FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# ASSIGNMENT OF BACHELOR'S THESIS

| | |
|---|---|
| **Title:** | An actor model implementation for the OCaml programming language |
| **Student:** | Narek Vardanjan |
| **Supervisor:** | Ing. Filip Křikava, Ph.D. |
| **Study Programme:** | Informatics |
| **Study Branch:** | Web and Software Engineering |
| **Department:** | Department of Software Engineering |
| **Validity:** | Until the end of summer semester 2020/21 |

## Instructions

In the actor model, an actor is the fundamental unit of computation that encapsulates state and behavior, communicating exclusively by exchanging messages. Because the state is completely encapsulated, invisible to the outside world, the use of this model has been popular for building highly concurrent applications.

The goal of this thesis is to design, implement and test an actor system for OCaml. The system should provide an API for creating a network of actors and a runtime system facilitating their communication and orchestration. Actors will be organized in a hierarchy allowing actor supervision and monitoring. The messaging system shall provide an at-most-once delivery guarantee. Actors should use location transparency allowing them to communicate across different runtimes. The implementation should be lightweight focused on low actor memory footprint and high message throughput. The API should leverage OCaml language features such as pattern matching and immutable values.

## References

Will be provided by the supervisor.

| | |
|---|---|
| Ing. Michal Valenta, Ph.D. | doc. RNDr. Ing. Marcel Jiřina, Ph.D. |
| Head of Department | Dean |

Prague February 1, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

# An actor model implementation for the OCaml programming language

***Narek Vardanjan***

Department of Software engineering
Supervisor: Ing. Filip Křikava, Ph.D.

June 4, 2020

# Acknowledgements

I would like to thank Filip Křikava for seamless communication and leadership of the thesis. To my family for their emotional support and especially to my grandfather Sasha Vardanyan for his unending believe in me.

# Declaration

In Prague on June 4, 2020 . . . . . . . . . . . . . . . . . . . .

## Citation of this thesis

Vardanjan, Narek. *An actor model implementation for the OCaml programming language.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020. Also available from: ⟨`http://github.com/vnarek/jude`⟩.

# Abstract

The actor model is an abstraction for concurrent programming, that uses actors, independent units of computations. These units are spawned, so they can communicate with each other via messages and change their states accordingly to them. Messages are processed serially, which guarantees needed synchronization for a state change. Thanks to that there is no need for using synchronization primitives like locks.

The work describes the core parts of an actor model library implementation. It consists of a brief introduction to the classic concurrent model with its drawbacks. Then it introduces the actor model and its most influential flavors. Aside from the core constructs of spawning, sending messages, and state changes, the library implements additional functionality for monitoring/linking the actors and name resolving. The library is written in Objective Caml leveraging its language features.

**Keywords** ocaml, concurrent patterns, actor model, distributed systems

# Abstrakt

Aktor model je abstrakce pro konkurentní programování, která pracuje s aktory, nezávislými jednotkami. Tyto jednotky vznikají, aby si mezi sebou posílaly zprávy a následně podle nich měnily svůj stav. Zprávy aktor zpracovává postupně, čímž zaručí nutnou synchronizaci pro změnu stavu. Díky tomu se vyhne potřebě používat nízkoúrovňové synchronizační primitivy jakými jsou zámky.

Práce popisuje návrh a implementaci hlavních částí aktor knihovny. Obsahuje stručné shrnutí klasického konkurentního modelu s jeho nedostatky a popis aktor modelu včetně příkladů vlivných implementací. Kromě základních konstrukcí aktor modelu pro spawnování, odesílání zpráv a změn stavu, implementuje i rozšířené funkcionality ve formě monitorování/linkování procesů či převody jmen na adresy konkrétních aktorů. Knihovna je napsaná v Objective Caml s využitím zajimavých aspektů tohoto jazyka.

**Klíčová slova** ocaml,konkurentní vzory, aktor model, distributivní systémy

# Contents

# List of Figures

# List of Listings

# Introduction

"The free lunch is over" [1]. Major boosts in the horizontal computational power are long gone and the industry has adapted. From CPU manufacturers that keep adding cores to a single chip to data centers, which grow exponentially in the number of commodity servers. It is apparent more than ever that concurrent and distributed programming is the way to move forward.

However, concurrent programs are not easy to reason about. Classical threads operating on shared memory with locks are error-prone and difficult to use. In the case of larger systems, this low-level operating system primitives do not scale well and thus researchers were seeking new, higher-level abstractions. One of them is the actor model.

The actor model of actors which communicate with one another through message passing. An actor encapsulates its state and the only way to change it is by sending messages. Upon receiving a message, the actor decides which action to take. Because the actor processes messages sequentially, there is no need for any locking. This is what makes the actor model so appealing. The actor model is theoretical, it does not pose any restriction to what communication protocol shall be used nor to how actors shall be scheduled. Since the first industry-strength implementation in 1986, the actor model has been adapted by many [2, 3, 4], and there are many numbers of implementations for a variety of programming languages.

The work aims to design and implement the actor model for the OCaml programming language to provide an alternative to the standard monad based concurrency inspired by Haskell. OCaml is an ML family, multi-paradigm language, which delivers functional programming mixed with OOP concepts. It is statically typed with parametric polymorphism, type inference, and a strong modular system. The resulting library, Jude, provides the core actor functionality with additional features like creating actor hierarchies for supervision or name resolving.

The thesis is structured into four main parts. The state-of-the-art chapter describes the actor model generally and compares common actor model implementations. Analysis and Design describe the architecture of the library in detail by documenting used components and main libraries. Next comes the Realization chapter describing implementation details. The actor library is then compared to other actor libraries in the Comparison to Other Actor Libraries chapter. Lastly, the Future Work chapter sets the direction for future development.

Chapter 1

# Background

This chapter provides a brief overview of the classical concurrency model, the inherent problems it has, and how they are addressed. Next, it introduces the actor model. Finally, it provides a quick overview of the OCaml programming language.

## 1.1 Concurrency

In general, concurrency allows one to execute multiple sequential processes at the same time. It can be done by interleaving those processes, which appears like they are proceeding all at the same time. This is the main difference between terms concurrency and parallelism. Parallel programs physically run at the same time, leveraging parallel hardware (multi-core processors) [5].

As seen in Figure 1.1 this simulation is beneficial. Well-written concurrent programs can be more effective than their sequential counterparts even without multiple cores because programs tend to wait for resources not computing anything. While one task wait for a resource another could proceed computation. It is defined as asynchronous I/O and languages with GIL usually use this kind of concurrency.

### 1.1.1 The correctness of a concurrent program

The correctness of a concurrent program is defined by properties which it holds. A correct concurrent program must fulfill two properties of computation:

- *Safety properties – The property must always be true*
- *Liveness properties – The property must eventually become true*

[5]

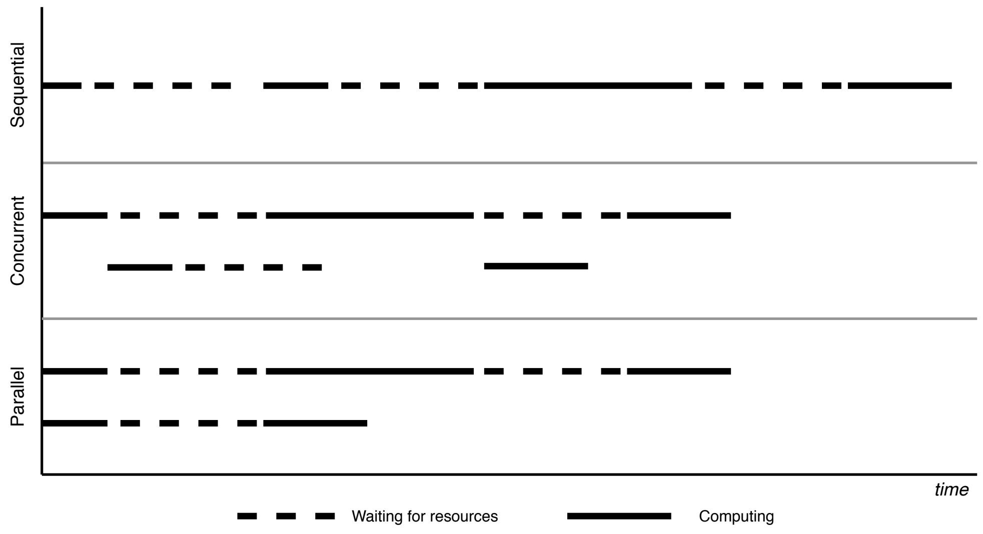

Figure 1.1: The difference in sequential, concurrent and parallel execution

Writing correct concurrent programs is a challenging problem. Without synchronization in a critical section, programs tend to violate the mutual exclusion principle [6], which makes them not safe. Synchronization in the classical concurrency is provided by using locks and constructs built on them.

But locks have their own set of problems, that need to be addressed. Misusing locks results in needless serialization, or even deadlocks. Deadlocks violate the liveness property of the program, so a concurrent program that has deadlocks is not correct. Apart from this, it is impossible to compose two thread-safe functions into a higher-level thread-safe function [7]. This results in the creation of unsafe functions that are then wrapped by their lock counterparts. Bugs are even harder to track when multiple locks are introduced in the concurrent system. Programmers need to enforce that all locks are ordered in the same arbitrary way, otherwise, deadlocks will occur [7].

