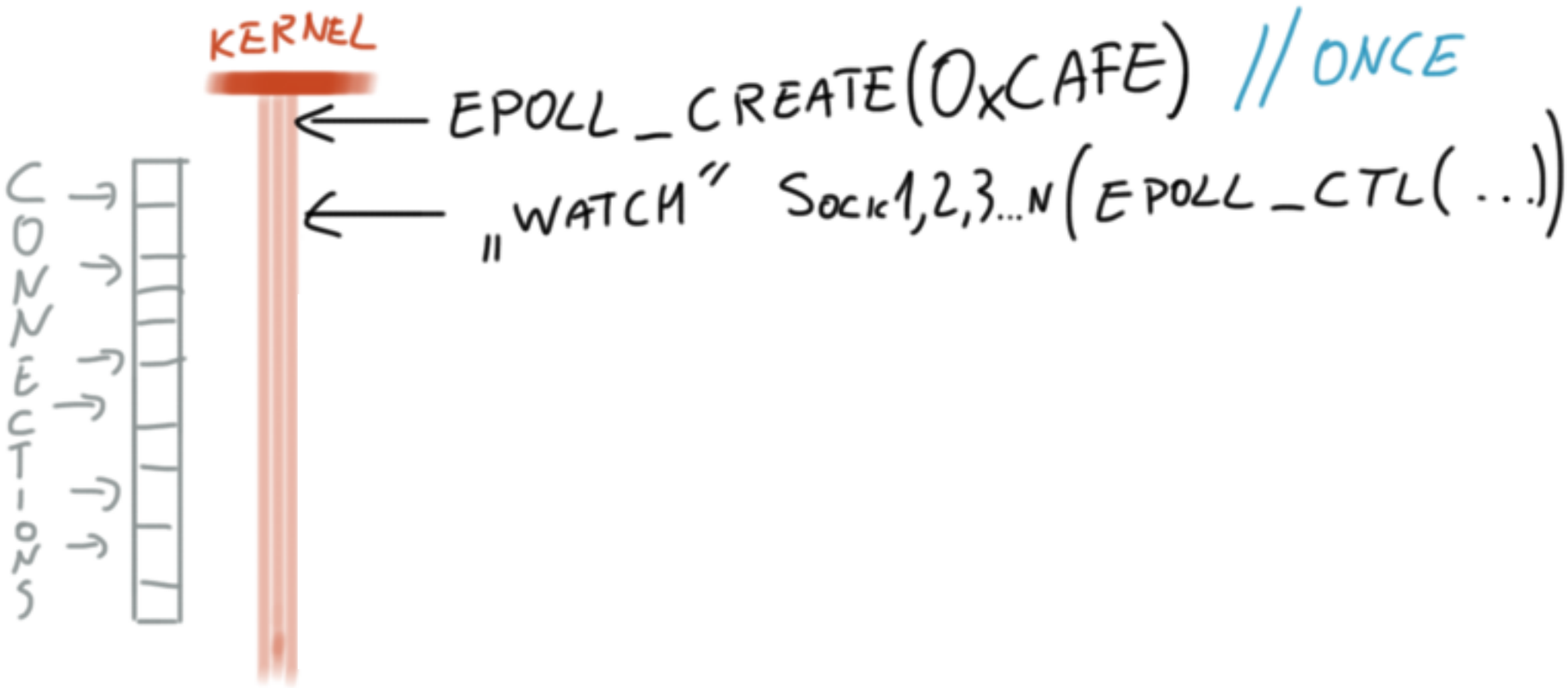

WAIT TIMEOUT
KERNEL
POLL (&FDS, 2, 100)
FOR (F ← FDS)
IF (HAS_EVENT(F)) //...
O(N)
CONNECTIONS

epoll

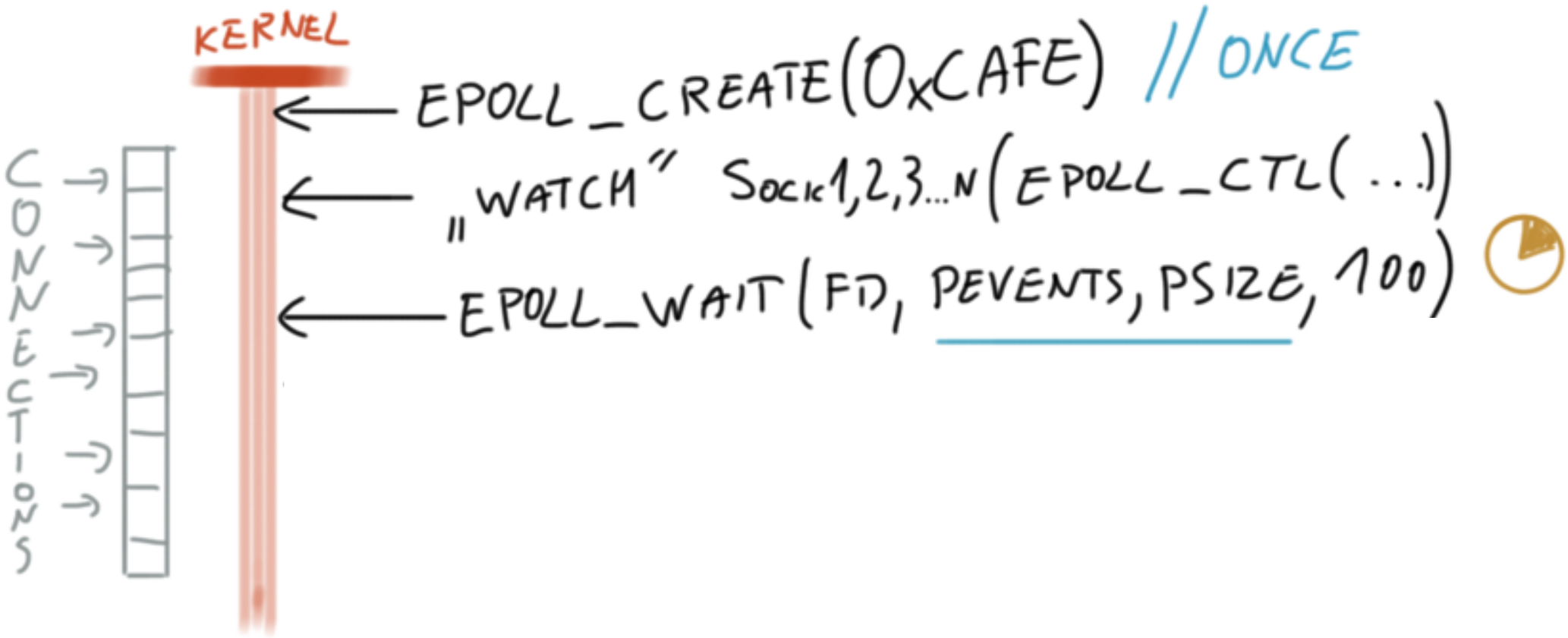
C10K – `epoll` [Linux]



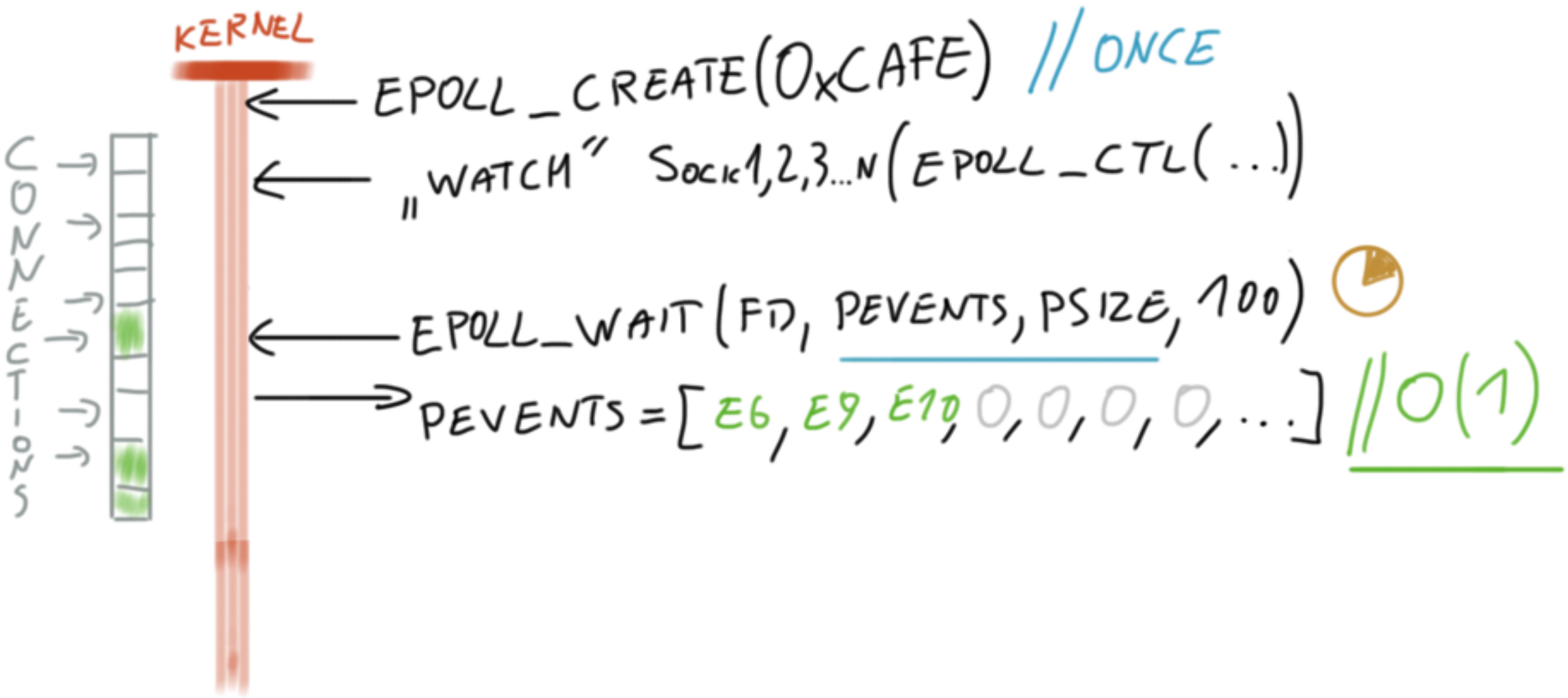
C10K – `epoll` [Linux]



