Towards a cross-platform, polyglot implementation of *Aggregate Computing* in ScaFi3

Luca Tassinari

25/03/2025

Motivations

- Aggregate Computing span heterogeneous devices and platforms;
- Several implementations of AC exist for different programming languages to:
    - target different platforms and environments;
    - leverage unique strengths of the host programming languages;

However:

- Each of these were developed from scratch, with no code reuse and compatibility in mind;
- No common framework led to fragmentation.

Goal

Investigate the feasibility of building a framework capable of targeting multiple platforms while offering interoperability with other languages.

In particular the work focuses on:

- architectural design of a portable, interoperable layer for Aggregate programming, preserving core abstractions and full code reuse.
- interoperability and distribution strategies enabling seamless data exchange and collective execution across heterogeneous devices and language runtimes;
- evaluation of performance, API idiomaticity, and maintenance effort.

$\Rightarrow$ *Scala 3* as the perfect fit to implement AC abstractions and model in a strongly typed internal DSL.

Scala 3 cross-platform capabilities

- **JVM** (desktop, server, Android) & *Java* interop;
- **JS** via Scala.js:
    - Supported platforms:
        - Web (browser);
        - Node.js;
        - WebAssembly (experimental).
    - *JavaScript* interop via annotations, indirectly supporting *TypeScript*;
    - Mature ecosystem.
- **Native** via Scala Native:
    - Supported platforms:
        - `x86-64` and `aarch64` on Linux, macOS and Windows;
        - experimental 32-bit support;
        - suitable for SoC-based IoT devices but not microcontrollers;
    - *C* interop via annotations;
    - Growing ecosystem maturity, limited toolchain support.