Concurrency has additional problems that need to be solved to guarantee the robustness of a program. Examples of such problems are starvation or unfairness. These problems are explained in greater detail in basic texts about concurrency [5, 8, 9].

The listing 1.1 shows how classical concurrency looks like. It defines heat aggregator, which receives new records of temperature taken from a sensor. After its window is full, it gives average temperature to callers of `computeAverage` method. Methods `lock` and `unlock` wrap critical sections of the code. Try/finally is needed to ensure that locks are released even when an exception is thrown. As seen in the listing, even simple programs relying on locks are cluttered with synchronization primitives.

```
import java.util.concurrent.locks.{Lock, ReentrantLock}

class HeatAggregator(val size: Int) {
  val lock: Lock = new ReentrantLock()
  val window = new Array[Float](size)
  var actualField = 0
  var full = false
  def newRecord(rec: Float) : Unit = {
    lock.lock()
    try{
      actualField = actualField + 1
      if (size == actualField) {
        full = true
      }
      window(actualField % size) = rec
    } finally {
      lock.unlock()
    }
  }
  def computeAverage() : Option[Float] = {
    lock.lock()
    try{
      if (full) Some(window.sum / size) else None
    } finally {
      lock.unlock()
    }
  }
}
```

Listing 1.1: Classic concurrency in Scala

### 1.1.2 Execution of concurrent processes

A continuation of a concurrent program is arranged by the process called scheduler. The job of the scheduler is to give access to computation time on the CPU to sequential processes that need it [5]. A well-written scheduler should exhibit fairness mentioned earlier.

There are two possible implementations of a scheduler called preemptive and cooperative (non-preemptive). In preemptive scheduling, sequential processes are switched by scheduler without their consent. Cooperative scheduling leaves decisions about switching on the scheduled process, which makes the scheduler much easier to implement and outperforms preemptive in switching speed [10]. Cooperative scheduling has its drawbacks too, when processes are not implemented well and do not yield control back to the scheduler when blocked. For this reason, cooperative schedulers are used a lot in user-level programs where a programmer has every process under control. User-level preemptive threads are not as common but there are exceptions. In a real-world situation, both preemptive and cooperative scheduling is used.

### 1.1.3 Inter-process communication

The hard part of concurrency lies in communication. Processes in concurrent programs communicate to exchange information or to access a resource. If poor measures are taken to synchronize concurrent programs, then problems described in the Concurrency chapter arise.

One way to implement communication between processes is by sharing memory location which is used for read/write operations. This could be described as communication using a resource acquisition. Implementation based on resource acquisition is prone to mutual exclusion violations and deadlocks. It is hard to achieve a high level of concurrency without sacrificing readability and correctness.

Message passing is another principle used for communicating between processes. Every process owns its private memory and communication happens via messages that are sent there[5, 11]. That means there is no need for synchronization of memory access.

Passing messages is synchronous or asynchronous. In the synchronous scenario, both sender and receiver need to be ready for communication. The sender is going to wait for the receiver or the other way around. Communication with itself is not possible in synchronous messaging. Sending and receiving are atomic operations and the process cannot execute both at the same time. This can be overcome by introducing a buffer process [12]. Synchronous messaging could be likened to a phone call.

Alternatively, mail is a form of asynchronous communication. The sender sends a message to a buffer which is a not blocking operation. The receiver then reads from buffer sequentially, blocking when no additional messages are available. This type of communication is prone to deadlocks when communicating because there is no blocking introduced [13]. It is still possible for actors to stop communicating because both wait for a message mutually, but no resources that other wants are held indefinitely.

## 1.2 The actor model

The actor model is a theoretical computation model that uses passing messages between actors as a form of communication. First introduced in 1973 paper by Carl Hewitt [14], its original use case was to model artificial intelligence (expertise) systems, but it soon found its way to concurrent and distributed applications.

### 1.2.1 Actor

Everything in the actor model is an actor [15]. Actors encapsulate a state and forbid all outside modifications. They run independently of another, so one's failure is not transitive to others. The behavior of an actor tells how it should handle the next message. Handling a message means reacting to it with a change of state or behavior. Only one message is processed at any given time making an actor's behavior execution completely sequential.

Presumption of implementation details is not defined in the theoretical model. For example, in the core model itself, there is no concept of actors having mailboxes. The reason is that if an actor needs to have a mailbox and everything is an actor, then a mailbox must be an actor itself needing its mailbox, leading to a cycle [16].

An actor can spawn additional actors, which could be used to create hierarchies of them. No reference to the actor itself should exist, that could violate encapsulation. All communication is done using an actor address. Addresses are obtainable by creating new actors, getting the address in a message sent to him, or by already having the address in a state [13].

Messages received by the actor are anonymous. Just by receiving a message alone, no one can deduce the sender. If the sender wants to be contacted, he should send his address as part of the messages. Addresses sent in messages are called *customers*.

Message delivery is non-deterministic. Suppose actors $A_0$ and $A_1$ exists. If $A_0$ sends messages $m_0$ and $m_1$, then no certainty is given that messages will be delivered in the order they were sent [14].

### 1.2.2 Basic constructs

To write comprehensive programs with actors some basic constructs need to be defined. Examples of minimal languages using these constructs can be found in Agha's 1990 book [13].

**Declaration** is useful if the default behavior of an actor needs to be specified. For that purpose *declare* construct was invented. It takes name and behavior as parameters and registers that to the actor system. This construct could be likened to the variable declaration of the first-class function.

**Spawn** takes a *declaration* and uses that to create a new actor definition. Spawn additionally creates an address and binds the actor with it. Then it returns the given address.

**Send** is used to send messages to a given address. Send never blocks and does not have a return value. Send does not forbid sending messages to itself. Thanks to that it is possible to define recursive actors. There is no possible way of knowing if the actor received the message. Addresses passed to the send function could be of an actor that does not exist.

**Become** serves the purpose of changing behavior for the next message. Become is a function that takes behavior function for the next message. State and information known to actors could be changed just by using become construct.

The listing 1.2 shows the heat aggregator program written using the actor model. As we can see there is no explicit synchronization even when we define a mutable state. The code is more compact and uncluttered.

```
abstract class HeatMessage
sealed case class NewRecord(rec: Float) extends HeatMessage
sealed case class ComputeAverage() extends HeatMessage
case class ComputeRespond(o: Option[Float])

class ActorHeatAggregator(size: Int) extends Actor {
  val window = new Array[Float](size)
  var actualField = 0
  var full = false
  override def receive = {
    case NewRecord(rec) =>
      actualField = actualField + 1
      if (size == actualField) {
        full = true
      }
      window(actualField % size) = rec
    case ComputeAverage() => sender() ! ComputeRespond(
      if (full) Some(window.sum / size.toFloat) else None
  }
}
```

Listing 1.2: Actor model Akka in Scala

### 1.2.3 Actor model vs CSP

The actor model is not the only higher-level abstraction for concurrency. Another popular model is Communicating sequential processes. The CSP is a model for understanding concurrent programs, which relies heavily on synchronous communication. A process is an isolated sequential code that runs independently from others. In the first version of CSP communication was done by named processes. Every process had its name and communication with that process was done using that [17].

The second version defined channels, first-class objects used for communication between processes [12]. This change affected how CSP could communicate. For example, it was possible to receive messages from multiple sources.

Thanks to CSP a lot of programming languages have embraced the notion of channels for building concurrent programs. Rob Pike was highly inspired by Hoare's CSP papers when designing Go programming language [18].

Both CSP and actor model relies heavily on communication via messages. Unlike the actor model, CSP is synchronous even when sending messages. That means for a message to be sent both producer and consumer must be ready to do so. Processes can read from multiple channels, which is not possible in the pure actor model (it is possible in Cloud Haskell implementation for example [19]). Channels could be simultaneously read from multiple processes that violate the actor's one address per process rule.

CSP is much less memory-intensive because by both reader and writer blocking only one message will be at any given time in the "mailbox". It is not useful in distributed environments, because both ends need to be synchronized to exchange messages, which can slow down communication. For this reason, the actor model was chosen for the implementation in place of CSP.

### 1.2.4 Similarities with object-oriented programming

*I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging"...* – Alan Kay [20]

Conceptually, actors in the actor model look like objects in OOP. It is because the actor model and the object-oriented programming share an interesting history. The model was influenced by a presentation Alan Kay gave to the MIT AI Lab about his objective language Smalltalk [21].

Objective programming is a paradigm where everything is an object [21]. Objects are defined as containers for holding data and behavior operating on that data. To change an internal state of the object message needs to be passed.

Definitions of actors and objects overlap. Objects contain attributes and methods, that are parallel to actors state resp. behavior. The internal state of both concepts is encapsulated and only communication via message passing is tolerated. Message passing in Smalltalk refers to method calling in contemporary languages. Some dynamic languages like Ruby [22], PHP[23] can use Smalltalk like message passing instead of method calls, by overriding magic method.

Differences consist mostly of the fact that the actor model was deliberately designed to tackle concurrent environments. OOP languages were not created with concurrency in mind. Message passing in OOP languages is synchronous and waits for a return value. Methods could be fired from multiple processes and no guarantees of synchronous message processing exist, so this paradigm is prone to race-conditions. The solution to this problem is comprehensive locking, which could lead to deadlocks.

The initialization of objects and actors differs too. Objects are constructed in the parent process which means they share a memory with it. In the actor model, every call to *spawn* function constructs an actor in a new process, and they are completely isolated.