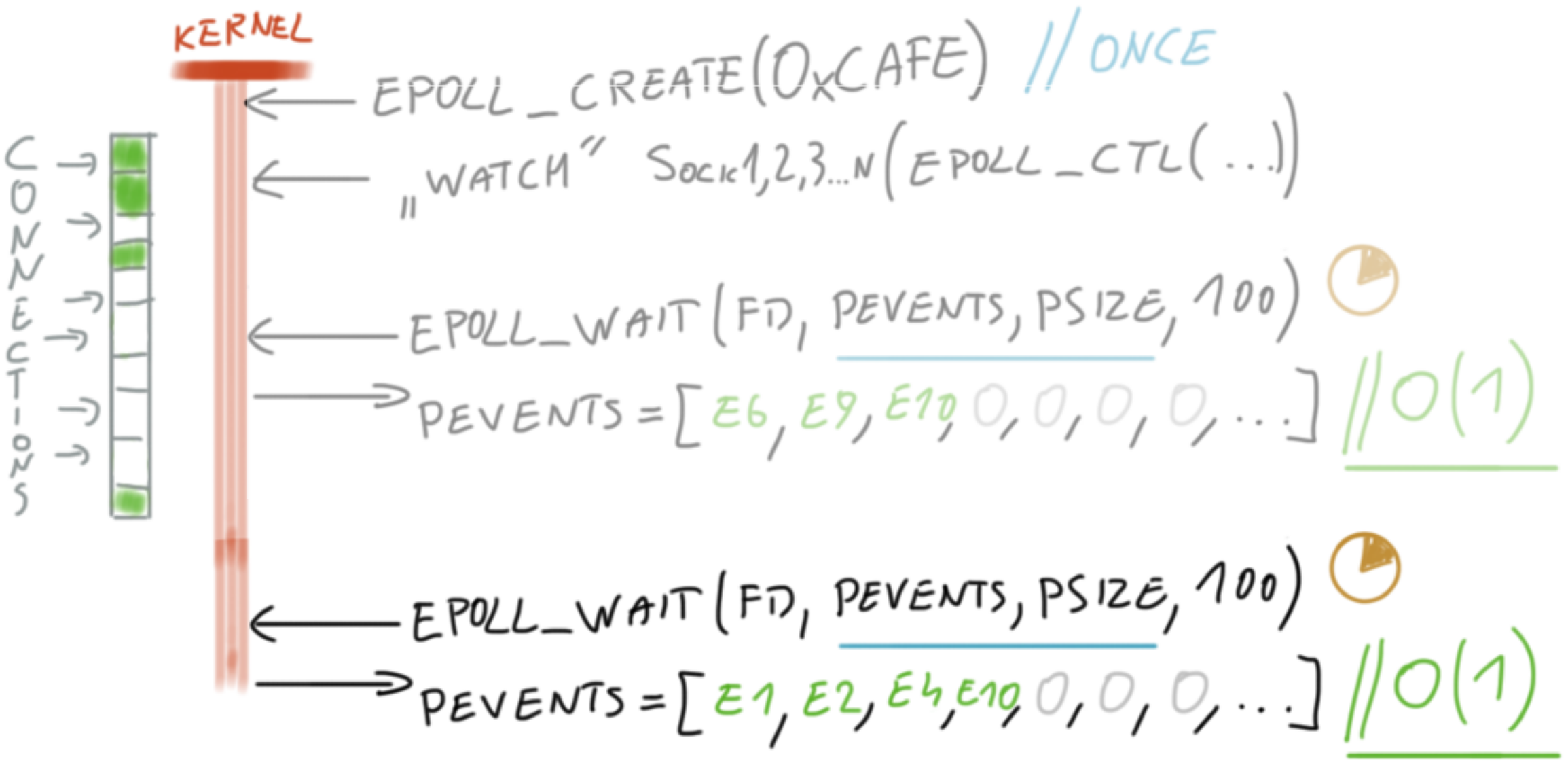
C10K – `epoll` [Linux]



C10K – `epoll` [Linux]



C10K – `epoll` [Linux]



C10K

`O(n)` is a no-go for *epic* scalability.

C10K

`O(n)` is a no-go for *epic* scalability.

State of Linux scheduling:

`O(n)` ⇒ `O(1)` ⇒ CFS (`O(1)`/ `O(log n)`)

And Socket selection:

Select/Poll `O(n)` ⇒ EPoll (`O(1)`)

C10K

O(n) is a no-go for *epic* scalability.

State of Linux scheduling:

O(n) ⇒ O(1) ⇒ CFS (O(1))

And Socket selection:

Select/Poll O(n) ⇒ EPoll (O(1))

Moral:

O(1) *IS a go for epic scalability.*

Distributed Systems

Distributed Systems

"… in which the failure of a computer you didn't even know existed can render your own computer unusable."

— Leslie Lamport

http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt

Distributed Systems

The bigger the system,
the more "random" latency / failure noise.

Embrace instead of hiding it.

Distributed Systems

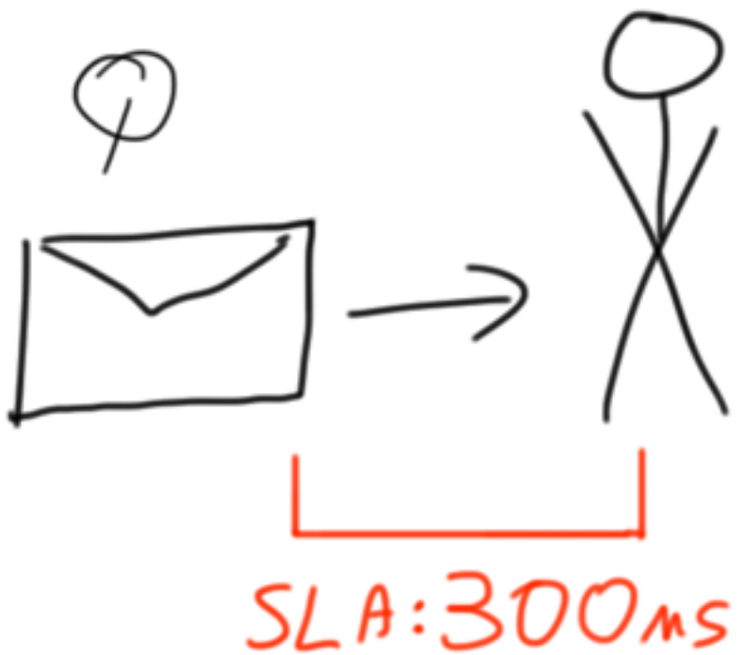
Backup Requests

Backup requests

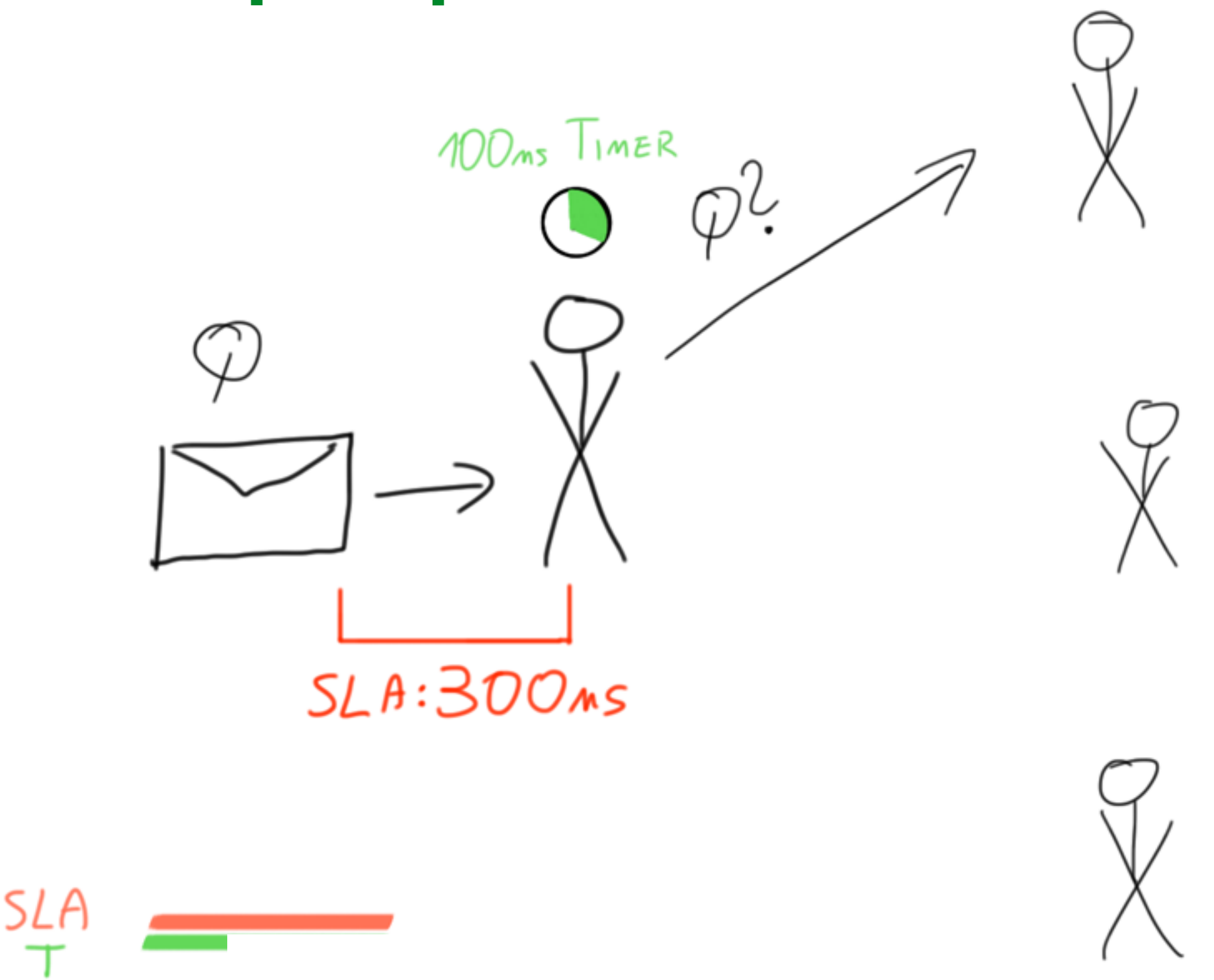
A technique for fighting "**long tail latencies**".

