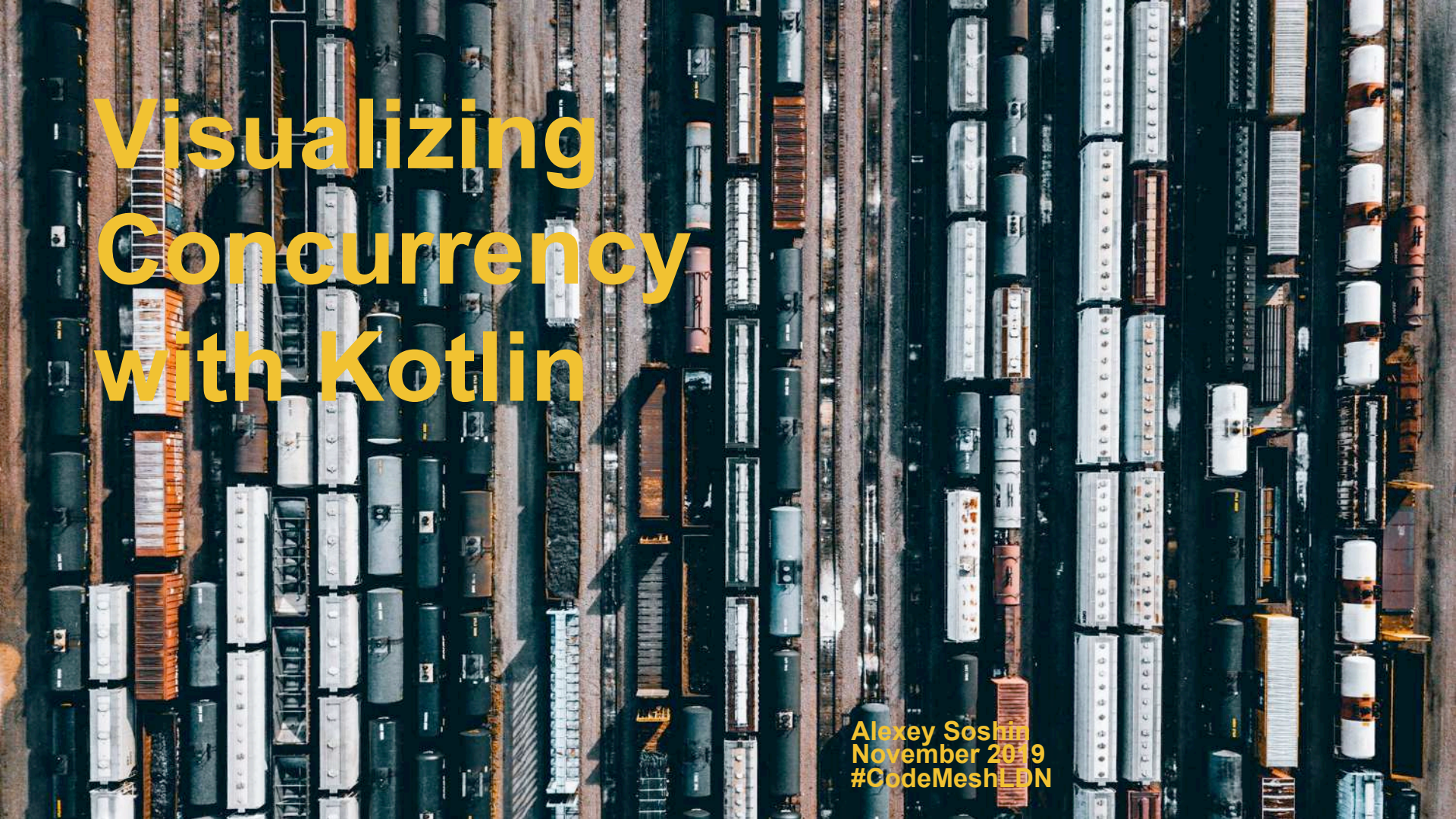


Visualizing Concurrency with Kotlin

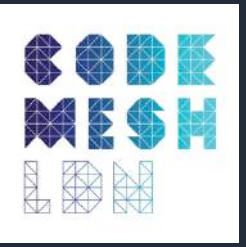
Alexey Soshin
November 2019
#CodeMeshLDN

#CodeMeshLDN

whoami

- Staff Software Engineer @Deliveroo
- Former Software Architect @Gett
- Author of “Hands-on Design Patterns with Kotlin” book