## 1.3 Flavors of the actor system

The actor model is as abstract as it gets. No implementation detail is defined in the theoretical model so its no surprise that multiple flavors of actor systems were built over-time. Systems could be assigned to four main categories [24].

From every category, one actor system will be picked to represent the whole group. Implementation of small actor behavior is going to be provided with this specification: *Actor fruit giver which has only two bananas. Accepts message of type give to which he decrements the number of bananas and reply with gave the message. If there are no more bananas to give, the fruit giver starts to reply with no_more message.*

### 1.3.1 Classic actor model

The classic actor model follows the definition by Gul Agha's actors [25]. Which defined three basic primitives *spawn*, *send*, *become*. Spawn created an actor, send was used to deliver messages and become changed actual state of an actor.

The state is immutable in classic actors so the only way to change it is by creating new behavior with the *become* primitive. Messages accepted by the actor are decided dynamically at the runtime.

Agha's Sal and Act languages are examples of this implementation. Akka is another classic actor model implemented on JVM [26] for Scala language.

Actors in the Scala are objects inheriting from `Actor` class. Every actor needs to implement the `receive` method. Optionally the actor anonymous syntax could be used. Become construct works as a stack to which behavior can be pushed and popped from. This way if FruitGiver eventually restocked it could call `unbecome` to get original behavior again.

The sequential subset of the actor system (behavior definition) was typically functional. Akka as a system used in industry is not that strict as we can see in the example.

### 1.3.2 Active objects

Active objects introduced by Yonezawa [27] takes a different approach to the most actor models. Every object runs in his process with isolated memory.

```
class FruitGiver extends Actor {
  var bananaCount = 2

  def empty = {
    case Give => sender() ! NoMore
  }

  // this behavior is used implicitly
  // when spawning the actor
  def receive = {
    case Give => {
      sender() ! Gave // sending message to the sender of Give
      bananaCount--
      if (bananaCount == 0) {
        become(empty) // changes the behavior to empty
      }
    }
  }
}

val fruitGiver = new FruitGiver
fruitGiver.start
```

Listing 1.3: Scala concurrent objects language example

The state is mutable and changes to the state are done by imperative constructs. No *become* operation exists.

They do define the order of messages as they are received by the consumer. If active object $A_0$ sends messages $m_0$ and $m_1$ in this order, then the same order is going to be abided on the receiver side.

Communication in ABCL/1 (Yonezawa's programming language) defines three types of communication.

**Past** Effectively *send* method as we know it in the classic actor model. Called past because when a consumer receives the message, the sender is already past that moment. The sender did not wait for the response and just went on with his computation.

**Now** After sending a message, the sender waits for the response in a blocking manner. Called now, because when the consumer receives the message, the sender is still on the line for the response.

**Future** The most interesting type of message communication. A message is sent asynchronously and the object does not wait for its completion.

Instead, a special object called the *future* is initialized, which is going to include the potential result of the call. Future objects can be checked to see if they do have the result at any time. Additionally, the `future` is data structure like any other, so it could be sent to another active object.

These constructs could be implemented by classic actors by defining multiple actors to serve the request. For example, the *future* could be implemented by creating an actor that could be set as the customer of the communication. The receiver computes the result it sets that to customer. Anyone can then ask the customer for the "future" value which is set in his state.

```
[object FruitGiver ; object definition
  (state [bananaCount := 2]) ; declaring state of the object
  (script
    (=> [:give sender] ; matches message :give
      (if (= bananaCount 0)
        then [sender <= [:none]] ; sends :none message
        else
          [bananaCount := (- bananaCount 1)]
          [sender <= [:gave]]
        )
    )
  )
]
```

Listing 1.4: ABCL/1 concurrent objects language example

There are no modern implementations for this particular language model, so the Listing 1.4 is written by the definition of the language described in 1986 Yonezawas paper[28].

### 1.3.3 Process-based actor model

The process-based actor model was developed at Erickson labs in 1986 to tackle problems that have arisen when programming distributed telephony applications. In that time developers did not know about the actor model so the development was completely isolated by the theory.

Actors in languages like Erlang were developed as processes running sequential programs (similar to OS processes). They start to run without actually receiving messages, then they stop on the `receive` block and wait for a message to continue. If there is no receive in the process at all, then it ends its execution as any normal function does. Multiple receives can be chained in a function body. Continuity is implemented by recursion of a function representing the process. Erlang’s scheduler works preemptively so processes are swapped out without them explicitly allowing.

The state is passed to the process function as a parameter and state changes happen by recurring with different parameters. This concept is similar to the `become` functionality of classic actors.

```
-module(main).
-export([start/0]).

fruit_giver(0) ->
  receive
    {Pid, give} ->
      Pid ! nomore,
      fruit_giver(0)
  end;
fruit_giver(Num) ->
  receive
    {Pid, give} -> Pid ! gave
  end,
  fruit_giver(Num - 1).

start() ->
  Pid = spawn (fun() -> fruit_giver(2) end),
  Pid ! {self(), give}.
```

Listing 1.5: Erlang language example

### 1.3.4 Communicating Event-Loops

Communicating Event-Loops (CLE) was first defined in the design of the programming language called E language [29]. E-lang stretches the actor model definition to its maximum. *Vat* are definitions of actors that consist of event-loop, which synchronously processes events (or messages), stack, queue, and a heap of objects.

Every object needs to be part of exactly one Vat. Objects, that are in the same Vat, are called *near-references* and those outside are called *eventual-references.* Thanks to event-loop processing events synchronously its safe to access all objects in the same Vat.

E-lang defines two types of communication Immediately (dot operator) or Eventually(arrow operator). Communication via dot operator could be likened to a normal function call but is restricted to only near-reference objects. Immediate pushes the next event to the stack which has read priority. When communicating outside of one's *Vat*, eventual communication must be used. Eventual communication pushes an event to the destination queue. This way every message passing is asynchronous.

As seen in the Listing 1.6 method definition for the E language is communication type agnostic. It does not care about the way it is going to be called. Replying to the messages is done by ordinary return or by throwing an exception, which can be handled by `when..catch` construct.

These near and eventual references restrict ways we can use objects. This restriction is defined as *object capability* and inspired language called Pony to implement a safe, concurrent actor model. Pony guarantees race condition free concurrent programming with no deadlocks using this system [30].

```
def makeFruitGiver() { # Object factory
  var bananaCount := 2
  def gave {} # Local object
  def noMore {}
  def fruitGiver {
    to give() {
      bananaCount := bananaCount - 1
      if(bananaCount == 0) {
        throw noMore # Replies to the caller with error
      } else {
        return gave # Replies to the caller
      }
    }
  }
  return fruitGiver
}

var fruitGiver := makeFruitGiver()
# Returns promise with eventually fulfils to value or error
var p := fruitGiver <- give()
# Returns immediately
var g := fruitGiver.give()
```

Listing 1.6: The E language example

## 1.4 OCaml programming language

OCaml (Objective Caml) is a programming language from ML family with first-class support for objects. ML family languages have a strong static type system with near-complete type inference. That means a lot of type-related errors are caught during compilation, but without the overhead of type declarations when reading the code [31]. The language was firstly introduced in 1987 as a project from Inria in France. Back then called Caml because object-oriented aspects of the language came after the initial release in 1998.

### 1.4.1 Core language

The main construct used in OCaml as a functional language is a function. It is defined by `let` keyword, which is used additionally for both global and local variables too. Besides classical primitive types (string, int, record, etc.), the language defines algebraic data types. They can be likened to enumerations, but more type-safe and with a possibility to hold data. For working with the ADT, pattern matching is defined in the language. An example of that can be seen in the Listing 1.7.

```ocaml
(* Variable declaration *)
let a = 5
let name = "OCaml"

(* Records *)
type language_info = { type : string; created : int }

(* Example of algebraic data type *)
type language =
  | Functional of language_info
  | Imperative of language_info

(* Function declaration *)
let print_language l =
  (* Pattern matching *)
  match l with
  | Functional {name;_} →
    print_endline ("Functional: " ^ name)
  | Imperative {name;_} →
    print_endline ("Imperative: " ^ name)

let () =
  (* Local variable declaration *)
  let name = "OCaml" in
  let created = 1996 in
  print_language (Functional {name; created})
```

Listing 1.7: The core language examples

### 1.4.2 Module system

An interesting aspect of the OCaml language is the rich module system, used as a code separation and implementation hiding concept. Every `.ml` file is automatically a module sharing name with filename only the first letter is upper-case [32, 2.5 Modules and separate compilation]. It groups code to a logical unit. Accessing a code from that module needs to be prefixed by a module name. A module can contain additional modules to define even more fine-grained abstraction. Modules are also called structures.

Some parts of a module can be hidden by using a module signature. Module signature defines how should module look like and anything that is not there can't be used outside the module. There is a file extension for module signatures called `.mli` and it is an idiomatic place to write documentation. Missing module signature file in the OCaml project means everything is public.

Modules can compose together using the module functor [32, 2.3 Functors]. It is a function on the module system level, which takes a module and generates another one. The generated value could be another functor that can be used to emulate multiple parameter functors. Sometimes there is a need for multiple instances of the same module because modules could contain a mutable state. For that purpose, there is a possibility to use functor generators. They are functors that do not take any parameters and every time they are initialized they return a new module independent of those generated before.