By **issuing duplicated work**, when SLA seems in danger.

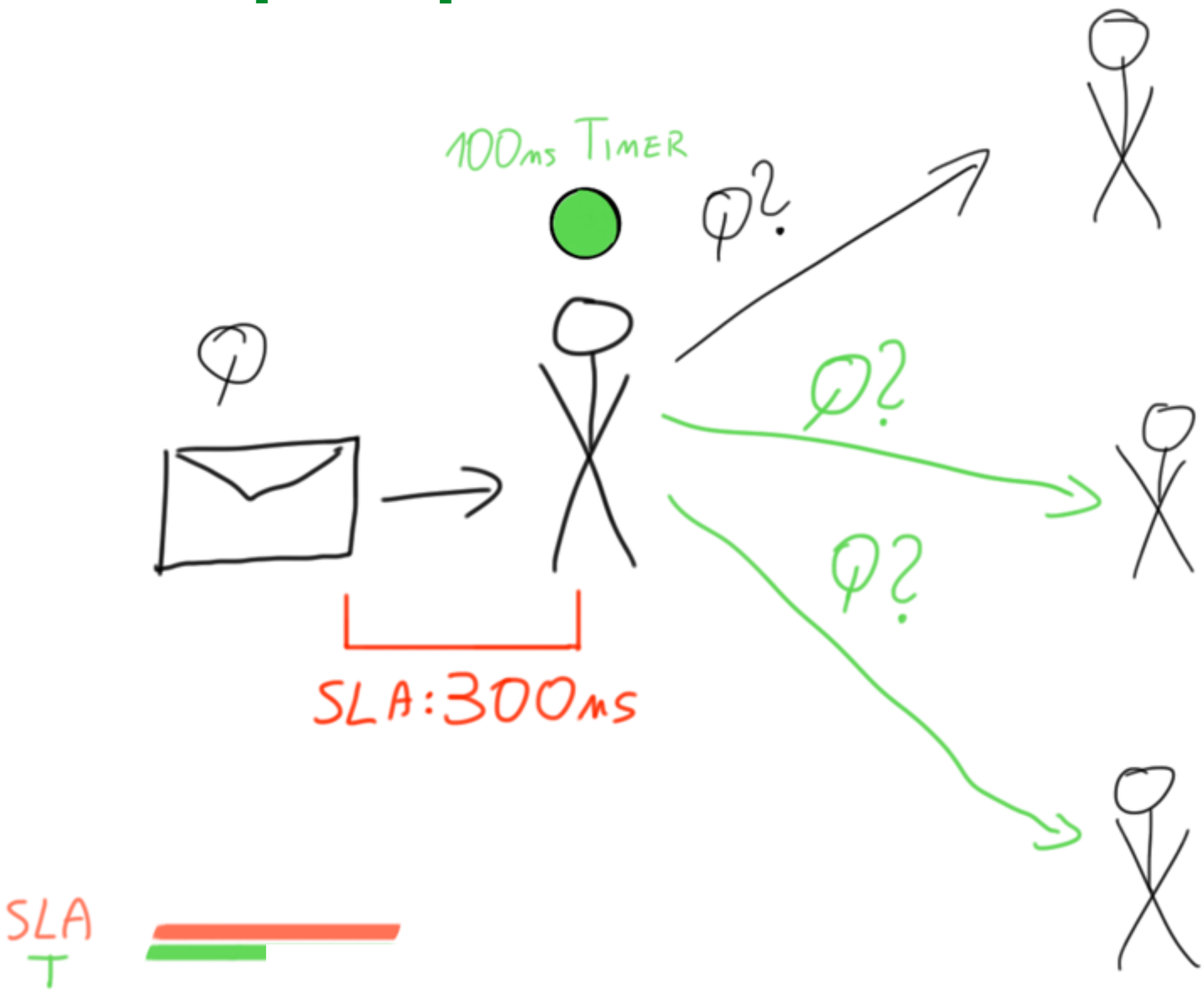
Backup requests



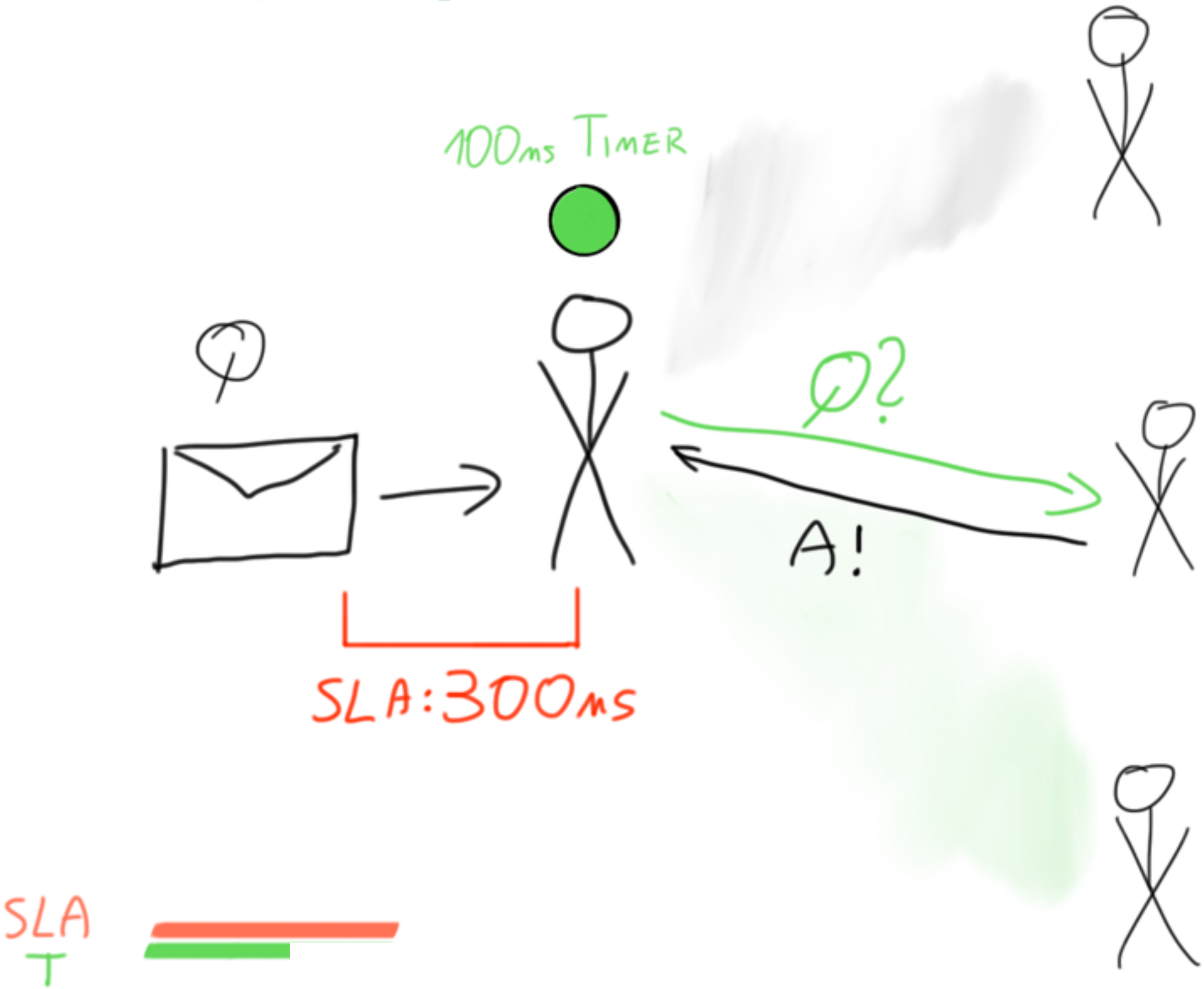
Backup requests



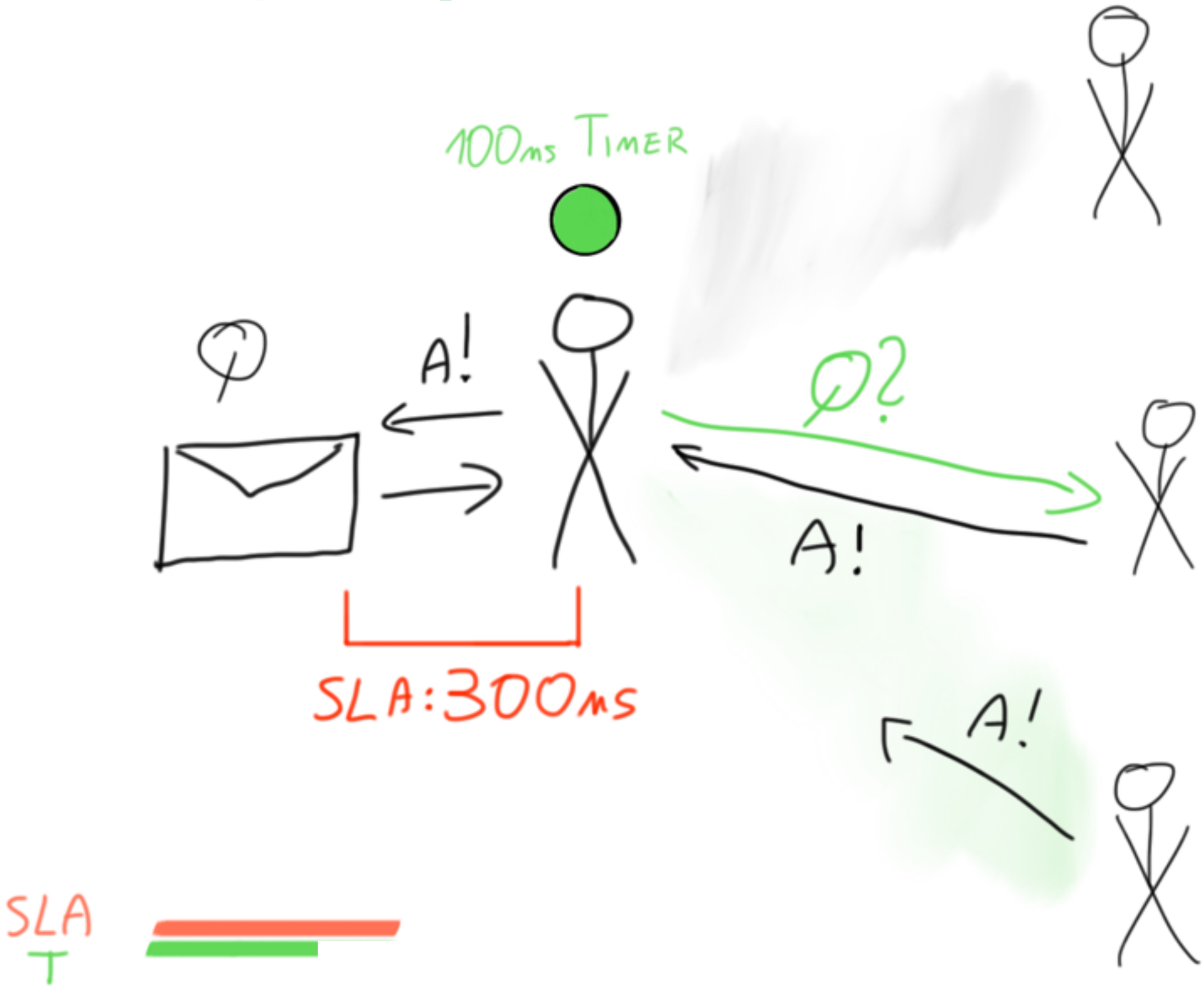
Backup requests - send



Backup requests - send



Backup requests - send



Backup requests

	Avg	Std dev	95%ile	99%ile	99.9%ile
No backups	33 ms	1524 ms	24 ms	52 ms	**994 ms**
After 10ms	14 ms	4 ms	20 ms	23 ms	50 ms
After 50ms	16 ms	12 ms	57 ms	63 ms	68 ms

Jeff Dean - Achieving Rapid Response Times in Large Online Services