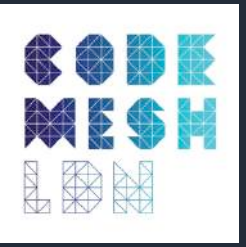
alexey_soshi
n



#CodeMeshLDN

Intro

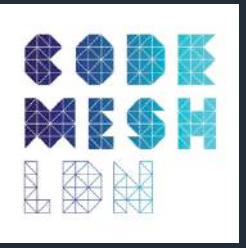
- Concurrency is a hot topic in the last 10 years
- The main reason for that is that we’re hitting the limit of how fast our CPUs can get
- We add more cores, meaning more parallelism. But there’s also limit for that
- Concurrency is then a way to optimise those resources
- Unlike some other problems, concurrency is hard to visualise, though



#CodeMeshLDN

Concurrency vs Parallelism

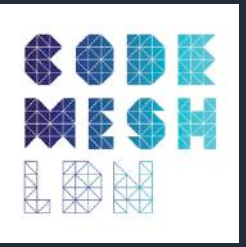
- Parallelism is “doing many things at the same time”
- Concurrency is “doing something, while you wait for something else”
- Our brains are not parallel, but they are concurrent
- We’re switching between different tasks, creating an illusion that we doing more than one thing at the same time
- Concurrency is an illusion



#CodeMeshLDN

Thread per request model

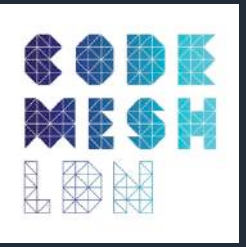
- If we go back 10 years, most webservers used “thread per request” model
- That’s how web containers such as Tomcat or web servers such as Apache operated
- That’s how many ecosystems still operate - take Ruby on Rails, for example
- This model has the benefit of simplicity, but it also has a major flaw



#CodeMeshLDN

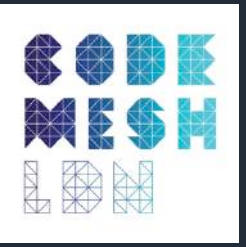
C10K problem

- Maybe you've heard about C10K problem
- How can we handle 10,000 concurrent connections on a single machine?
- Creating an OS thread has an overhead of about 1MB RAM
- While being able to create 10K threads on a modern machine, 100K are still a problem
- And there's still thread context switching done by the OS, which is very expensive



Let’s see

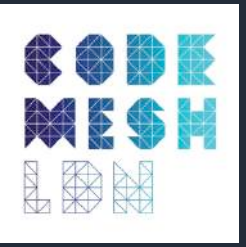
#CodeMeshLDN



#CodeMeshLDN

C10K solution

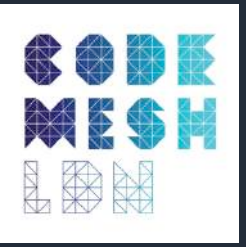
- Different languages, runtimes and frameworks solve C10K problem differently
- NodeJS is known to run an event loop on a single thread
- Erlang uses actors to pass messages
- Kotlin and Go have coroutines/goroutines



#CodeMeshLDN

Coroutines

- Coroutines are known as “lightweight threads”
- Every computation is put within “continuation”
- You can think of continuation as a simple state machine, or a function with a callback
- Coroutine can be suspended
- While coroutine is suspended, other coroutines can run
- Meaning - more concurrency

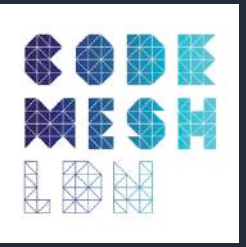


#CodeMeshLDN

Coroutines under the hood

```
suspend fun downloadImage(imageName: String) {
    val url = createUrl(imageName)
    println("URL is $url")
    val bits = fetchImage(url)
    println("Size: ${bits.size}")
    saveImage(bits)
    println("Done")
}
```

```
fun downloadImage(imageName: String) {
    val url = createUrl(imageName)
    a(url)
}

fun a(url: String) {
    println("URL is $url")
    val bits = fetchImage(url)
    b(bits)
}
fun b(bits: ByteArray) {
    println("Size: ${bits.size}")
    saveImage(bits)
    c()
}

fun c() {
    println("Done")
}
```

