λRPC

Simple native RPC with high order functions support

andrey.stoyan.csam@gmail.com

Inspired by MIPT Communicator

Motivation

Kotlin project needs to use:

- C++ library for complex CERN data format
  - Readed from CERN file data structure cant be trivially serialized
- Julia scientific library
  - With high-order functions
  - Functions bytecode cant be simply serialized (Kotlin → Julia)

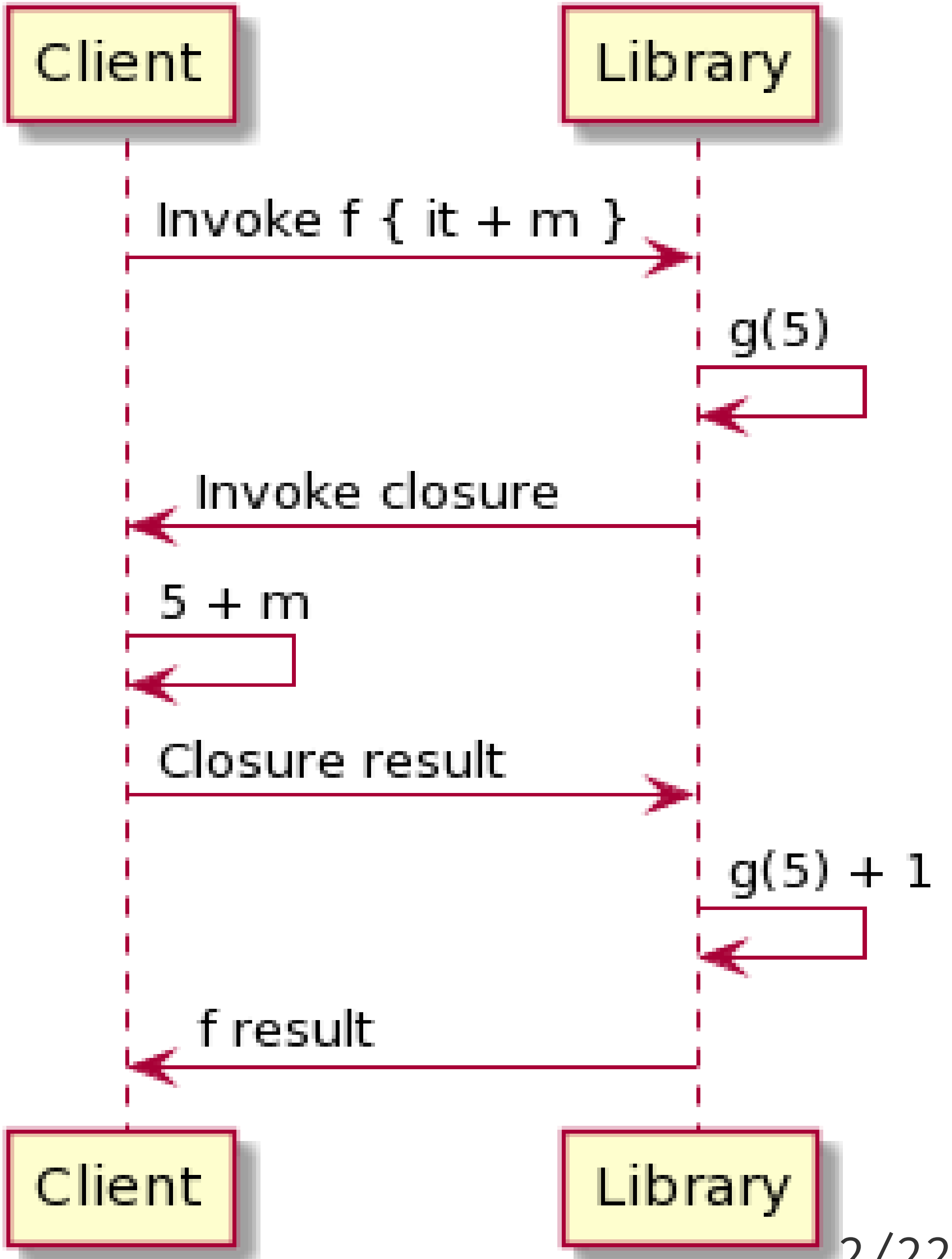
HOF implementation idea

Library

```
suspend fun f(
    g: suspend (Int) -> Int
) = g(5) + 1
```

Client

```
val m = 36
f { it + m }
>>> 42
```



Communicator vs λRPC

Communicator

- Based on ZMQ
- Big and complex, lots of effort
- Hard to port to other ecosystems:
  - Use C FFI
  - Implement ZMQ client-server
  - Tried to implement simple ZMQ client in Julia
- Iterative development