OCaml language consists of two parts, the core language, and the module system. First-class modules bridge the gap between these two systems by permitting packing modules that can be used as ordinary values [32, 8.5 First-class modules]. Thanks to that it is possible to hot-swap implementations of modules at run time, implement functions that are generic over a module passed to them, or implement polymorphic behavior as known from OOP languages.

Because there is a huge overlap between modules and objects in OCaml rule of thumb for OCaml codebase is to not use OOP aspects of the language until necessary. Encapsulation is provided by module signatures. Inheritance is replaced by composition using functors and for polymorphism higher-order modules are used.

The Listing 1.8 shows how these concepts translate to actual constructs in the language.

### 1.4.3 Language extensions

Strongly typed languages give safety nets for writing code. They are however limiting in describing some program behavior when types are erased at runtime. This restricts usage of reflection on data types for providing serialization, equality operators, printers etcetera. These functions can be written by hand, but they are repetitive and could be derived from the types itself. An alternative for reflection is code generation.

OCaml defines the PPX language feature, useful for extending arbitrary OCaml syntax in a type-safe way. It uses extension points (nodes), annotations that define what should exactly happen by referencing a specific rewriter[32, 8.13 Extension nodes]. The language preprocessor then calls the rewriter mentioned in the extension point. Rewriters do not work on the text representation

```
(* Module signature *)
module type PLUS = sig
  type t
  (* Infix function *)
  val ( + ) : t → t → t
end

(* Module structure definition *)
module Plus_int = struct
  type t = int
  let ( + ) = Int.add
end

(* Module functor *)
module Sum (P : PLUS) = struct
  include P
  let list = List.fold_left ( + )
end

(* New module from functor *)
module Sum_int = Sum (Plus_int)
```

Listing 1.8: The module system examples

of code itself. Instead, they operate on the AST. They are described as mappings between two abstract syntax threes. Listing 1.9 demonstrates how to generate functions for returning the string representation of the user-defined value.

PPX is also useful for other reasons like extending OCaml syntax to support language constructs that are not accessible in the core language. An example of such syntax could be an implementation of the monadic bind (Haskell's do-notation) [33].

```
  type user = {
    name: string;
    surname: string;
    age: int
  } [@@deriving show]

  (* These function are generated automatically *)
  val pp_user : Format.formatter → user → unit
  val show_user : user → string
```

Listing 1.9: Deriving show for type user

### 1.4.4 Concurrency in OCaml

OCaml does not have multi-core support on thread-level right now so parallel programming is only possible by firing multiple processes manually and communicating between them via pipes/sockets [34]. In other words, there is no thread-based parallelism because the garbage collector is not parallel. It is possible to create OS threads, but they will be under GIL, which prevents more than one thread to run [35].

Concurrency on the other hand is pretty much established in a form of two monadic libraries Lwt [36] and Async [37]. These two libraries are pretty much identical in the ideology and differ only in implementation details.

The Lwt provides `Lwt.t` type, representing a promise of a value, which is going to be filled after computation ends. Combinators provided by the implementation of Lwt help with processing the value. `Lwt_unix` is a sub-library wrapping Unix system calls to return `Lwt.t` values.

OCaml has also an actor based library called distributed, providing erlang based actors. The library is inspired by implementation called Cloud Haskell [38]. A detailed description of differences is available at the end of the Analysis and Design chapter.

Chapter 2

# Analysis and Design

Any actor framework must support the three basic operations: spawning actors, message passing, and state management. The previous chapter presented some commonly used approaches. They are going to be leveraged in this chapter to design the Jude actor framework developed in the thesis.

Apart from elementary constructs, Jude provides more advanced tools to simplify development. It supports the supervision of actor hierarchies and location transparency allowing actors to communicate in a distributed environment.

Finally, using Jude should feel idiomatic to OCaml programmers, leveraging the language constructs.

## 2.1 Functional and non-functional requirements

It is important to define requirements for the library, to define an overview of features it should implement. The rest of the chapter focuses on achieving set requirements. There are many more types of requirements, but they were not considered worth it for a library project.

### 2.1.1 Functional requirements

Next to the actor model requirements, Jude implements additional tools for distributed computing and fail-tolerance. Basic operations were discussed in the actor model chapter.

**Supervison** Supervision allows one to create a hierarchy of interconnected actors with well-defined strategies for handling errors. This forms a parent-child relationship with the parent being responsible for any child actors it has spawned. Furthermore, hierarchies allow one to naturally decompose computation tasks to smaller pieces which are worked by child actors. The results are then sent back to the parent actor which combines them together, pushing the new result up in the hierarchy.

**Location transparency** Location transparency allows actors to communicate across the boundaries of a single process. Each actor is identified with a unique reference. The actor runtime knows how to relay messages between these references regardless if they are local or remote. Messages can, therefore, be sent among actors independently of where they are running.

### 2.1.2 Non-functional requirements

These non-functional requirements are going to be met by the Jude library.

**Fairness of execution** every actor should get the same chance to communicate in the system.

**Reliable message passing** messages should be delivered at most once, resent when lost or corrupted, but without the guarantee that they are delivered.

**Fault-tolerancy** processes should be able to restore the line after losing connection.

**Consistent ordering** message ordering should be expectable for a user of the library.

**Compatible** processes should be able to communicate even when using different versions of the library or when compiled under a different version of OCaml.

**Documentation** documentation of the API should be provided.

**Open-source** library should be available to the public for examination.

**Idiomatic to the user** using build tools programmers, that want to use the library already knows. Make added complexity as little as possible.

## 2.2 Scheduling actors and OS communication

Operating systems define APIs to communicate with the resources they provide. There are both synchronous APIs that block the thread and wait for the resource to be accessible and asynchronous where caller subscribes with a callback to be notified when the resource is ready to be used, which helps decrease the I/O bound of the software.

OCaml has the Unix module, which wraps Unix system calls and threads, as part of the standard library, but its usage for concurrency is not recommended. The reason for that is it works with OS threads which are resource heavy and their parallelism is not used thanks to GIL. Additionally, the Unix module is not fully implemented on Windows operating systems [32, Ch.27 The unix library].

For these reasons another library called Libuv was chosen in place of the Unix module [39]. The Libuv is a C library for asynchronous concurrency. It is used by languages that have GIL (Node.js, Julia) as OCaml does, so it is a good fit. It provides a whole package for communicating with an operating system including file system communication or networking. This library is much more cross-platform than the standard library alternative, with full support for a variety of platforms.

Users of the library subscribe to particular events to get notifications after the event is completed [40]. This helps to decrease I/O bound of the program because when waiting for an event other processes can move forward. Operating systems provide some way of subscribing for input/output. The Libuv uses this subscription mechanism to optimize the asynchronous IO.

The Libuv uses event-loops as the main tool to provide asynchronous programming. Event-loop is a scheduler to which events, with their continuation (callback functions), are registered. After the operating system informs Libuv that resource is ready to be consumed, it fires callback with given information.

One drawback that Libuv has is missing thread-safeness. That means when running in multi-thread environment all communication with Libuv must be on the same thread. Thread safety is important but resource-intensive and Libuv was intended for uses in single-thread environments. For that reason, developers preferred single-thread performance over thread-safety. Jude consists of two threads one with the Libuv and one where the scheduler is running. It is still possible to write thread-safe code using the library [40, Ch. Inter-thread communication] and Jude leverages that.

There already exists a FFI binding to this C library for OCaml which Jude uses for its implementation [41].

## 2.3 Serializing messages

Jude allows actors to communicate independently to where they run. In the case a message is addressed to a remote actor, the message has to sent over a network and therefore it has to be serialized. A lot of different serialization formats exist.

One example of such a format would be JSON. This format got popularized in web development and soon established itself as a most known data exchange format. There are still some drawbacks which should be acknowledged before choosing this standard. First of all, JSON supports only data types used in javascript. Other data types are encoded as a string and need some parse capabilities at both sides. Additionally, human-readable format tends to be more verbose which adds up to cost for parsing and sending data in that format. It would make sense to use JSON for Jude if communication to the outside world directly was desired, but we are going to use this format only internally so OCaml specific libraries and formats are welcomed.

One such example is the Marshall module in the OCaml standard library [32, 25.29 Module Marshal : Marshaling data structures]. Marshal does not work with types and instead encodes the memory model of data used. Almost every type defined in the OCaml is serializable by the Marshal module. The module has some significant drawbacks, like not adding type information of the data or that compatibility is not guaranteed between different compiler versions. Both drawbacks are problematic for specified use-cases, so third-party alternatives were considered over the standard library.

The OCaml community uses s-expressions for a lot of serialization in the ecosystem. They were first invented by Lisp creator John McCartney for encoding both data and code [42]. The data structure for the serialization type is really simple as seen in the Listing 2.1. It encodes data as a tree, where intermediate nodes define its children in the form of a list and leaves with string values. The serialized version of such data structure is represented by a Lisp-like syntax of lists and atoms. S-expressions are unnecessary verbose. They can be used for debugging purposes when to give more human-readable debugging messages.

```
type t = Node of t list | Leave of string
```

Listing 2.1: S-expression data type

Another library which provides OCaml specific format for serialization is Bin_prot. It uses a binary format and so it does define smaller messages. There is no type encoding in the format, but for every type MD5 digest is generated to enable type comparisons. Every message used for communicating with other actors then has this digest as part of the communication. MD5 Digest in binary format is 16 bytes long, but Jude takes only the first 8 bytes to determine if values match. Because MD5 does exhibit avalanche effect [43], a term used for describing that even small changes to the hashed data change the resulting hash drastically [44], it is safe to choose some prefix of the hash. Birthday paradox tells that approximately $1.9 * 10^8$ hashes of length 8 need to be generated to have a chance of 0.1%. That is more than enough for our use case of determining if types match.