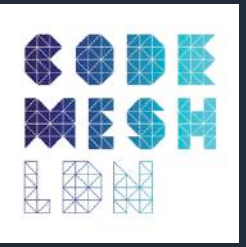


#CodeMeshLDN

Coroutines under the hood

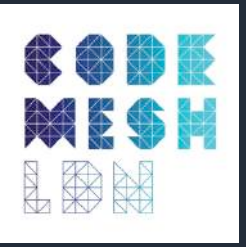
```
downloadImage("cat")

fun downloadImage(imageName: String) {
    val url = createUrl(imageName) //🐱
    a(url)
}

fun a(url: String) {
    println("URL is $url")
    val bits = fetchImage(url) //🐱
    b(bits)
}
fun b(bits: ByteArray) {
    println("Size: ${bits.size}") //🐱
    saveImage(bits)
    c()
}

fun c() {
    println("Done") //🐱
}
```

```
downloadImage("dog")

fun downloadImage(imageName: String) {
    val url = createUrl(imageName) //🐶
    a(url)
}

fun a(url: String) {
    println("URL is $url")
    val bits = fetchImage(url) //🐶
    b(bits)
}
fun b(bits: ByteArray) {
    println("Size: ${bits.size}") //🐶
    saveImage(bits)
    c()
}

fun c() {
    println("Done") //🐶
}
```



#CodeMeshLDN

```
fun downloadImage(imageName: String) {
    val url = createUrl(imageName) //🐱
    a(url)
}
fun downloadImage(imageName: String) {
    val url = createUrl(imageName) //🐶
    a(url)
}
fun a(url: String) {
    println("URL is $url")
    val bits = fetchImage(url) //🐶
    b(bits)
}
fun a(url: String) {
    println("URL is $url")
    val bits = fetchImage(url) //🐱
    b(bits)
}
fun b(bits: ByteArray) {
    println("Size: ${bits.size}") //🐱
    saveImage(bits)
    c()
}
fun b(bits: ByteArray) {
    println("Size: ${bits.size}") //🐶
    saveImage(bits)
    c()
}
fun c() {
    println("Done") //🐶
}

fun c() {
    println("Done") //🐱
}
```



#CodeMeshLDN

```
fun downloadImage(imageName: String) {
    val url = createUrl(imageName) //🐶
    a(url)
}
fun a(url: String) {
    println("URL is $url")
    val bits = fetchImage(url) //🐶
    b(bits)
}
fun downloadImage(imageName: String) {
    val url = createUrl(imageName) //🐱
    a(url)
}
fun a(url: String) {
    println("URL is $url")
    val bits = fetchImage(url) //🐱
    b(bits)
}
fun b(bits: ByteArray) {
    println("Size: ${bits.size}") //🐱
    saveImage(bits)
    c()
}
fun b(bits: ByteArray) {
    println("Size: ${bits.size}") //🐶
    saveImage(bits)
    c()
}
fun c() {
    println("Done") //🐱
}


fun c() {
    println("Done") //🐶
}
```

#CodeMeshLDN

Coroutines make everything better

```
fun main() {
    runBlocking {
        repeat(100_000) {
            launch {
                println("$it")
            }
        }
    }
}
```

```
func main() {
  var wg sync.WaitGroup
  wg.Add(100_000)
  for i := 0; i < 100_000; i++ {
     go func(n int) {
        fmt.Println(n)
        wg.Done()
     }(i)
  }
  wg.Wait()
}
```

This code prints numbers between 0 and 99,999

Good luck doing that with threads

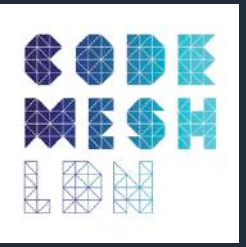
#CodeMeshLDN

Coroutine dispatchers

```
fun main() {
    runBlocking {
        repeat(100_000) {
            launch {
                println("$it")
            }
        }
    }
}
```

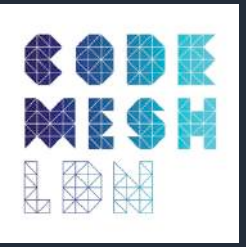
```
fun main() {
    runBlocking {
        repeat(100_000) {
            launch(Dispatchers.Default) {
                println("$it")
            }
        }
    }
}
```