Peter Bailis - Doing Redundant Work to Speed Up Distributed Queries
Akka - Krzysztof Janosz @ Akkathon, Kraków - **`TailChoppingRouter`** (docs, pr)

Distributed Systems

Combined Requests

Combined requests

A technique for **avoiding duplicated work**.

By **aggregating** requests, ***possibly increasing latency***.

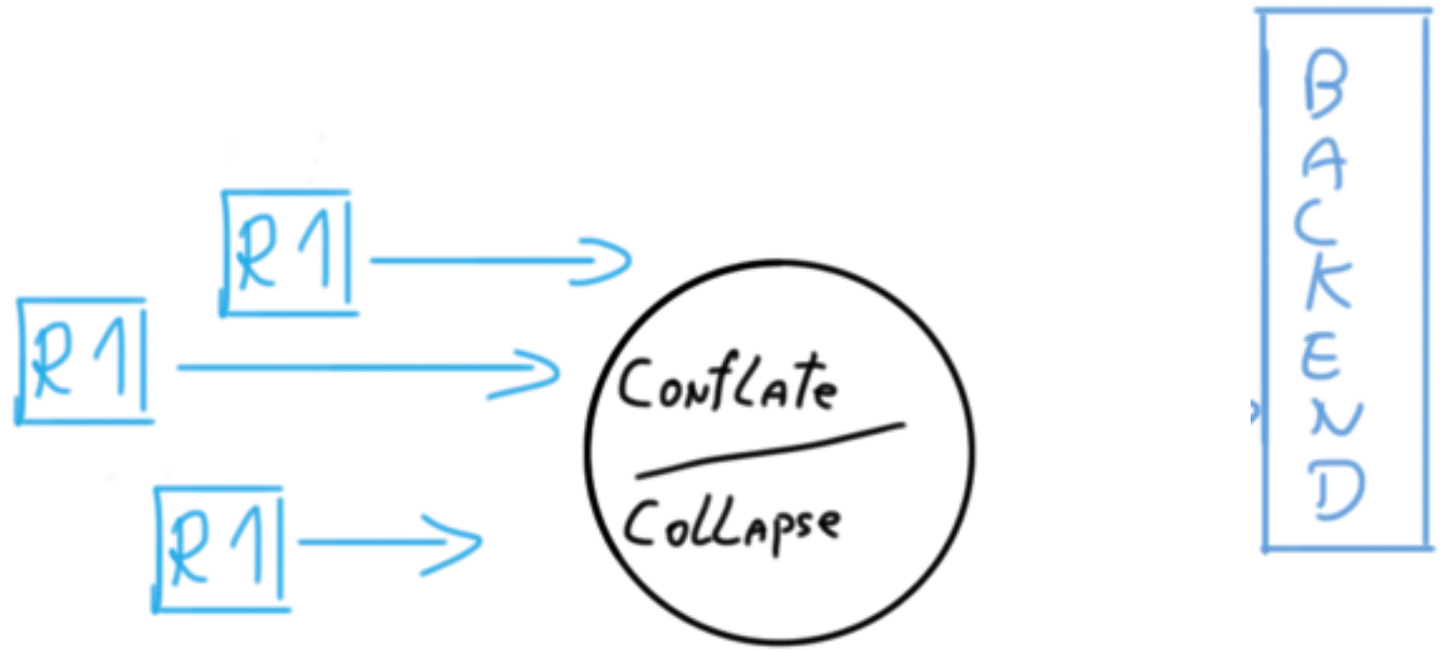
Combined requests

A technique for **avoiding duplicated work**.

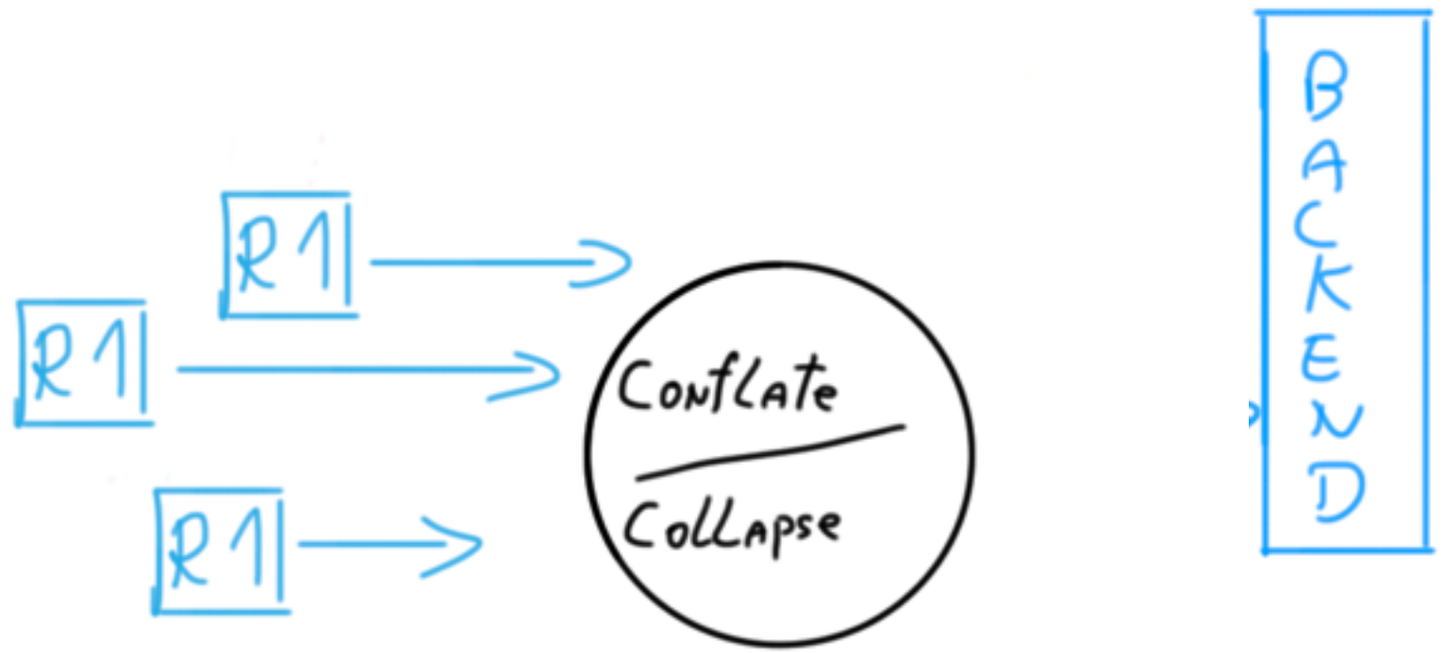
By **aggregating** requests, ***possibly increasing latency***.