Both Sexp and Bin_prot define PPX rewriters to implement serialization and deserialization functions automatically so there is no burden on a user of the library to write them manually. Bin_prot generates PPX for serialization from and to Sexp, for the purpose of debugging.

## 2.4 Arbiter

Actors in the Jude library are not fully self-contained. They run in the Arbiter which manages actor orchestration. It is because of scarce resources available in the operating systems (creating thousands of actors each with their process is resource-heavy).

Arbiter corresponds to one running program of Jude. He manages the scheduling of actors to let every one of them run when possible. It is the runtime in which actors spawn and communicate locally. The role of the arbiter is to make sure communication between actors is fair and that communication with other arbiters is possible.

As seen in the Listing 2.1, communication between actors is provided by the TCP protocol at the transport level. This type of connection is used, because of its reliability and stability. Another reason is that disconnections are monitored by protocol alone so there is no reason for a keep-alive on the application level. There is no plan in using more than a few arbiters and this design choice would let us use thousands of them, so the risk of reaching the upper bound of simultaneous TCP connections is negligible.

For an established TCP connection it is needed to get the address and port of the destination server. For that purpose, all arbiters use discovery protocol based on UDP. Every few seconds arbiters send information about themselves to a known multicast address. Information sent consists of IP address and port to which other arbiters connect using TCP.

There are multiple states of this communication. In the beginning, arbiter $A_0$ is undiscovered and only broadcasts his information to the network. When his communication is received by another actor $A_1$, it establishes communication by sending Syn message on the application layer. $A_0$ then responds with Ready message and waits for Ready message with ack flag set to true from $A_1$. This application protocol is in some ways similar to a three-way handshake implemented in TCP. They are both ways to establish communication but on different layers of TCP/IP protocol. Ack flag in the Ready message is used to make sure we don't call Ready infinitely. This message type is additionally used to communicate metadata about arbiters.

## 2.5 Spawning

Jude allows spawning of the actors only locally without the possibility to spawn them on other arbiters. This happens because of the OCaml type system that erases type information after compilation. No reflection on OCaml values is possible at the runtime, because type information is forgotten [31, Ch. Memory Representation of Values]. One could advocate for the Marshal library which lets serializing functions, but it uses internal information about type representation in memory and is not type-safe. It is impossible to know

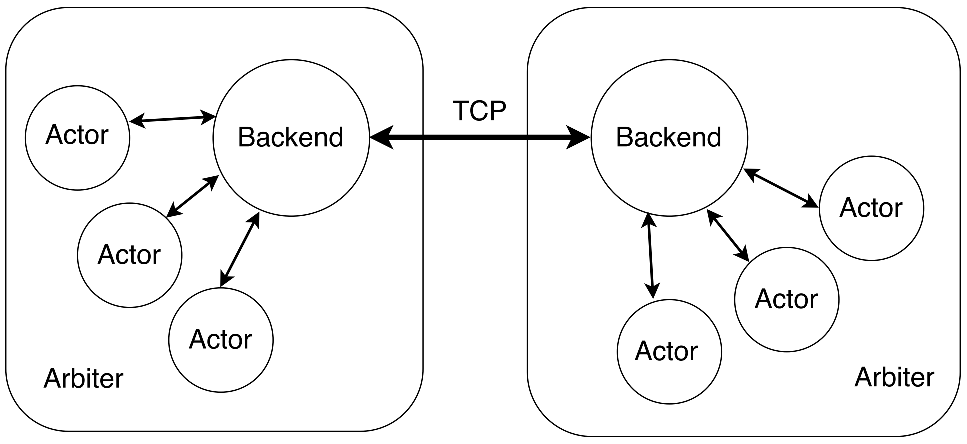

Figure 2.1: The architecture in Jude

how would function look at the other side and Marshal library does not inform the user about the type mismatch. For that reason, Jude does not support the remote spawning of actors.

It is still possible to implement distributed spawning without function serialization by compiling the program and then sending it to another arbiter that then runs that program as another arbiter in the process. Both sides would have tokens defining that actor behavior uniquely. Spawn would then send this behavior token to other arbiters in place of actual function. This architecture is for example used in the Go Circuit library [45], where serializing functions are not possible as well.

The new actor is spawned by calling `Arbiter.spawn` with actor behavior. It creates new universal identification of type `Pid.t` and connects behavior to the PID. Behavior is implemented as a function which takes actor context, metadata used to identify the actor, and returns matcher.

### 2.5.1 PID (Process ID)

PID is a data type consisting of a remote and local part. The remote part defines an IP address and port which is used to identify arbiter which hosts the actor and local part identifies actor in the arbiter. It is used for most of the work with actors such as sending messages, monitoring, linking, or exiting. Every action that is publicly performable on the actor operates on `Pid.t`. Operations private to the actor itself operates on `Actor.t` type. PID type is serializable using `Bin_prot`, unlike `Actor.t`.

The Jude does not have processes, so the PID is more of an actor reference. The reason for the naming is because it is a more compact name.

## 2.6 Changing states

Become function is used to change the state of an actor. Become takes new actor behavior and replaces the old one. Using become actors can change messages they want to receive and their inner state. Similar API for this functionality is used by other actor systems, for example Akka [46].

When an actor state changes, arbiter schedules the actor to make sure that previously ignored messages, that are now receivable can be delivered.

## 2.7 Sending messages

Actors send messages via send function defined on the Arbiter. Send function takes message type in the form of a module, address of the destination actor and a message, which is sent.

There is no certainty that the message would be delivered, so the user of the library should not rely on it. At-most-once delivery is guaranteed, which means that one message could not be delivered multiple times. This guarantee comes from the choice of the protocol used for the main communication. The reason for using at-most-once delivery is that at-least-once delivery is harder to reason about (programmer must check if the message already came) and exactly-once-delivery cannot be implemented at all as described in the "Two Generals Problem" [47], prooven in 1975 [48].

## 2.8 Receiving messages

Consistent messaging is important for actor libraries because developers design their applications based on that. One can reason about message arrival, by adding a timestamp to every message and then sort them. This works until distributed computing is involved. It is impossible to create a global clock, because of the special relativity [13]. Information can be distributed maximally at the speed of light and that is not enough to guarantee fully synchronized clocks. By trying to force a global clock to a distributed system, programs tend to slow down and be less distributed. One way of doing that is to define a master process that dictates the time which creates a bottleneck. Others created logical clocks, where the clock is determined by the total ordering of events in the distributed system [49].

Jude does not guarantee global consistency. Instead, it defines the order of arrival between two actors. Messages sent in a specific order from the source actor to the destination actor arrive in the same order. This is important because a user of the library does not need to buffer messages in case messages are received in the wrong order.

## 2.9 Monitoring actors

Aside from core functionality already described, arbiters define monitoring semantics to enforce the supervision of actors. This is useful for error handling, graceful shutdown, and logging. It provides only limited functionality which can be extended by actors, that wrap that functionality, to provide a higher-level interface. Monitoring actors can be themselves monitored which creates supervision threes. Jude took inspiration mostly from BEAM implementation of monitoring and linking [50], but these concepts can be found in other implementations as well [38, 46].

There are two types of supervision functionality used in Jude called links and monitors. Both are used for notification of exiting processes, but they differ in the strategy they use to do so.

**Linking** creates a bidirectional connection between two actors. It means that if an actor exits all linked actors follow. This leads to a chain of actor destructions. There is a possibility to trap the exit. Trapping denies the exiting of an actor and instead a special message is sent with information about the exited actor. Trapping is useful when cleaning the state is needed. This strategy is useful when some event creates a chain of actors that computes and potentially delivers information to the main actor which created them. If any of them exit during this communication others should exit too and report this incident to the caller.

**Monitoring** does not create a bidirectional connection like linking does. Instead, a connection is created only from the caller to the destination. There is no automatic exiting involved so there is no risk of the monitoring actor accidentally kill itself. When an exception is thrown in the actor body or actor dies, a special message indicating that is sent to the monitor. This message is unique to the monitoring construct.

Both monitoring and linking remote actors are not supported. Because these concepts are mainly used for actors created and Jude does not support the creation of remote actors.

### 2.9.1 Supervisor behavior

Using the monitor feature directly is error-prone so Jude defines an actor called the supervisor. It takes a list of actor recipes and strategy and one by one creates given actors with monitoring. The strategy then defines what will happen if some actor died. There are two strategies for dealing with exiting actors. One is called one-for-one which creates the same actor as the one that died. The second one called all-for-one kills the rest of the actors and then reinitializes all of them.

The supervisor is a normal actor, which means that it can get linked to, and potentially used as a recipe for another supervisor. This linkage of supervisors is called a supervisor tree.

## 2.10 Resolving names

To send a message actor needs the PID address of the destination. To communicate remotely with unknown actors, service that procures PIDs for communication is needed. For this purpose, Jude defines the name registry. Name registry assigns the name of the type `string` to the `Pid.t`. Multiple names could be registered to one `Pid.t`, but only one `Pid.t` is defined per name. This limitation could be bypassed by implementing a custom resolving actor which would use a multimap data structure instead of a normal hashmap as its name storage.