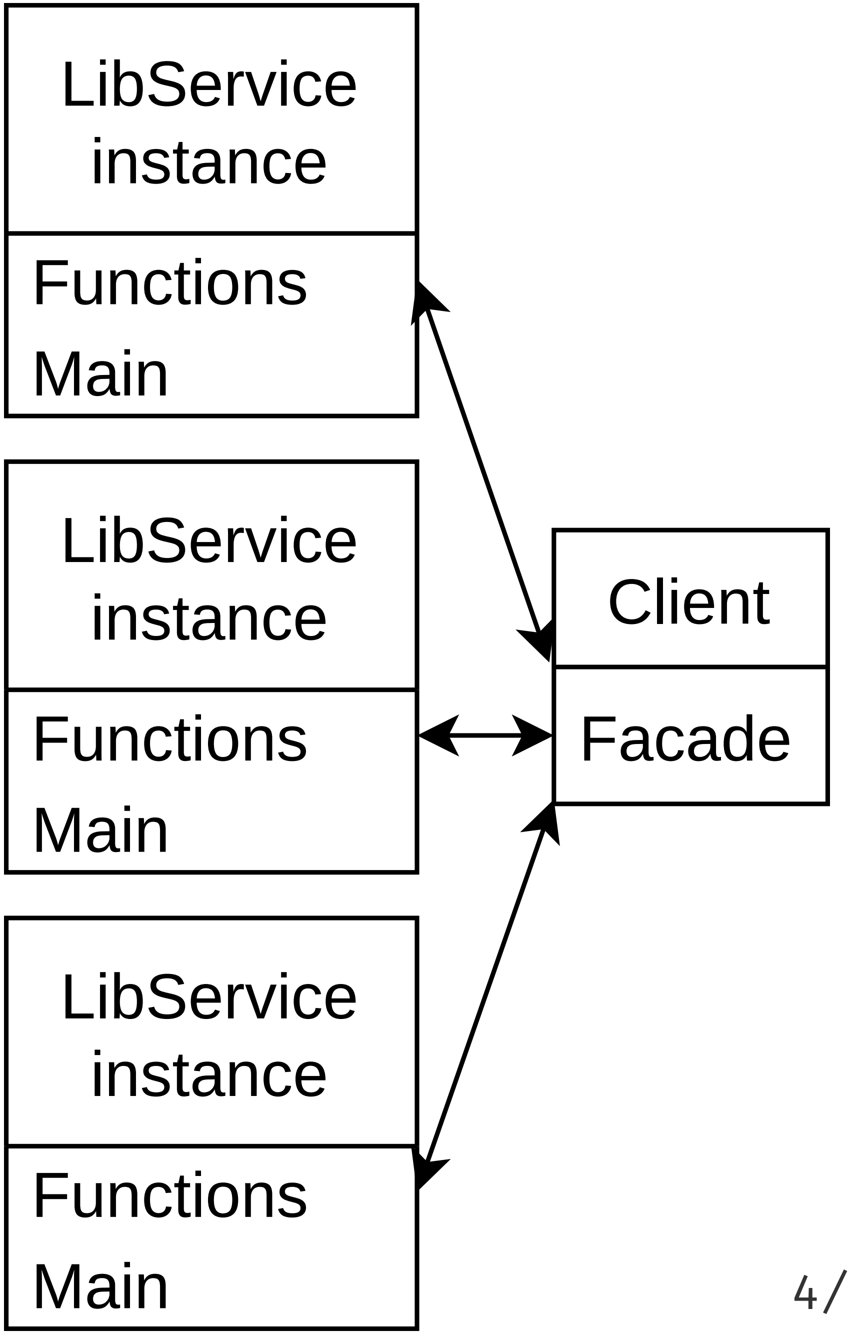
λRPC

- Based on gRPC
- Relatively small (~2kloc)
- gRPC helps to port
- Incremental development

LibService

*Let **Libservice** be a triple of*

1. Library code with functions: Julia in our example
2. Facade (declarations of exposed library functions): in Julia and in Kotlin
3. Main:
   - Starts a libservice
   - Sets up a correspondence between declarations and library functions



Example: `distance` libservice

Library

```
@Serializable
data class Point(val x: Double,val y: Double)

fun distance(a: Point, b: Point): Double =
    return sqrt((a.x - b.x).pow(2) + (a.y - b.y).pow(2))
```

Facade

```
val conf = Configuration(serviceId = serviceId1)
val distance by conf.def<Point, Point, Double>()
```

Main

```
fun main() = LibService(serviceId1, endpoint1) {
    distance of ::distance
}.apply { start(); awaitTermination() }
```

Example: `distance` client

```
import lib.facade.distance

val serviceDispatcher = serviceDispatcher(
    serviceId1 to endpoint1,
)

fun main(): Unit = runBlocking(serviceDispatcher) {
    println(distance(Point(9.0, 1.0), Point(5.0, 4.0)))
}
```

Passing high-order functions (HOFs)

```
suspend fun eval5(f: suspend (Int) -> Int): Int = f(5)
val eval5 by conf.def(f1<Int, Int>(), d<Int>())

val m = 34; eval5 { it + m }
```