"Wat? Why would I increase latency!?"

Combined requests

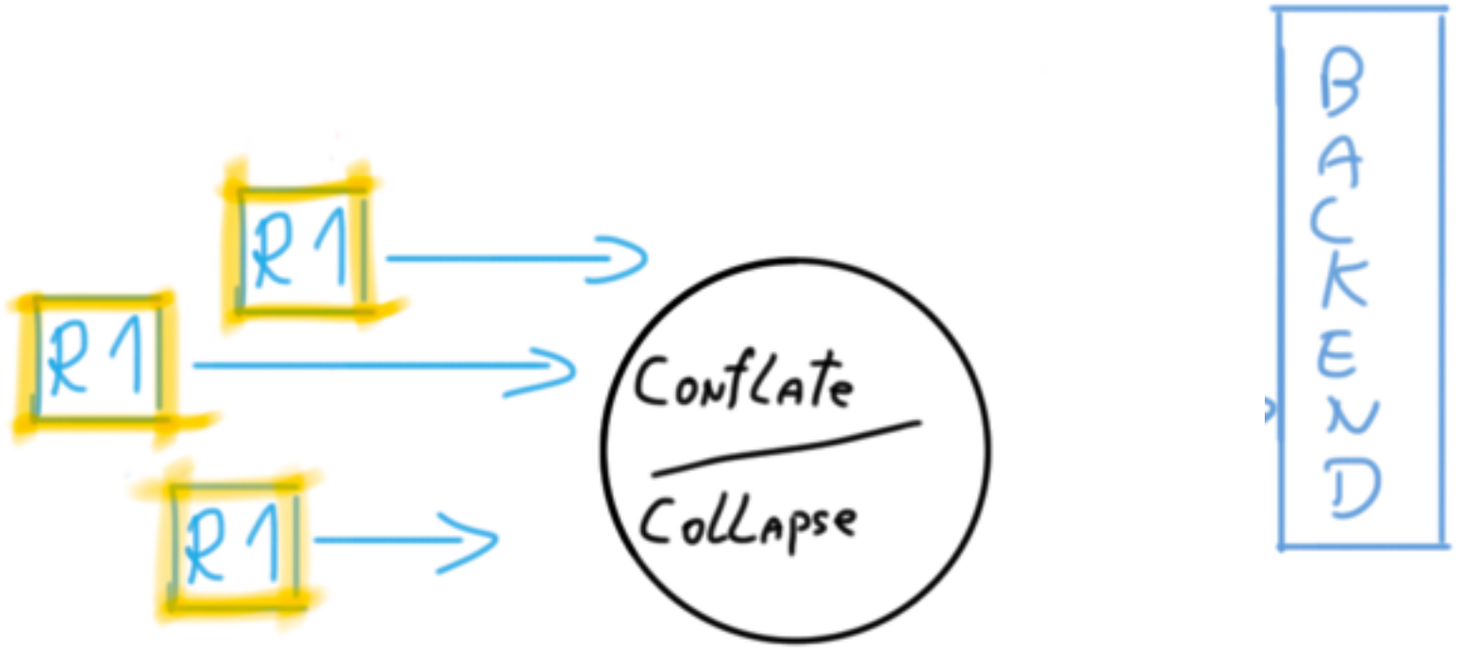


Combined requests



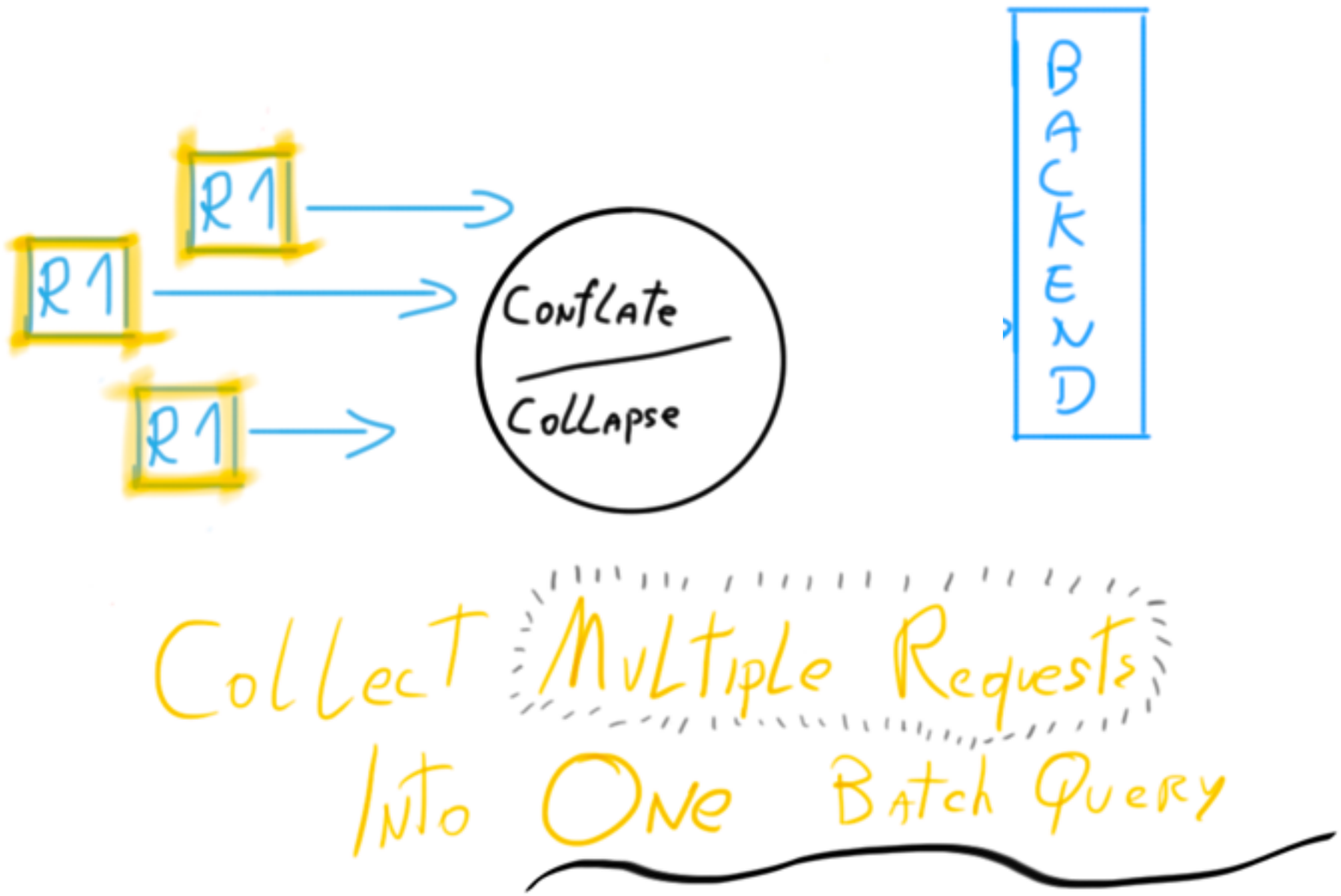
Collect Multiple Requests
Into One Batch Query