Actors are not always initialized in order which makes sure that a given name exists when needed. An example of that could be recurrent communication, where both sides need the name of the other side. A couple of solutions exist as if it is known that one actor $A_0$ is going to be initialized before the $A_1$, then $A_1$ waits for the name and sends its PID to the first one. This strategy is not possible when actors are not created sequentially in one process.

Resolver behavior solves this problem by introducing a way of subscribing to the concrete name. When the name gets assigned PID, then the actor subscribed to that event is notified about it. This way one side could wait for the name without physically blocking the thread execution.

Name resolving is useful for supervisor behavior too because PIDs are not transferable, and so they become obsolete when actors happen to be restarted. The supervisor does not provide PID addresses of its actors. Instead, he registers their names after spawning. Names are part of the recipe definition.

Chapter 3

# Implementation

This chapter discusses the implementation details of the Jude actor framework. It provides concrete examples of how the framework can be used.

## 3.1 Serialization

Messages, that are sent over the network are serialized to the `Bin_prot` format. This is mostly facilitated by the PPX plugin which generates the corresponding serialization and deserialization code. However, the generated API is not easy to use. The reason is that PPX's focus is on performance rather than user-friendliness.

Because of this, Jude includes helper functions, defined in the binable module, that are used by its implementation to make serialization more manageable. The centerpiece of the binable module is type `Binable.m` which is first-class module type wrapping Bin_prots own `Binable.S` module. This type is polymorphic, so unlike marshal deserialization of the message is type-safe.

The signature of the binable module can be seen in the Listing 3.1. Module serializes to two different formats, buffer defined by Bin_prot and bytes from the standard library. The underlying type of the Buffer is `Array1.t` from the standard library [32, Ch. 25.4 Module Bigarray], which makes the type compatible with a variety of other types like libuv buffer type. It is convenient because serialization can be written right to the network connection, without intermediate representation.

To use these functions you need to first create a module with the type of data you want to serialize. These functions take a first-class module because it is the way to generalize simple functions in the language. OCaml does not have a function or operator overloading so there would be a need to have functions for every message type that is defined in the system.

The bytes version of the interface returns digest for checking if types match. Two different types are defined to make sending encoding messages possible.

```
type 'a Binable.m =
  (module Bin_prot.Binable.S with type t = 'a)

val to_bytes : 'a m → 'a → string * bytes
val to_buffer : 'a m → 'a → Bin_prot.Common.buf

val to_buffer :
  'a Binable.m →
  'a →
  Bin_prot.Common.buf

val from_buffer : 'a Binable.m →
  ?digest:string →
  Bin_prot.Common.buf →
  ('a, string) result
```

Listing 3.1: Binable signature

Example message definition and usage of the binable module can be seen in Listing 3.2.

```
module MyMsg = struct
  type t =
    | Foo of string
    | Bar of int [@@deriving bin_io]
end

let buf =
  Binable.to_buffer (module MyMsg) (Foo "baz")
```

Listing 3.2: Usage of the binable module

### 3.1.1 Matcher

A matcher is a combinator used for determining which message should trigger which path and when the message should be consumed. It is similar to receive function in erlang. Every actor must return a matcher. Jude defines basic matchers to make the development of actors easier. The most useful ones are react and case. With them, one can emulate pattern matching syntax used in language. The reason for using this custom construct in place of the already defined language construct is to make message passing more universal. Default one only matches patterns of the same underlying type.

```
type message = string * bytes

type match_result = Matched | Next

type t = message → match_result
```
Listing 3.3: Matcher data type in Jude

The combinator takes a message (digest, the actual message in bytes) and returns `Matched` to indicate that the matcher consumed the message. The `Next` message is returned if the digest did not match with the type.

By using this combinator one could build complex matching strategies. React matcher is the parent matcher which makes sure that all his children would be processed. Case matcher matches the given bytes message to a specified data type. If the match succeeds function that is associated with a given matcher is called and the message can be pattern matched. One can use multiple case matchers in one react function to simulate matching over multiple message types.

When nothing matches, react returns error indicating that. If there is a need for consuming messages even if the match did not happen, a special matcher called the sink could be used. Sink accepts every message sent to it, so react would not fail. The sink is built on top of the matcher called any. The any matcher lets you define a function that is executed when any consumes a message. It exists so programmers can define behavior when nothing matches.

Sometimes it is desirable to reject every message that is tried. For this purpose, a block matcher is defined. It is used to indicate that no more messages should be sent or when we need to throw out receiving messages until the actor is ready. Jude logs rejected messages in the debug mode. This is the default behavior, but Jude insists that `Actor.beh` returns a matcher. Using an empty react matcher leads to the same functionality.

## 3.2 Backend

The backend module encapsulates communication with other arbiters. It runs both discovery and main connection and provides subscription callbacks to define behavior when a new main connection is established. It is written in a way that provides the possibility to implement another backend and discovery protocol. Arbiter is generalized by its backend, so another communication protocol needs to implement this signature.

The `conn` type defines the connection data structure which is a pair of string for IP address and int for a port number. Start method runs the backend and subscribes callbacks `on_disc` for discovery and `on_conn` for the main connection. Send takes connection with buffer and sends a message to

```
module type B = sig
  type conn = string * int

  val server_conn : conn

  val start :
    on_disc:(conn → unit) →
    on_conn:(Luv.Buffer.t → unit) → unit

  val send : conn → Luv.Buffer.t → unit

end
```

Listing 3.4: Backend signature

that connection. If a connection is not already present, it tries to create one with given parameters. This is just the signature and actual implementation is a functor, which takes the Config module to configure the backend.

Config is filled by using the function called `create_config`. The reason for using a function to create a module is to enable both main and discovery connections to be configured optionally. Unlike functors, functions in OCaml supports default arguments for their parameters.

The data structure representing the internal state can be seen in the Listing 3.5. The server field defines libuv type for the TCP server. This is initialized by the main connection and is used for reliable communication with other arbiters.

```
  type t = {
    server : Luv.TCP.t;
    discovery : Discovery.t;
    clients : (Conn.t, Luv.TCP.t) Hashtbl.t;
    send_ch : (Conn.t * Luv.Buffer.t) Channel.t;
  }
```

Listing 3.5: Backend data type

The discovery field holds the main type used by the Discovery module, used for arbiter finding arbiters on the local network. It does that by repeatedly sending messages about its connection address and port. UDP is an unreliable transport protocol so there is a chance that the message would not be delivered. It is not an issue because eventually it will be delivered and the whole message could be sent in one datagram. Multicast does not support TCP [51, Introduction], because of its client-server nature. Even though other transport layer protocols exist which are more reliable decision was made to use the UDP protocol.

All connections established by the backend module are defined in the `clients` field. It is a hash table that maps connection type (used to define backend and sent over when discovery stage takes place) to the libuv TCP type. When new arbiter is discovered its connection is saved to this field after establishment. When a connection is terminated record associated with the terminated connection is removed from the hash table.

Lastly, backend defines send_ch. Libuv is not a thread-safe library and Jude got two threads. The main one runs the libuv event loop and the second one schedules individual actors to run. That means there is a need for some synchronization when sending messages from actors to a remote arbiter. For these purposes, libuv defines the async handler [52]. Async handler registers up front initialized function which can be run on the event loop and defines send function, which is thread-safe. This callback function can be used to interact with other Luv handlers (like the TCP handler), which are not thread-safe.

Every time actors send a message that needs to be delivered to a remote arbiter, it gets stored to the channel type and the async handler is called. Handler consumes messages one by one and sends them to connections defined in the `clients` field. Messages that should be delivered to an arbiter without records in the `clients` field are ignored. Data pushed to the channel consist of the connection handler and buffer with a message.

Channel data structure is used instead of a queue, because of async handler implementation. It does not guarantee that every call to the handler will run the callback [40, Ch. Inter-thread communication]. Channels try to consume all messages delivered and stop consuming when there are no more messages left. It is safe to call the async handler even when no messages are being sent.

## 3.3 Actor

Multiple versions of the actor implementation were tried over the duration of this thesis. In the beginning, actors were supposed to leverage first-class modules heavily. The actor had its type of message and receive function and these two parts were wrapped to the FCM. The first-class module does have a performance impact, so using it is always weighted by its drawbacks. Additionally, they are verbose and heavy usage of them clutters the code.

Object and classes were tried too, but because they are not that common in the OCaml world and library should be idiomatic they were abandoned. When experimenting with the designs after some time it was apparent that there is no need for grouping the type of message and behavior. For this reason, the final design choice settled on just using a function.

Two major types are defined by the actor. The first one is `Actor.beh`. It describes a function that takes `Actor.t` of the actor and outputs `Matcher.t`. This type is used when spawning actors and corresponds to the basic declare construct.

The second type defined in the actor module is `Actor.t` this type defines the internal structure used by an actor. The type is not accessible outside the module. It stores actors state such as PID, so it can access it inside the behavior function. For accessing PID, function `self_pid`, defined in the actor module, is used.

Every actor owns a mailbox to which raw messages are sent. The mailbox is located in the `Actor.t`. It is implemented as a reference to an immutable queue because the FIFO data structure was needed. Additionally, by using an immutable data structure, one can process messages while getting new ones. The queue is taken from the containers library. Messages in the mailbox are processed using `process\_message` function, which takes another function to determine if the message should be removed from the mailbox. The algorithm used for message receiving was inspired by Erlang's mailbox [50]. When receiving the message, the mailbox tries to match the message to given matchers. If matchers cannot consume the message it goes to local stack until a message is matched. After matching the message everything in the stack is pushed back to the queue preserving original ordering. For that reason, the algorithm needs to have a good time complexity for pushing both to the front and the back of the queue. The algorithm can be futher improved by not recomputing the stack on every match. The only way for the messages in the stack to match is a call to `become` function.