Again, we print numbers from 0 to 99,999, now using all available CPU cores



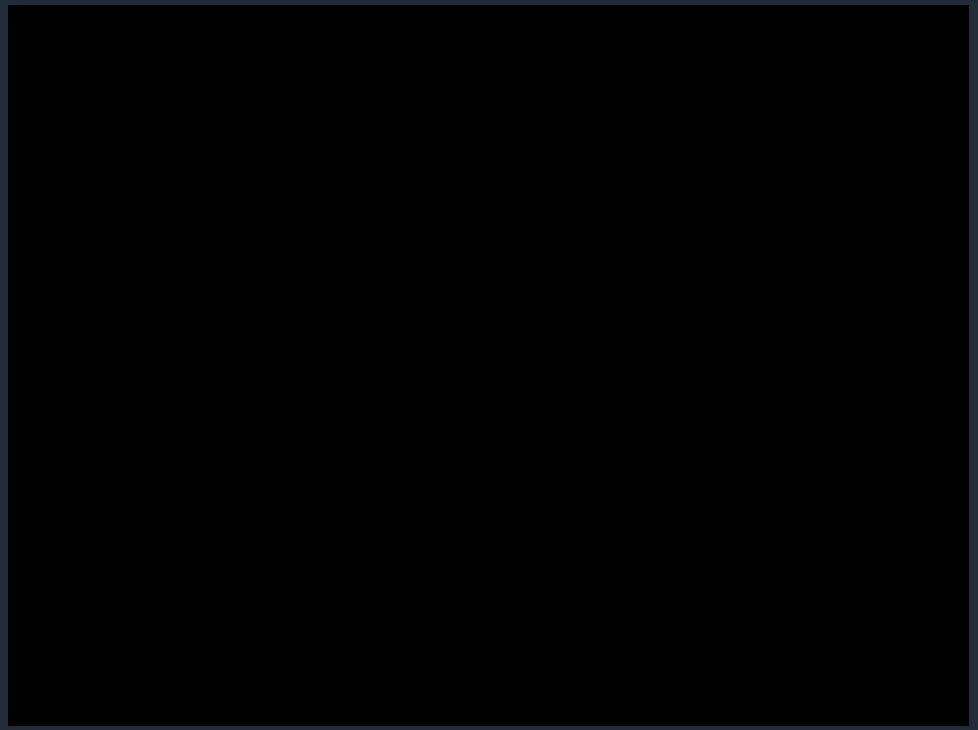
Let’s visualize!

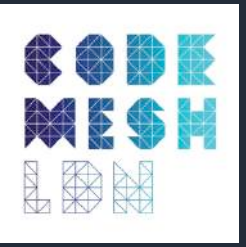
#CodeMeshLDN



#CodeMeshLDN

Let’s visualize!

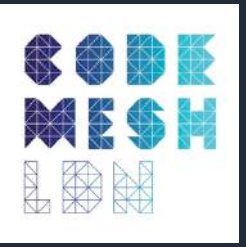




#CodeMeshLDN

Coroutines summary

- Coroutines are broken into smaller tasks, continuations
- Continuations are executed on a thread pool
- They aren’t spread evenly across CPU cores
- This pattern has a name, and it’s called Multi-Reactor



#CodeMeshLDN

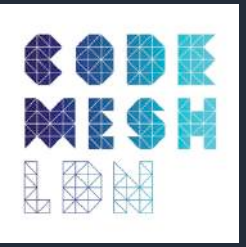
Divide and conquer

Say you want to fetch all images from a certain Internet page

You'll have to:

1. Fetch the HTML
2. Parse it to find all image tags
3. Download each image
4. Save it to disk

And of course you'd like to do in concurrently



#CodeMeshLDN

Divide and conquer

Fetching and parsing is something which is done only once

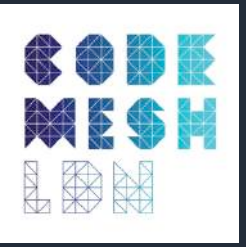
But there can be many images on the same page

We'd like to distribute this work

We can use produce(), which starts a coroutine and binds it to a channel for that

```
fun scrap(url: URL): ReceiveChannel<String> = produce {
    val html = url.fetchAsHtml()
    val links = parseLinks(html)

    for (link in links) {
        send(link)
    }
}
```

Then we send all the links over this channel



#CodeMeshLDN

Message passing

Channels are a way to communicate between coroutines

```
fun scrap(url: URL): ReceiveChannel<String> = produce {
    val html = url.fetchAsHtml()
    val links = parseLinks(html)

    for (link in links) {
        send(link)
    }
}
```