Combined requests

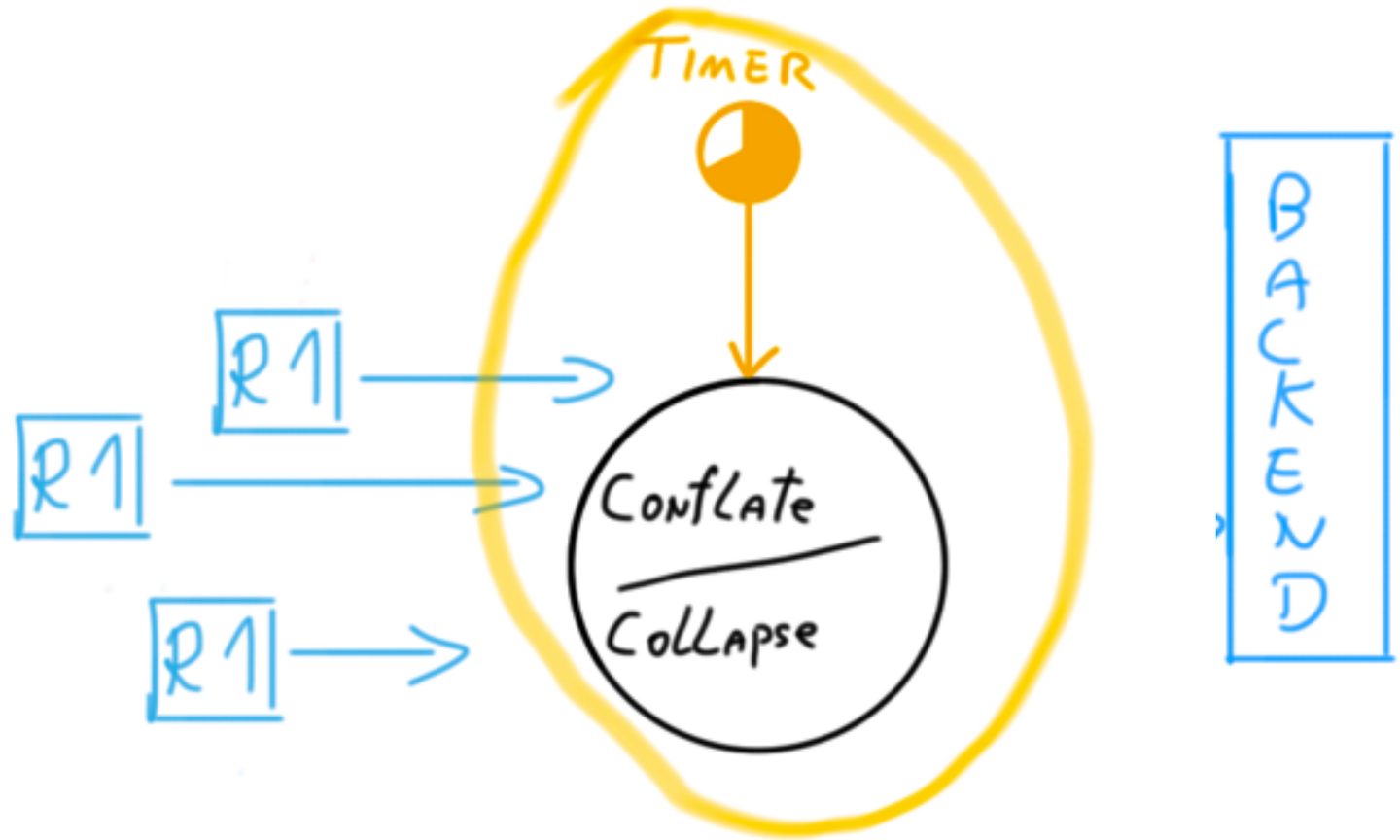


Collect Multiple Requests
Into One Batch Query

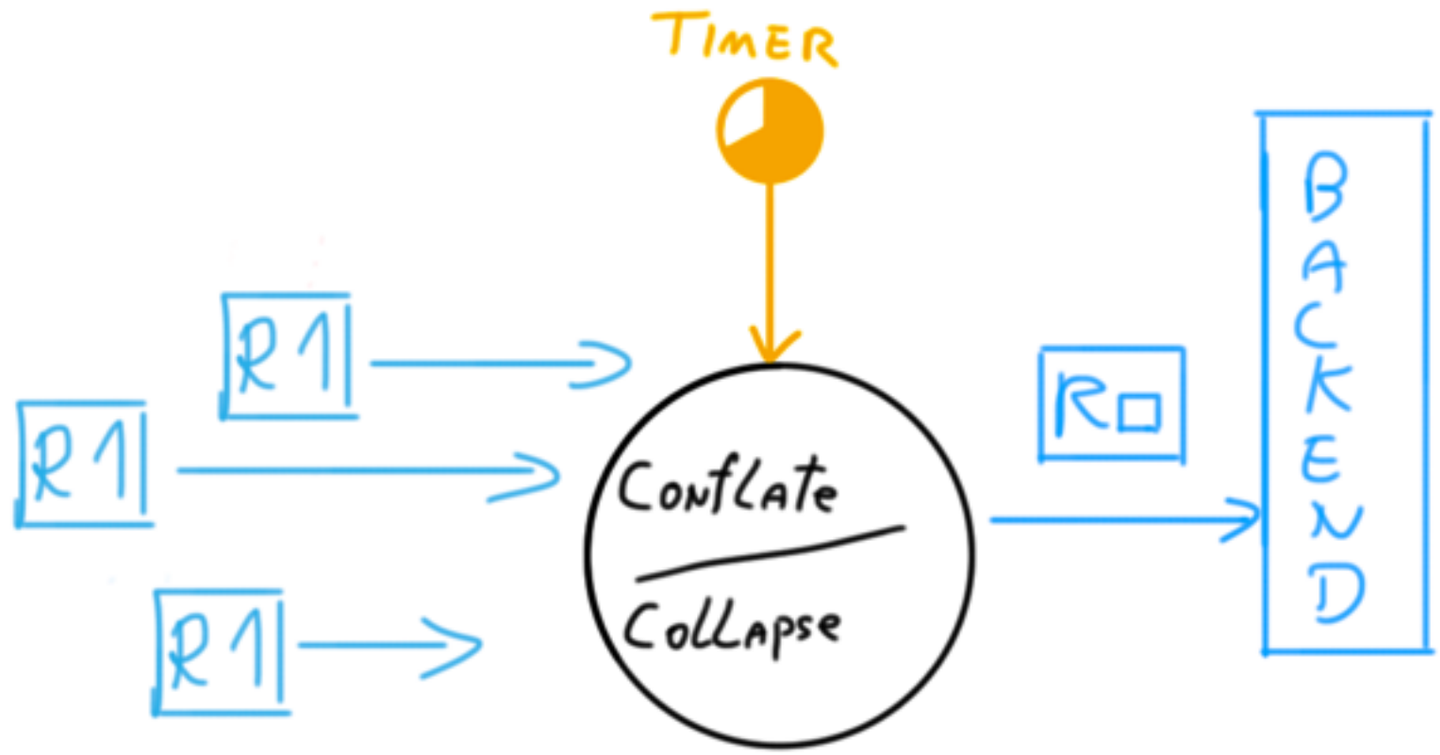
Combined requests



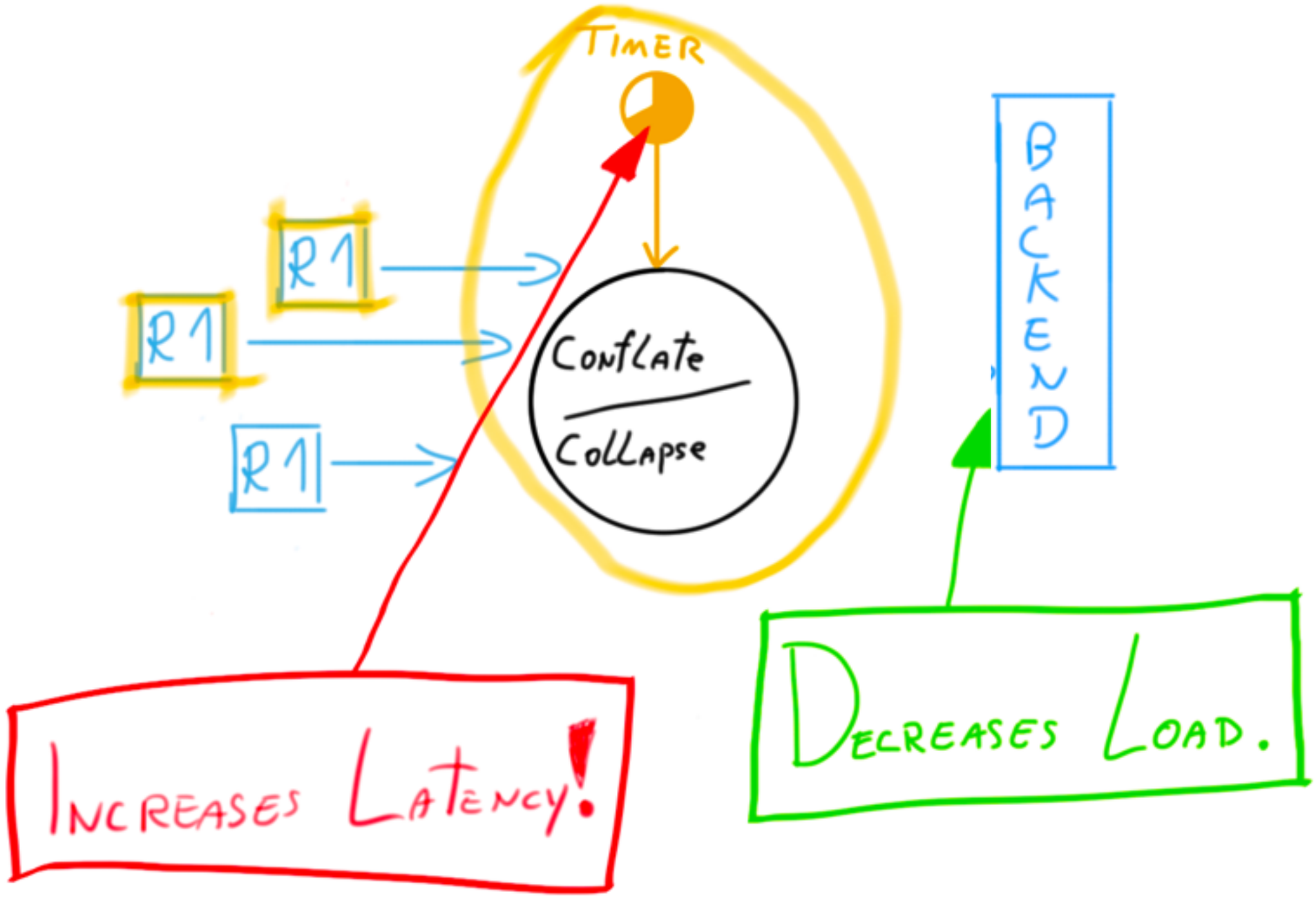
Combined requests



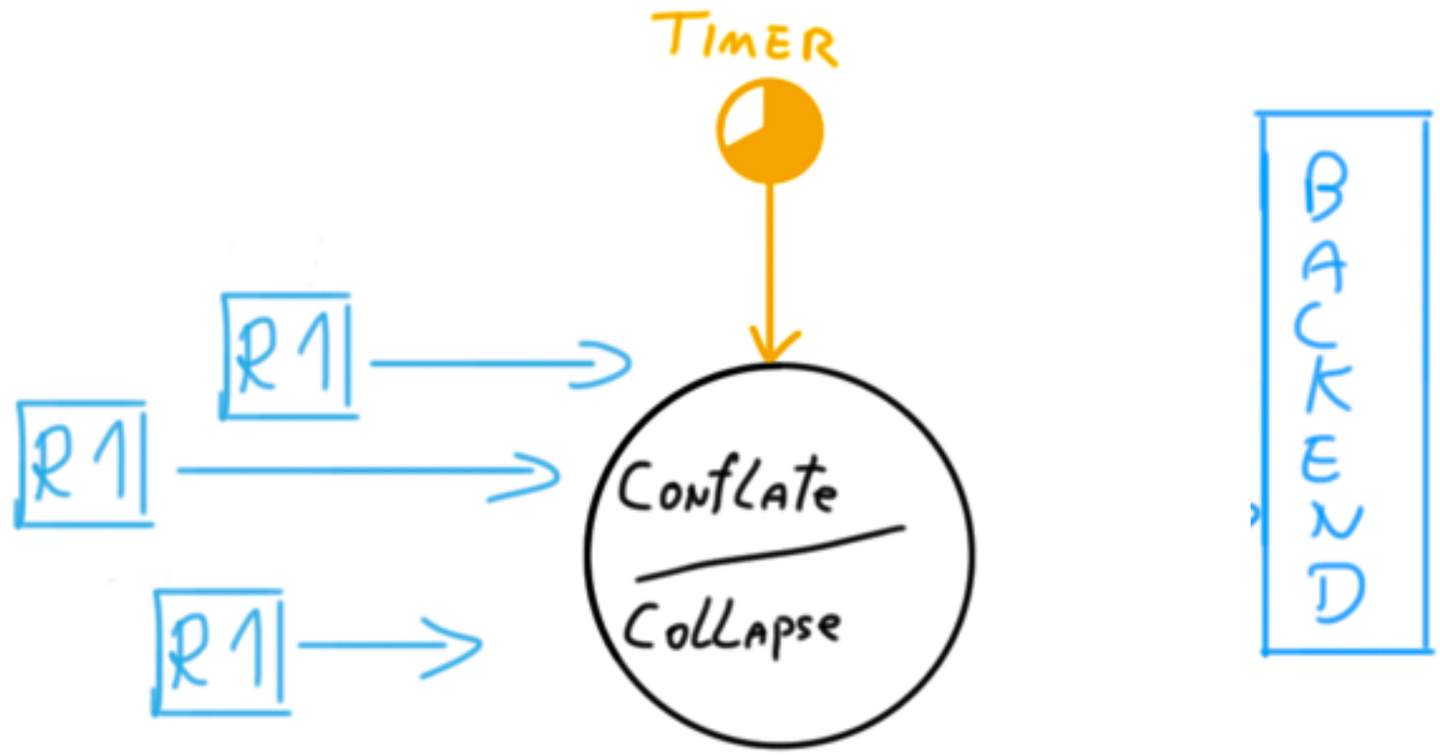
Combined requests



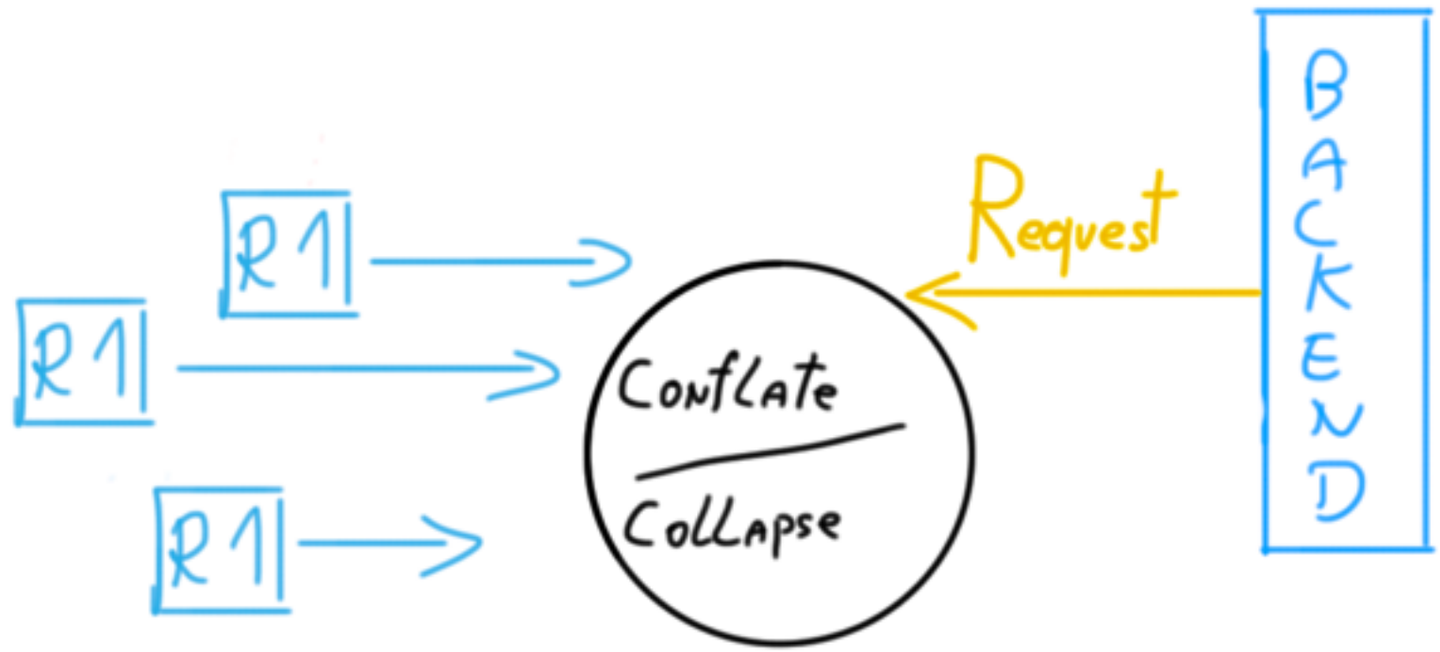
World of Tradeoffs



Combined requests: timer → backpressure

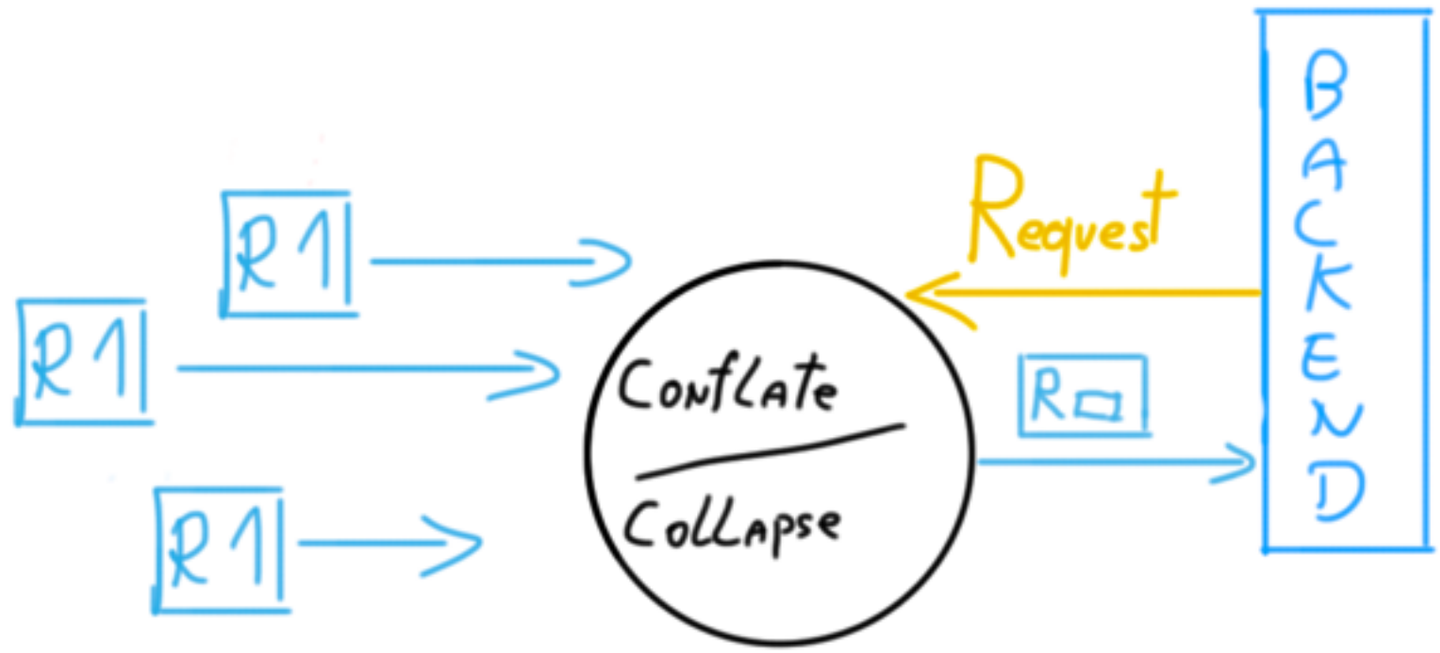


Combined requests: timer → backpressure



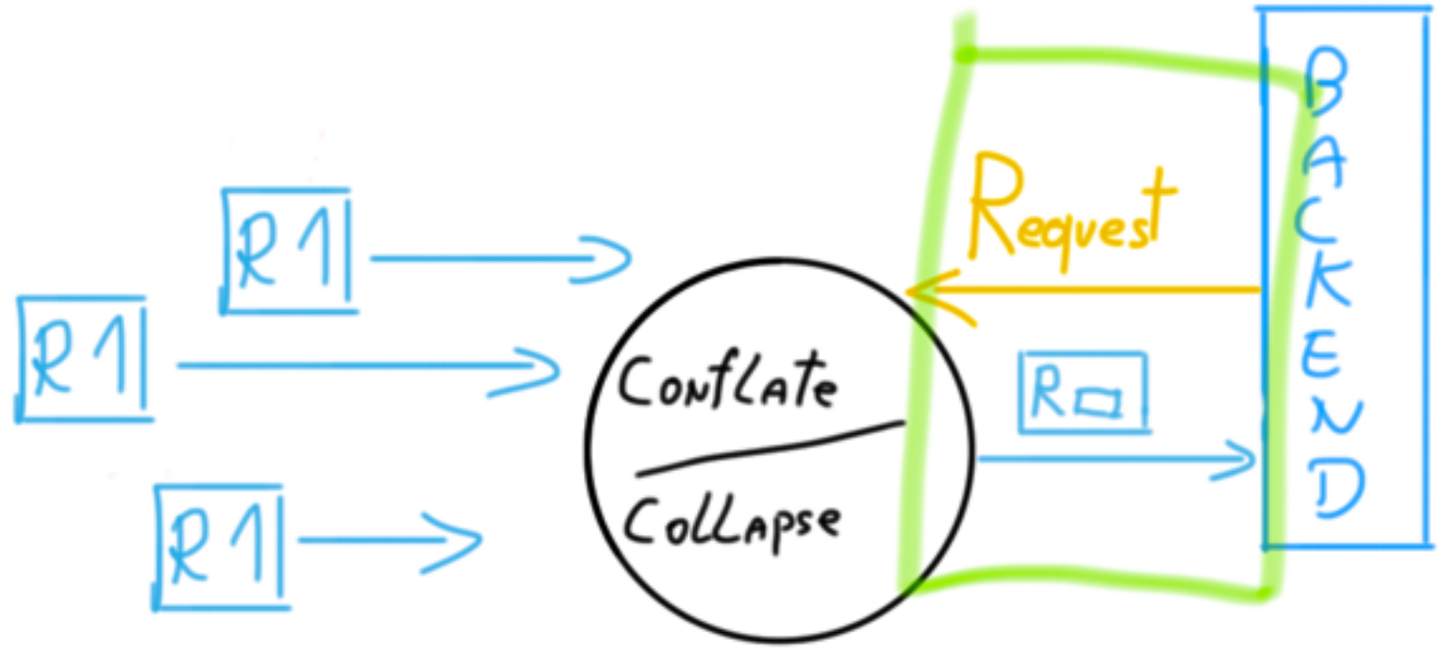
Instead of timers,
the backend can signal *"requesting work"*.

Combined requests



Instead of timers,
the backend can signal *"requesting work"*.

Combined requests



This is **pull-based back-pressure**.

Wrapping up

Wrapping up

Functional programming is awesome,

Wrapping up

Functional programming is awesome,

- *Someone* has to *bite the bullet* though*!*
- We're all running on *real* hardware.
- We do it so you don't have to.

Wrapping up

Functional programming is awesome,

- *Someone* has to *bite the bullet* though*!*
- We're all running on *real* hardware.
- We do it so you don't have to.

- Keep your apps pure
- Be aware of internals
- *Async all the things!*
- *Messaging all the way!*

Yesterday on Jan Pustelnik's talk:

Ok, so tell me about the performance . . .

All times in milliseconds

Program / Input size	20k	200k	500k	1m	2m
MergeSort Scala	9.5	–	411	1035.5	–
Haskell Imperative	4.2	61.5	161.2	347.7	–
Haskell Functional	18.9	360.4	1072	2747	–
Haskell Fnct. saving	6.9	31.8	515.4	1280	–
StdLibSort C++	1.0	–	29.1	58.4	117.1
C++ Functional	44.5	–	1626	3751	8881
Scala Imperative	1.7	–	54.3	112.7	231.8
Scala Imp. Vector	17.1	–	816.5	1790	4648
Scala Functional	33.46	–	1058	2192	4726
Scala Funct. Vector	22.9	–	865.7	1981	4375
Scala ST Monad	138.3	–	5535	13428	35290

Typesafe

Just sayin'…

https://github.com/gosubpl/sortbench

Yesterday on Jan Pustelnik's talk:

Classes: 933 **Instances:** 94,221,896 **Bytes:** 2,812,265,088

Class Name	Bytes [%]	Bytes	Instances