ADD PRO AND CONS

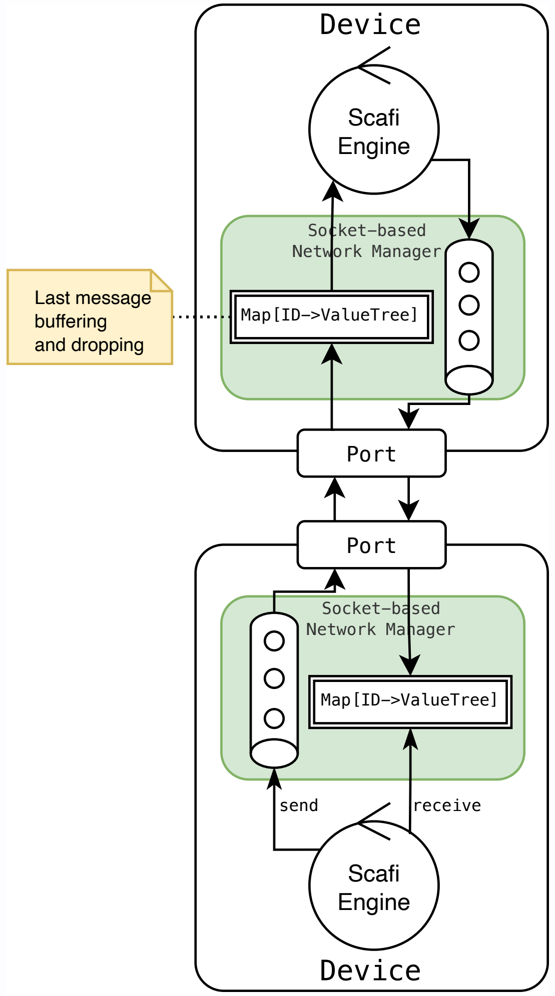
Contribution

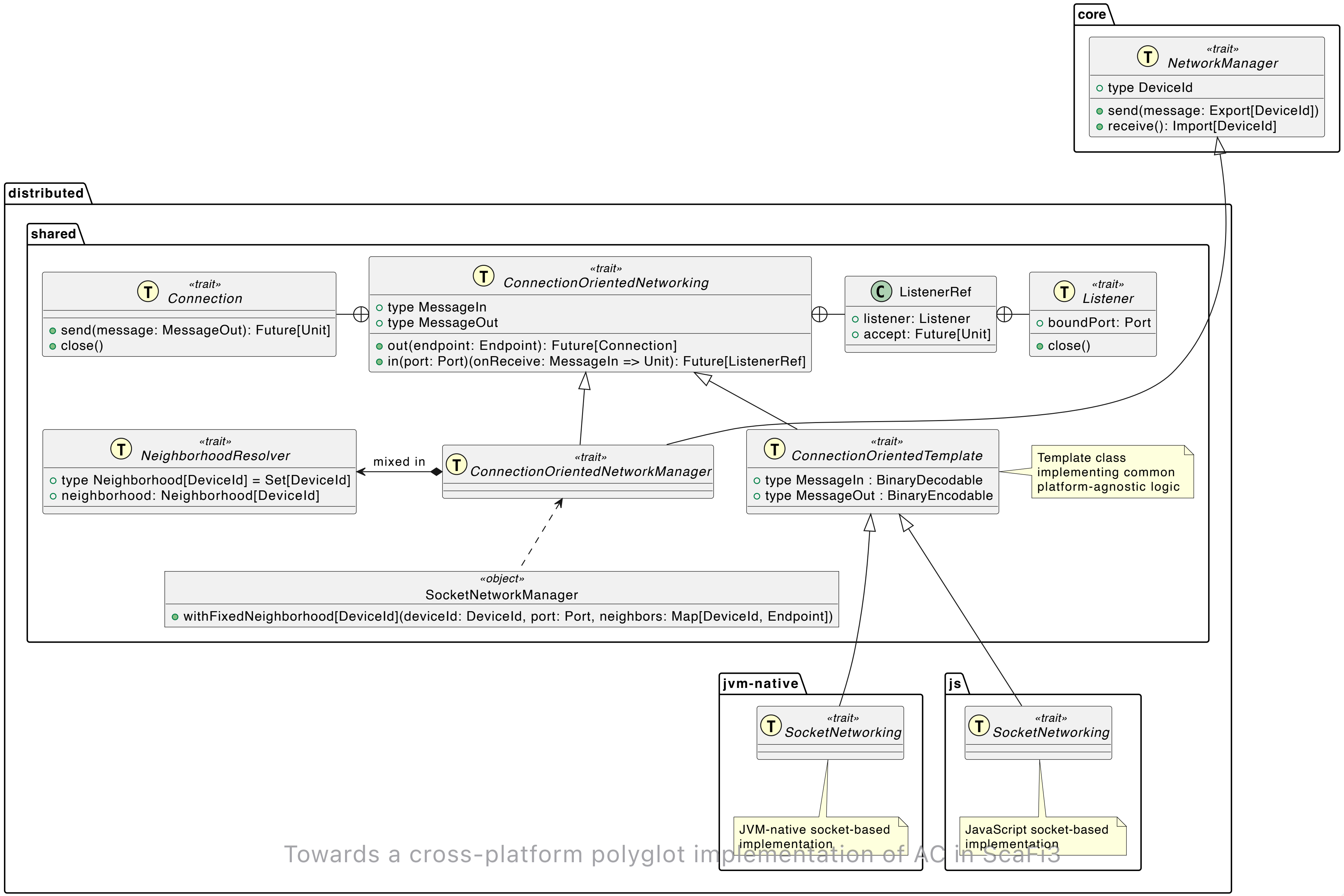
The contribution of this thesis span three main axes:

1. **Add a cross-platform *distribution* module;**
2. Add support for a general *cross-platform* and *polyglot* serialization binding;
3. Add a *cross-platform*, *polyglot* library abstraction layer.

Cross-platform distribution module

- Technology: *stream*, *TCP*-based *connection-oriented sockets*;
  - Each device is bound to a specific *endpoint* (IP + port);
  - Point-to-point connections between neighbors;
  - Neighborhood is statically *fixed* at initialization but can be extended in the future with dynamic discovery strategies;
- Support for multiple platforms: *JVM*, *JS* (Node.js), *Native*;
  - *JVM* + *Native* support via Java Standard *sockets* library;
  - *JS* support via *Node.js net* module using Scala.js type facades;
  - **Implications**:
    - shared code cannot perform blocking operations;
    - all the API is designed to be asynchronous and non-blocking using Futures;
    - the primary goal: write as much shared code as possible, minimizing platform-specific implementations.


Device
Scafi Engine
Socket-based Network Manager
Last message buffering and dropping
Map[ID->ValueTree]
Port
Port
Socket-based Network Manager
Map[ID->ValueTree]
send
receive
Scafi Engine
Device


core
«trait»
NetworkManager
type DeviceId
send(message: Export[DeviceId])
receive(): Import[DeviceId]
distributed
shared
«trait»
Connection
send(message: MessageOut): Future[Unit]
close()
«trait»
ConnectionOrientedNetworking
type MessageIn
type MessageOut
out(endpoint: Endpoint): Future[Connection]
in(port: Port)(onReceive: MessageIn => Unit): Future[ListenerRef]
ListenerRef
listener: Listener
accept: Future[Unit]
«trait»
Listener
boundPort: Port
close()
«trait»
NeighborhoodResolver
type Neighborhood[DeviceId] = Set[DeviceId]
neighborhood: Neighborhood[DeviceId]
mixed in
«trait»
ConnectionOrientedNetworkManager
«trait»
ConnectionOrientedTemplate
type MessageIn : BinaryDecodable
type MessageOut : BinaryEncodable
Template class implementing common platform-agnostic logic
«object»
SocketNetworkManager
withFixedNeighborhood[DeviceId](deviceId: DeviceId, port: Port, neighbors: Map[DeviceId, Endpoint])
jvm-native
«trait»
SocketNetworking
JVM-native socket-based implementation
js
«trait»
SocketNetworking
JavaScript socket-based implementation