If more than one coroutine listens to the same channel, only one wins each time

```
fun downloadImage(links: ReceiveChannel<String>) = launch {
    for (link in links) {
        val bytes: ByteArray = link.download()
        //TODO do something with those bytes
    }
}
```

```
fun downloadImage(links: ReceiveChannel<String>) = launch {
    for (link in links) {
        val bytes: ByteArray = link.download()
        //TODO do something with those bytes
    }
}
```

#CodeMeshLDN

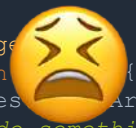
Message passing

Channels are like a BlockingQueue, but they suspend a coroutine instead of blocking a thread

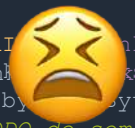
```
fun scrap(url: URL): ReceiveChannel<String> = produce {
    val html = url.fetchAsHtml()
    val links = parseLinks(html)

    for (link in links) {
        send(link)
    }
}
```

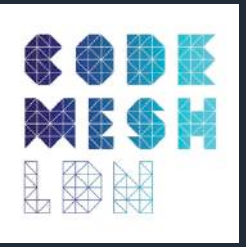
That happens only if the channel is full, of course

```
fun downloadImage(... ReceiveChannel<String>) = launch {
    for (link in ... {
        val bytes ...Array = link.download()
        //TODO do something with those bytes
    }
}
```



```
fun downloadI... ...nks: ReceiveChannel<String>) = launch {
    for (lin... ...ks) {
        val by... ...yteArray = link.download()
        //TODO do something with those bytes
    }
}
```





#CodeMeshLDN

Storing images

So, up until now we only download the images

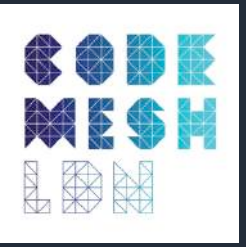
Now we need to store them

Of course we can decide that each coroutine stores the image it downloaded independently

But what about separation of concerns?

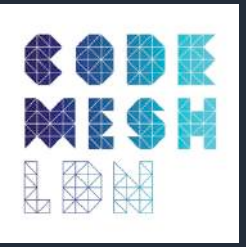
Wouldn’t it be more efficient if only one of them would do that?

```
data class SaveFileMessage(val name: String, val content: ByteArray)

fun CoroutineScope.saver() = actor<SaveFileMessage> {
   for (msg in channel) {
       saveToDisk(msg.name, msg.content)
   }
}
```



Let’s visualize!

#CodeMeshLDN



#CodeMeshLDN

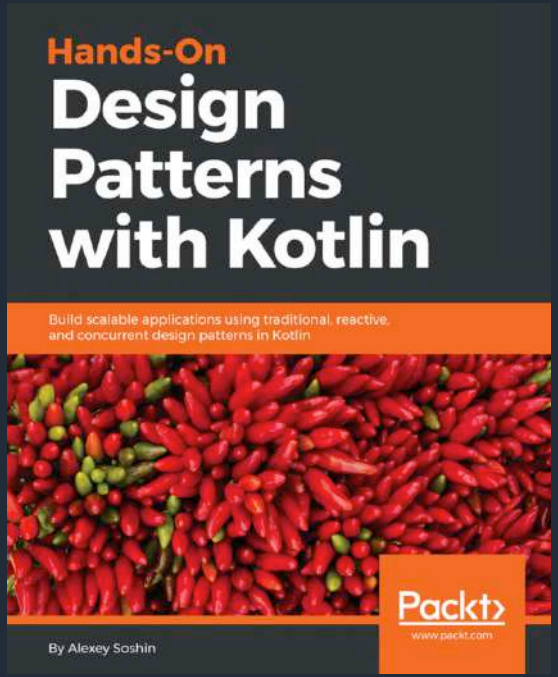
Summary

- Seeing is believing
- Concurrency is an illusion
- Goroutines are just coroutines
- Concurrency with Kotlin is awesome

#CodeMeshLDN

There’s a book about it

https://www.amazon.com/Hands-Design-Patterns-Kotlin-applications-ebook/dp/B079P7Q5HX



Thanks a lot for attending!

Code: https://github.com/AlexeySoshin/VisualizingConcurrency

Stay in touch:

- https://twitter.com/alexey_soshin
- https://stackoverflow.com/users/5985853/alexey-soshin
- https://github.com/alexeysoshin