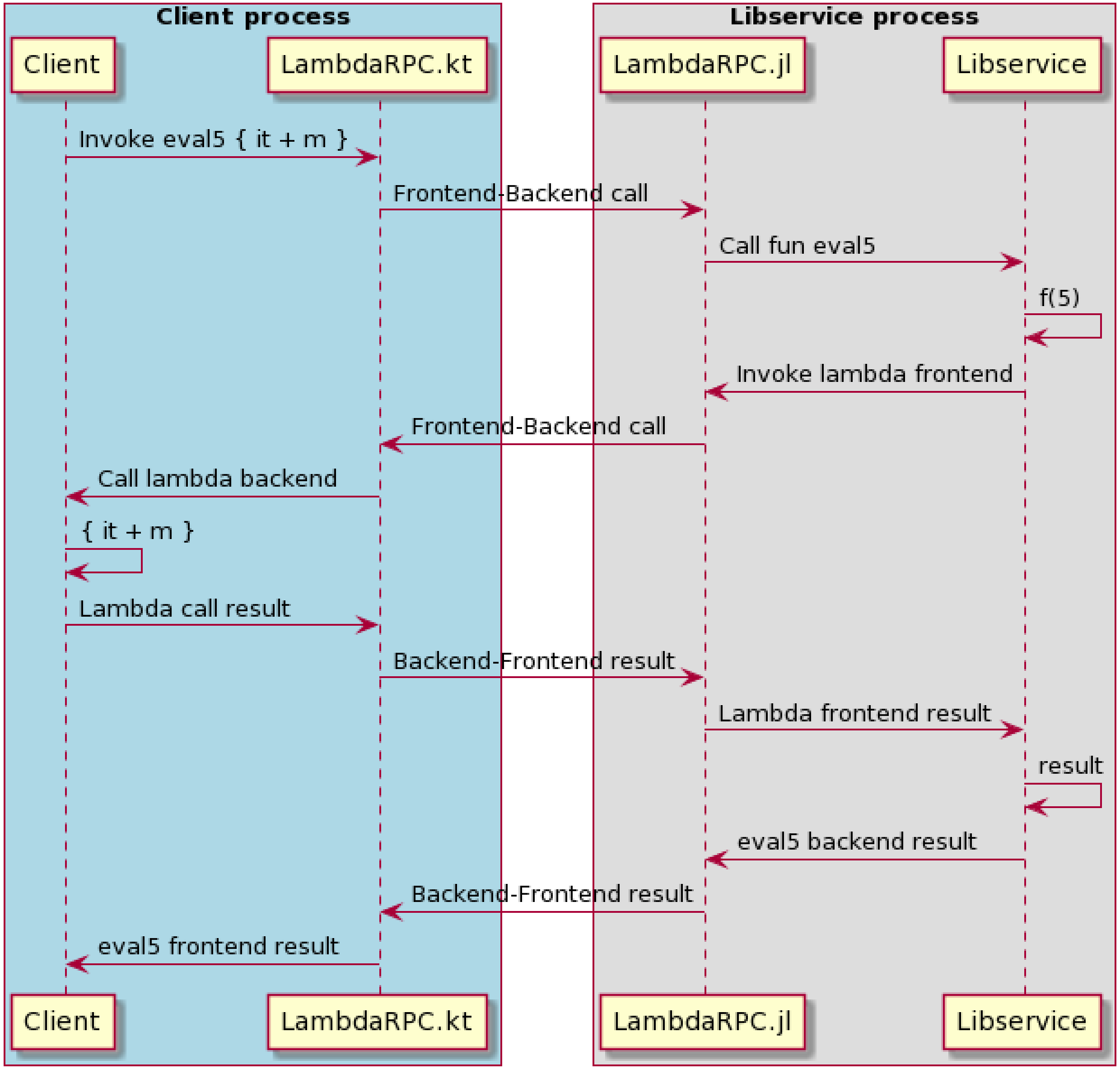
Function encoding

- Function + argument decoders → **Backend function**
- Send backend function identifier

Function decoding

- Received backend function identifier + argument encoders → **Frontend function** — a callable proxy object that communicates with the backend function on invocation

Passing high-order functions (HOFs)



Example: basic HOF and currying

Library

```
suspend fun eval5(f: suspend (Int) -> Int): Int = f(5)

fun specializeAdd(x: Int): suspend (Int) -> Int = { it + x }
```

Facade

```
val eval5 by conf.def(f1<Int, Int>(), d<Int>())

val specializeAdd by conf.def(d<Int>(), f1<Int, Int>())
```

Client

```
val m = 34
eval5 { it + m }

specializeAdd(5)(37)
```

Example: custom coder

Library

```
class NumpyArrayCoder<T> : DataCoder<NumpyArray<T>> {
    override fun encode(value: NumpyArray<T>): ByteString = ...
    override fun decode(data: ByteString): NumpyArray<T> = ...
}

fun transformArray(arr: NumpyArray<Double>): NumpyArray<Double> =
    ...
```

Client

```
val transformArray by conf.def(
    NumpyArrayCoder<Double>(), NumpyArrayCoder<Double>()
)
```

Example: `normFilter`

Library

```
suspend fun normFilter(
    xs: List<Point>,
    p: suspend (Point, suspend (Point) -> Double) -> Boolean
) = xs.filter { po -> p(po) { sqrt(it.x.pow(2) + it.y.pow(2)) } }
```

Facade

```
val normFilter by conf.def(
    d<List<Point>>(),
    f2(d<Point>(), f1<Point, Double>(), d<Boolean>()),
    d<List<Point>>()
)
```

Client

```
normFilter(ps) { po, norm -> 2 <= norm(po) }
```

Example: lazy pipeline (1)

Additional definitions

```
typealias Accessor<R> = suspend () -> R

fun <A, B, R> adapt(
    f: suspend (A, B) -> R
): suspend (Accessor<A>, Accessor<B>) -> Accessor<R> = { a, b ->
    require(a is ClientFunction)
    require(b is ClientFunction);
    {
        coroutineScope {
            val aa = async { a() }
            val bb = async { b() }
            f(aa.await(), bb.await())
        }
    }
}
```