java.util.**TreeMap$Entry**		1,092,023,960 (38.8%)	27,300,599 (28.9%)
java.util.**TreeMap**		262,062,624 (9.3%)	5,459,638 (5.7%)
java.lang.**StackTraceElement**		187,209,920 (6.6%)	5,850,310 (6.2%)
java.lang.**Integer**		178,660,576 (6.3%)	11,166,286 (11.8%)
javax.management.openmbean.**CompositeDataSupport**		131,030,760 (4.6%)	5,459,615 (5.7%)
scala.**Tuple2**		118,659,552 (4.2%)	4,944,148 (5.2%)
int[]		107,853,032 (3.8%)	3,308 (0.0%)
scalaz.effect.**STFunctions$$anon$5**		98,696,464 (3.5%)	6,168,529 (6.5%)

MergeSort Scala	9.5	–	411	1035.5	–
Haskell Imperative	4.2	61.5	161.2	347.7	–
Haskell Functional	18.9	360.4	1072	2747	–
Haskell Fnct. saving	6.9	31.8	515.4	1280	–
StdLibSort C++	1.0	–	29.1	58.4	117.1
C++ Functional	44.5	–	1626	3751	8881
Scala Imperative	1.7	–	54.3	112.7	231.8
Scala Imp. Vector	17.1	–	816.5	1790	4648
Scala Functional	33.46	–	1058	2192	4726
Scala Funct. Vector	22.9	–	865.7	1981	4375
Scala ST Monad	138.3	–	5535	13428	35290

Typesafe

Just sayin'…

https://github.com/gosubpl/sortbench

Links

- **akka.io**
- reactive-streams.org
- **akka-user**

- Gil Tene - How NOT to measure latency, 2013
- Jeff Dean @ Velocity 2014
- Alan Bateman, Jeanfrancois Arcand (Sun) Async IO Tips @ JavaOne
- http://linux.die.net/man/2/select
- http://linux.die.net/man/2/poll
- http://linux.die.net/man/4/epoll
- giltene/jHiccup
- Linux Journal: ZeroCopy I, Dragan Stancevis 2013

- Last slide car picture: http://actu-moteurs.com/sprint/gt-tour/jean-philippe-belloc-un-beau-challenge-avec-le-akka-asp-team/2000

Links

- http://wiki.osdev.org/Context_Switching
- **CppCon: Herb Sutter "Lock-Free Programming (or, Juggling Razor Blades)"**
- http://www.infoq.com/presentations/reactive-services-scale
- **Gil Tene's HdrHistogram.org**
  - http://hdrhistogram.github.io/HdrHistogram/plotFiles.html
- Rob Pike - Concurrency is NOT Parallelism (video)
- Brendan Gregg - Systems Performance: Enterprise and the Cloud (book)
- http://psy-lob-saw.blogspot.com/2015/02/hdrhistogram-better-latency-capture.html