An example of Scala.js facade over the Node.js `Net` class to allow interoperability with Node.js networking APIs:

```scala
@js.native
@JSImport("net", JSImport.Namespace)
object Net extends js.Object:

  /** A factory function which creates a new Socket connection. */
  def connect(port: Int, host: String): Socket = js.native

  /** A factory function which creates a new TCP or IPC server. */
  def createServer(connectionListener: js.Function1[Socket, Unit]): Server = js.native
```

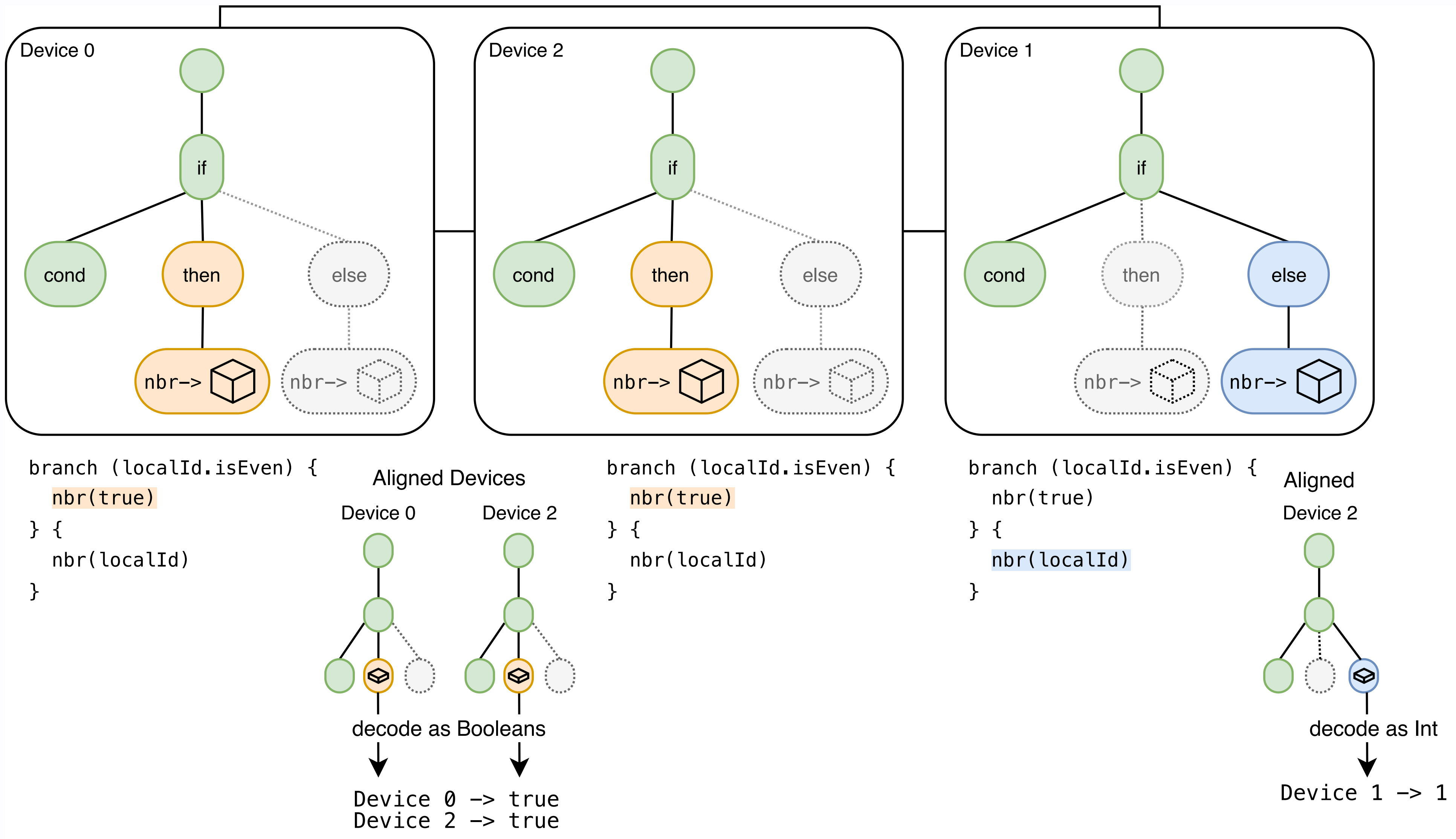
Contribution

The contribution of this thesis span three main axes:

1. Add a cross-platform *distribution* module;
2. **Add support for a general *cross-platform* and *polyglot* serialization binding;**
3. Add a *cross-platform*, *polyglot* library abstraction layer.

Cross-platform and polyglot serialization binding

- Devices exchange (ID, Value Tree) pairs
- When exchanging data, values are inserted into the Value Tree encoded using a specific serialization format
    - This is possible since, in the context of an `exchange`, the type information of the value is known
- When receiving data, the Value Tree is decoded but values remain encoded in their serialized format
- Only when the corresponding exchange in the aggregate program is evaluated the value is decoded
    - Again, this is possible since the type information of the expected value is known at that point
- Technically, this is achieved via a combination of Scala 3 *type classes* and *type lambdas* that abstract over the serialization format and allow to cleanly express encoding and decoding requirements as *type bounds*.



```
branch (localId.isEven) {
  nbr(true)
} {
  nbr(localId)
}
```

```
branch (localId.isEven) {
  nbr(true)
} {
  nbr(localId)
}
```

```
branch (localId.isEven) {
  nbr(true)
} {
  nbr(localId)
}
```

```
Device 0 -> true
Device 2 -> true
```

```
Device 1 -> 1
```

- Encodable and Decodable type classes for encoding and decoding generic messages from/to a format (e.g., JSON, binary, ...)

```scala
/** A type class for encoding messages. */
trait Encodable[-From, +To]:

  /** @return the encoded value in the target type. */
  def encode(value: From): To

/** A type class for decoding messages. */
trait Decodable[-From, +To]:

  /** @return the decoded data in the target type. */
  def decode(data: From): To

/** A type class for encoding and decoding messages. */
trait Codable[Message, Format] extends Encodable[Message, Format] with Decodable[Format, Message]

// Type alias for express encodable and decodable capabilities as type bound on values

type EncodableTo[Format] = [Message] =>> Encodable[Message, Format]

type DecodableFrom[Format] = [Message] =>> Decodable[Format, Message]

type CodableFromTo[Format] = [Message] =>> Codable[Message, Format]
```

Every function dealing with, possibly, values distribution add as type bound a Codable instance

```scala
// inside this function body, Values can be both encoded and decoded
override def xc[Format, Value: CodableFromTo[Format]](
  init: SharedData[Value],
)(
  f: SharedData[Value] => (SharedData[Value], SharedData[Value]),
): SharedData[Value] =
  alignmentScope("exchange"): () =>
    val messages = alignedMessages.map { case (id, value) => id -> value }
    val field = Field(init(localId), messages)
    val (ret, send) = f(field)
    writeValue(send.default, send.alignedValues)
    ret

// extracts the aligned values from the Value Tree and decode them using contextually
// available decoder for Value
def alignedMessages[Format, Value: DecodableFrom[Format]]: Map[DeviceId, Value] = ...

// add a new value into the Value Tree that will be sent to neighbors already serialized using
// contextually available encoder for Value
def writeValue[Format, Value: EncodableTo[Format]](default: Value, overrides: Map[DeviceId, Value]): Unit =
  ...
```

- In non-distribution scenarios, like simulation or local testing, encoding and decoding is a no-op:

```scala
given forInMemoryCommunications[Message]: Codable[Message, Message] with
  inline def encode(msg: Message): Message = msg
  inline def decode(msg: Message): Message = msg
```

- Useful for API using exchange primitive only for state evolution, like `evolve`, where we do not want to force users to provide encoders/decoders for their values:
    - network managers needs to be implemented to ignore any non-Format values