It is implemented as a reference to a immutable single list because there is no need for random access to a specific index. Messages in the mailbox are processed using `process\_message` function, which takes another function to determine which messages were processed. The function takes a list of messages that are currently in the mailbox and returns messages that are not processed. Thanks to the immutability of the list messages can be received, even when processing happens. The returned list is merged to messages that came in the meantime.

Continuation is the actual matcher that is set up in the actor. The representation consists of mutable reference holding `Matcher.t` variable. OCaml supports explicit mutable variables by prefixing fields with `mutable` keyword. Reference wraps given value in the record, which has only one mutable field. It is a syntax sugar for a mutable variable in the OCaml [32, Ch. 1.5 Imperative features]. Actor initialization sets this continuation field to the `block` matcher, so the messages are not lost in the initialization phase.

Both links and monitors of an actor are defined in the `Actor.t`, but they should not be used directly. They exist here as a mere data structure and for them to work one needs to use wrapper functions inside the arbiter. Arbiter versions link actors bidirectionally, unlike this API which links only first one

```
module type ARBITER = sig
  val run : unit → unit
  val spawn : Actor.beh → Pid.t
  val spawn_link : Pid.t → Actor.beh → Pid.t
  val send : Pid.t → 'a Binable.m → 'a → unit
  val link : Pid.t → Pid.t → unit
  val register : string → Pid.t → unit
  val unregister : string → unit
  val get_name : string → Pid.t option
  val resolve_name : string → Pid.t → unit
  val exit : Pid.t → System.Msg_exit.t → unit
  val unmonitor : Pid.t → Pid.t → unit
  val monitor : Pid.t → Pid.t → unit
  val become : Actor.t → Actor.beh → unit
end
module Make : functor (_ : Backend.B) → ARBITER
```

Listing 3.6: Arbiter signature and Make functor defining the arbiter

to the second one.

Lastly, the actor defines flags that could be set up to change internal behavior for the actor. An example of such a flag could be `` `Trap_exit `` flag used to deny an actor its exit capabilities.

Some functions in the actor module operate only on `Actor.t` and not on his `Pid.t`. This ensures that no one would influence the actor state from the outside when it is not desired. It helps isolate actors more.

## 3.4 Arbiter

Arbiter signature defines common operations on actors that need some orchestration (communication of multiple actors). The actual implementation of the arbiter signature is defined as a functor, which takes module with the backend signature.

The `run` function is used as the last statement in the main function. It runs the event-loop and blocks for the duration of the program. When no more events are in the loop, run method returns and effectively exits the program.

The `spawn` function creates a new actor from `Actor.beh` that has type `Actor.t → Matcher.t`. It generates a PID and maps it to the new actor. There is a spawn_link version too, used when one wants to both spawn and link itself to the new actor.

For sending messages between actors, send function is implemented. It checks whenever the message is addressed to a local or remote actor, based on the PID. If it is a remote PID, arbiter wraps the given message to the

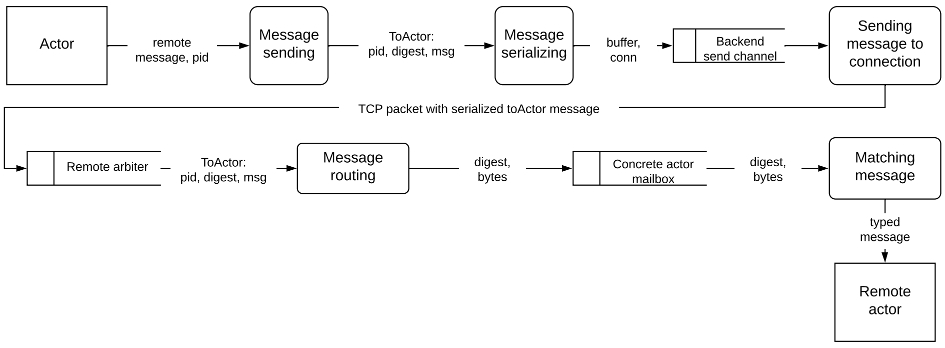

Figure 3.1: Data flow of the remote send

`Deliver_msg` message type which takes PID, a digest of the message, and the message itself in bytes. After sending the message, it is routed by destination arbiter to the right mailbox and schedules assigned actor for the execution. When an actor is scheduled, it tries to consume the message. If message consumption succeeds, it removes the message from the mailbox. As seen in the Listing 3.6, the send function uses the binable type defined in the serialization chapter. Messages that are directed to unknown arbiters will be discarded because Jude does not guarantee delivery and applications using Jude should be ready to handle this kind of situation. Whole remote communication can be examined in the Listing 3.1. Labels on the arrows define the data format used for sending between processes which are represented by squares.

The `register` function is used to assign a name to a PID. When the `register` is called, arbiter sends a notification to others with diff containing the change. So all register data is globally distributed. When a new connection is established registered names are sent as part of `Ready` message.

The `get_name` function tries to get the PID for the given name and throws an exception if it fails. The function should not be used if it is possible that registering the name would not happen before this call. It is prone to race conditions and should be used only in a local setting. One cannot ensure that the arbiter would send the ready message containing the names before this call happens.

If subscription to a remote arbiter is needed, `resolve_name` should be used. It subscribes given PID for this name. So when the PID becomes available, a message of type `Resolution_msg.t` is sent. An actor could match for this message and retrieve the needed PID. In case the name is already resolvable, `resolve_name` sends the message immediately.

Resolving code this way involves a lot of repetition. Every actor that wants to resolve a message needs to wait for the message to be delivered, then change its behavior to the actual one it was designed for. Higher-level API is defined around resolving capabilities to abstract away working with the `Resolution_msg.t` directly. A user of the resolver actor needs to provide a name to register and a function which takes PID and outputs `Actor.beh`.

This actor is located in the `Beh` module. It is a functor that takes the arbiter module. Generalization over the Arbiter module is necessary because it needs to spawn new actors. It works by calling `resolve_name` function and then match for the message. After the name is resolved, it spawns a new actor and exits itself.

Link and monitor functions are low-level tools for creating hierarchies of actors. Links are designed for direct use with the `spawn_link` function. They are suitable for defining actor chains where if one actor in the chain fails, all of them do. If some of them want to free resources it holds it should use `` `Trap_exit `` flag, catch the message free the resources, and exit itself.

It is better to use higher-level APIs for monitoring than primitives defined in the `Arbiter` module. Jude's behavior library defines generic supervisor to manage groups of actors standardly.

Similar to the resolve behavior, the supervisor behavior is a module functor that takes `Arbiter`. It takes a list of recipes, which are used to initialize supervised actors. A recipe is a record consisting of a name, behavior function, and `on_error` field. It initializes actors based on their recipes and registers them by their names. The `on_error` field is used to determine if throwing the exception should result in the actor reinitializing.

The behavior defines type `t`. The type is a record composed of the policy which the supervisor uses, recipes, and running actors. Recipes are stored in an immutable map where the key is the name of the resulting actor and value of the actual recipe that came by the input parameter. Running is defined as a hash table mapping name to the actual PID.

After a supervised actor exits, a notification is sent to the supervisor. This notification is a composite of two things, the reason why an exit happened and which actor was exited. If an error is the cause actor exited, then the supervisor logs the reason why that happened. After that, it checks its strategy to determine what to do next.

For `One_for_one` strategy, supervisor retrieves its recipe and replaces old PID, which he got from the exit message, from the `running` hash table by the new one which he got by spawning the actor using its recipe. Both the mailbox and the state of the actor are lost using this supervisions.

The `All_for_one` strategy is more destructive. When this policy is set, then all actors are reinitialized from recipes. To do that it needs to stop monitoring all supervised actors otherwise monitoring would immediately add another message and the supervisor would exit its actors indefinitely.

Actors can exit in multiple ways. One way of doing that is to explicitly call the `exit` function. It takes PID and a type of exit. Normal type indicates that exiting was not caused by an error state, unlike the error type. Error type takes string describing why the error happened.

In both ways, arbiter looks up all supervisors and sends them a message that the actor is down. Then it looks up all the linked actors and exits them too.

Error handling in the Jude library is managed by monitors too. If an exception is thrown in the actor, the special message `` `Exception `` is sent to all of its monitors. Supervisor then determines, based on the recipe, if it should ignore the exception or reinitialize the actor.

## 3.5 Build system and the package manager

Integration in the language ecosystem is a crucial part of library development. The build system should be idiomatic to the language community and for that reason build system called Dune was chosen.

Dune focuses mainly on the OCaml and is not usable as a general-purpose build system. This means that its functionality and workflow is tailored to OCaml and languages that are part of the ecosystem (ReasonML, Coq). It is tightly coupled to the OCaml tools (testing frameworks, linting, formatting), [53] which is a good thing because it decreases the entrance level for new developers.