- Jeff Dean, Luiz Andre Barroso - The Tail at Scale (whitepaper, ACM)
- http://highscalability.com/blog/2012/3/12/google-taming-the-long-latency-tail-when-more-machines-equal.html
- http://www.ulduzsoft.com/2014/01/select-poll-epoll-practical-difference-for-system-architects/
- Marcus Lagergren - Oracle JRockit: The Definitive Guide (book)
- http://mechanical-sympathy.blogspot.com/2013/08/lock-based-vs-lock-free-concurrent.html
- Handling of Asynchronous Events - http://www.win.tue.nl/~aeb/linux/lk/lk-12.html
- http://www.kegel.com/c10k.html

Links

- www.reactivemanifesto.org/
- Seriously the only right way to micro benchmark on the JVM:
  - **JMH** openjdk.java.net/projects/code-tools/jmh/
  - **JMH** for Scala: https://github.com/ktoso/sbt-jmh
- http://www.ibm.com/developerworks/library/l-async/
- http://lse.sourceforge.net/io/aio.html
- https://code.google.com/p/kernel/wiki/AIOUserGuide
- **ShmooCon: C10M - Defending the Internet At Scale (Robert Graham)**
  - **http://blog.erratasec.com/2013/02/scalability-its-question-that-drives-us.html#.VO6E11PF8SM**
- User-level threads..... with threads. - Paul Turner @ Linux Plumbers Conf 2013
- Jan Pustelnik's talk yesterday: https://github.com/gosubpl/sortbench

Special thanks to:

- Aleksey Shipilëv
- Andrzej Grzesik
- Gil Tene
- Kirk Pepperdine
- Łukasz Dubiel
- Marcus Lagergren
- Martin Thompson
- Mateusz Dymczyk
- Nitsan Wakart
- Peter Lawrey
- Richard Warburton
- Roland Kuhn
- Sergey Kuksenko
- Steve Poole
- Viktor Klang a.k.a. √
- Antoine de Saint Exupéry :-)
- and the entire *Akka Team*
- the *Mechanical Sympathy* Mailing List

Thanks guys, you're awesome.

Learn more at:

- SCKRK.com – Software Craftsmanship Kraków
  Computer Science Whitepaper Reading Club Kraków
- KrakowScala.pl – Kraków Scala User Group
- LambdaKRK.pl – Lambda Lounge Kraków
- GeeCON.org – awesome "all around the JVM" conference, May 2015

Questions?

Typesafe

ktoso @ typesafe.com
twitter: ktosopl
github: ktoso
team blog: letitcrash.com
home: akka.io