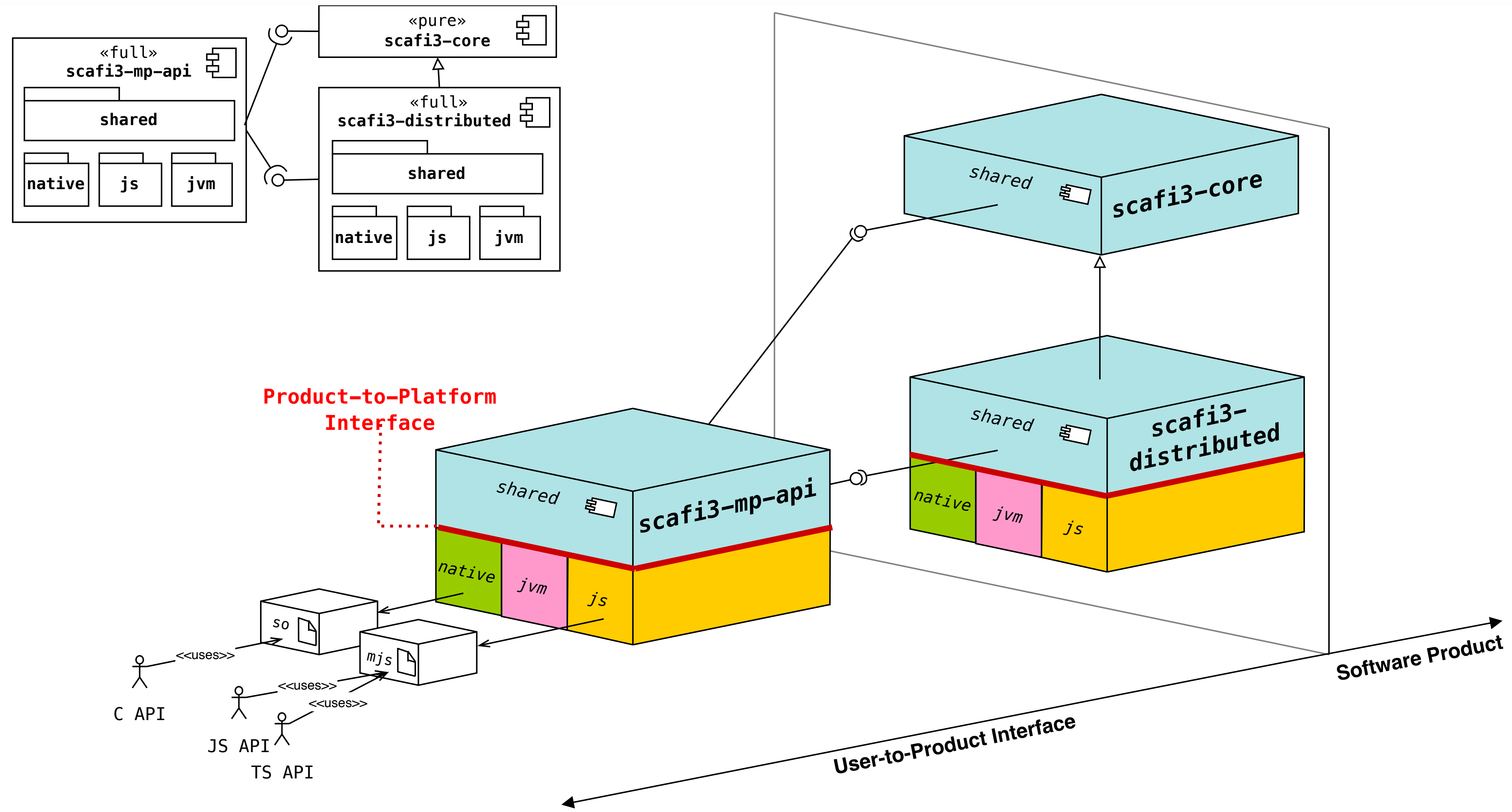
```scala
override def evolve[Value](initial: Value)(evolution: Value => Value): Value =
  // `exchange` is called only to update the self-value: `None` is shared with neighbors, so an in-memory
  // codec is enough; non-in-memory network managers will ignore it since it is not serialized.
  exchange(None)(nones =>
    val previousValue = nones(localId).getOrElse(initial)
    nones.set(localId, Some(evolution(previousValue))),
  )(using Codables.forInMemoryCommunications)(localId).get
```

Contribution

The contribution of this thesis span three main axes:

1. Add a cross-platform *distribution* module;
2. Add support for a general *cross-platform* and *polyglot* serialization binding;
3. **Add a *cross-platform*, *polyglot* library abstraction layer.**

Primary problem: both Scala Native and Scala.js


«pure»
scafi3-core
«full»
scafi3-mp-api
shared
native
js
jvm
«full»
scafi3-distributed
shared
native
js
jvm
Product-to-Platform Interface
shared
scafi3-core
shared
scafi3-distributed
native
jvm
js
shared
scafi3-mp-api
native
jvm
js
so
mjs
<<uses>>
<<uses>>
<<uses>>
C API
JS API
TS API
Software Product
User-to-Product Interface