Example: lazy pipeline (2)

Library & Main

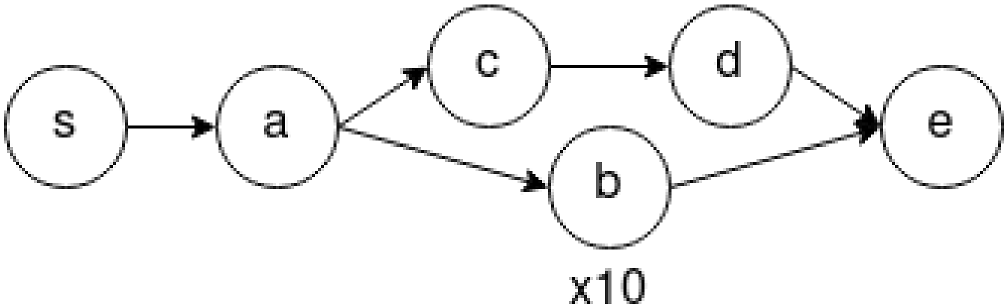
```
fun source(): Int = 1
fun e(x: Int, y: Int): Int = x + y
...

fun main() = LibService(
    serviceId, Endpoint("localhost", port)
) {
    ss of adapt(source)
    ee of adapt(e)
    ...
}.apply { start(); awaitTermination() }
```

Example: lazy pipeline (3)

Client

```
fun main(args: Array<String>) = runBlocking(
    ServiceDispatcher(serviceId to args.map {
        Endpoint("localhost", it.toInt())
    })
) {
    val s = ss(); val a = aa(s)
    val b = List(10) { bb }.fold(a) { b, f -> f(b) }
    val c = cc(s, 2); val d = dd(c); val e = ee(b, d)
    println("The answer is: ${e()}")
}
```



LambdaRPC.jl

Facade

```
using LambdaRPC

@facade "f74127d2-d27f-4271-b46e-10b79143260e" begin
    add5::Int => Int
end
```

Client

```
using LambdaRPC
using Lib

function main()
    setendpoint(lib, "localhost", 8088)
    println(add5(37))
end
```

Service as a library

- λRPC does not use standalone declarations to generate code (native). It uses library-specific data structures and default or custom coders instead.
- Functions can receive and return other functions as first-class objects. Provided lambdas are executed on the client side, so they can easily capture state and be "sent" to the other language process.

All of it makes multi-process communication smooth enough to recognize remote service as a common library.

HOF use cases

- Communication protocol simplification:
  - Service function can easily request additional information in some cases
  - Reduce service code duplication: make HOF and receive specific operations from the client
- Security:
  - Send closures operating on the sensitive data instead of the data
  - Provide computational resources as a library of functions that are parametrized by client lambdas instead of receiving client’s code and executing it

LambdaRPC.kt

- README
- issues

Service-decomposition purposes

- Code execution in different containers or on various hardware (GPU for instance)
- Parallel execution of independent tasks
- Communication with code written in other language
- Rerun subtasks in case of failures or resume using some state snapshot
- Microservice architecture

High order functions (HOF) use-cases (1)

- Communication protocol simplification:
  - Service function can easily request additional information in some cases
  - Reduce service code duplication: make HOF and receive specific operations from the client
- Interactive computations:
  - receive client lambda, provide information about computation (loss function value for instance)
  - lambda cancels computation (machine learning process) if something is not good

High order functions (HOF) use-cases (2)

- Security:
  - Send closures operating on the sensitive data instead of the data itself
  - Provide computational resources as a library of functions that are parametrized by client lambdas instead of receiving client's code and executing it
- Computation location dynamic choice: compute something using amount of data on a client or send data to the server and compute there

High order functions (HOF) use-cases (3)

- Load balancing: task is done, request new via client's lambda
- Stateful streaming computations: nodes provide theirs lambdas for a mapper