As a format of choice dune uses s-expressions for its definition. Good support of this format and ease of processing (this format is capable of representing both code and data) in the OCaml language is the reason why they picked it. Example of configuration can be found in Listing 3.7 or in Listing 3.8.

```
(library
(name jude)
(public_name jude)
(flags
  (:standard
    -safe-string
    -w
    +A-48
    -warn-error
    +A-3-44))
(libraries luv bin_prot uuidm logs)
(preprocess
  (pps
    ppx_bin_prot ppx_deriving.eq)))
```

Listing 3.7: The dune file

```
(lang dune 2.0)
(name jude)
(generate_opam_files true)
```

Listing 3.8: The dune-project file

The project file should be in the root of the project and is used by dune to determine the source tree for other packages. It names the dune version used for libraries as well as the name of the project. Every package in the project has its `dune` file. This file defines the build configuration that dune uses for compiling that package. Jude has three packages in the project. The main library, testing library, and example library. Dependencies packages need are specified under the library key. If a package uses PPX, then it needs to be declared under the preprocess key.

Dependencies needed by libraries are installed by the package manager called OPAM. It is used to install both language versions in the OCaml ecosystem as well to install libraries project needs. Multiple switches, independent environments, can be defined to isolate unrelated projects and to prevent collisions of the library versions.

When everything is set up, calling `dune build` from the command line builds all the packages defined in the project you are currently in. Additional commands that are automatically available exist for installing binaries, testing, and executable running purposes.

## 3.6 Testing

Jude testing happens in the `lib\_test` package using a library called Alcotest. This library generates a compilable program that could be run to determine if tests pass or not. Dune provides integration for the testing using the test package dune file. By specifying that package is used to test the code, dune generates a new command called `dune test` which compiles the code and runs it from that directory.

Alcotest provides a simple comparison and checking primitives. It is not a full-featured testing framework and expects users to wrap those primitives to make higher-level API. Type implements `TESTABLE` signature so the Alcotest can use its checks on them. Example in the Listing 3.9 shows the test that makes sure `on_name` callback is called when register assigns PID to the name. The `check` function seen in the example is used to compare data equality of an `TESTABLE` module.

```
let test_on_name_callback () =
  let reg = Registry.create () in
  let succ = ref false in
  let pid = Pid.create ("127.0.0.1", 7000) in
  Registry.on_name reg "test" (fun _ → succ := true);
  Registry.register reg ~local:true "test" @@ pid;
  Alcotest.(check bool) "should be succ" !succ true
```

Listing 3.9: Example test

## 3.7 Documentation

Idiomatic OCaml library writes their documentation to `.mli` files as special documenting comments. When those comments are made library called Odoc takes them and generates web documentation. It provides good linking between comments, so other types and functions can be referred from the comment. It creates links in the HTML document. The Odoc is a low-level generator, which is wrapped by a build system to provide higher-level API. Dune does that by using command `dune build @doc`. Documentation generated for Jude is accessible on the enclosed CD.

## 3.8 Logging

Without logging, it is hard to determine what went wrong. OCaml ecosystem provides logging infrastructure in the form of a library called `Logs` [54]. Logging function lets format the message using `Formatter` module from the standard library. Functions for logging messages do not take the formatted string directly instead they take lambda function which returns the formatted string to decrease the performance impact of message creation. The lambda function is called only if the level of the log record is lower or equal to the configured value.

Every library should use its log source, which this then used for configuring the logger. Jude logging location and debug level can be configured from the main function of the program.

## 3.9 Project setup

The section introduces the reader to Jude's integration with developer tools. Github is used for hosting the code. When push is made to a branch Travis-CI runs scripts provided in the `scripts` folder. Firstly, Travis compiles the Jude library in three different OCaml versions. Then it runs the test and if they succeed, it generates a new version of the documentation. For the master branch, the Odoc documentation is hosted at Github pages.

## 3.10 Case study – The Heat Aggregator

Classic concurrency and Akka actor versions of this example were already introduced. Jude's version in the Listing 3.10 is more verbose than the Akka version, but that is because of OCaml being more explicit language. Some implementation details (as arbiter and backend initializations) were omitted to create a more concise snippet. The full working example can be found in the enclosed CD.

Unlike the Akka version, Jude uses a function to create an actor and no other methods are invoked. Akka defines user-overridable life-cycle hooks which are used to influence the startup and teardown of an actor. In Jude, the start-up hook is just the body of a function. The actual actor is represented by the continuation stored in the matcher. The tear-down is implemented by

```
module Heat_request = struct
  type t = Add_record of float | Compute_avg of Pid.t [@@deriving bin_io]
end

module Heat_reply = struct
  type t = float option [@@deriving bin_io]
end

let sum_arr = Array.fold_left Float.add 0.

(* The actor declaration *)
let heat_aggregator ~size _ctx =
  let window = Array.make size 0. in
  let actual = ref 0 in
  (* Using imperative constructs
  because it is easier in this scenario *)
  let full = ref false in
  (* Local open to get combinators in the scope *)
  let open Matcher in
  case (* Using case matcher without react because of one case*)
    (module Heat_request)
    (function
      | Add_record f →
          actual := (!actual + 1) mod size;
          if size - 1 == !actual then full := true;
          window.(!actual mod size) <- f
      | Compute_avg pid →
          let msg =
            if !full then
              Some (sum_arr window /. Int.to_float size)
            else None
          in
          Arbiter.send pid (module Heat_reply) msg)
```

Listing 3.10: The heat aggregator in the Jude library

trapping the exit and handling it when the exit message is received. Encapsulation is provided by local variables that are referenced in the continuation.

Chapter 4

# Comparison to Other Actor Libraries

The chapter compares three modern actor systems, which were the inspiration for Jude design. Actor systems that are discussed in this chapter are Erlang, Akka and distributed [50, 26, 55]. The former two are picked for their long-lasting production usage. The third actor library was chosen because it is written in OCaml and represents an alternative to Jude in the same ecosystem.

## 4.1 Erlang

Erlang itself is an actor language, so there is first-class support for the actor model. It was the first industrial-strength actor model implemented to tackle distribution problems related to telephony switching. It uses the Erlang OTP framework to provide higher-level API to its primitives for monitoring and name resolving [50].

Jude took inspiration in the design of these higher-level concepts. It could not effectively use Erlang's process-based approach to actor spawning, because there is no multi-core support in the language and erlang leverage's its runtime system to guarantee process memory isolation.

Jude used similar concepts and signatures to those in erlang. Both define resolving, monitoring, and linking functions. Even if they look similar internal implementation differs, because the runtime system of the systems is not compatible. The only difference in the API is that OCaml explicitly passes actor identification to the functions, whereas erlang determines which actor should be affected by the process from which was the function called. It can be done, but passing identification is more flexible for implementing certain patterns. Additionally, it would mean a more mutable state in the arbiter (to change actually running actor).

Erlang has a rich ecosystem and a standard library called OTP. The ecosystem defines supervisor behavior, which features generic supervision for processes in the Erlang language [56]. Jude used its recipe system to implement a simpler version of the supervisor. There are some strategies missing and only the shutdown of an actor is currently supported.

Globally available register for assigning names to the processes is provided by the OTP. API differs more in this feature. Erlang lets function decide which PID is the right one in the case of a name clash. Jude ignores clashes and just replaces the PID without checking. Additionally, Jude provides `resolve_name` function and resolver behavior to make distributed name resolving more accessible. Erlang does not provide this kind of functionality out of the box.

## 4.2 Akka

Akka is an actor framework created for JVM languages Scala and Java. It is one of the most widespread actor libraries. Akka provides a hybrid system for actors, where threads used by scheduler are extended when all threads block their computation, in a non-preemptive way. If an actor ends its computation explicitly, the underlying thread is reused by another actor which is ready to be run. Comparison is using the newer actor typed API [57].

Jude's actor running is similar to Akka's cooperative switching. Both register continuation to match when new messages come. Akka defines the actor as a class extending the `Actor` trait. It provides two functions to handle communication with actors. The `onMessage` is used for normal communication and the `onSignal` is used for erroneous states. Jude uses one function and message receiving way for both signals and messages, so when the user knows how to handle messages he automatically knows how to handle signals.

Actor's state changing differs in Akka significantly in comparison to the Jude library. Actors in the Akka return next behavior (which is alternative to matchers in the Jude), every run needs to define what kind of behavior changes should happen for the next message. Jude lets you use the same behavior until changed explicitly by calling the `become` from the body of a function.

The supervisor strategy is not baked into the actor definition when using Jude. In Akka, the parent actor defines a supervisor strategy for their children. Parent reference is passed when creating a new actor (similar to `spawn_link`) and that actor is automatically responsible for handling the supervision, unlike in the Jude where the supervisor is just a function (same like any other actor), which has only one job and that is to supervise its children. It is still possible to implement failure handling the way Akka does by leveraging actor links.

## 4.3 Distributed

Distributed is the only relatively active actor framework for OCaml. Other frameworks are in early alpha, or they are not supported anymore. Distributed is inspired by Cloud Haskell implementation a lot and tries to bring typed actors to the OCaml language.

There are some differences between Jude and distributed. One of them is that distributed uses Lwt as its backend and builds on top of it. That is good for ordinary programmers that have experience with the Lwt concurrency because it works the same way. It is possible to add this compatibility to Jude by promisyfing callback API and using Lwt as the promise type.

Another difference lies in the way distributed handles messages. It defines messages that actors can send and receive in a way that cannot be changed. One can change the way messages are handled, but not the type of messages that are handled. One way to solve that is to define all acceptable messages when initializing the actor, even when not every message is going to be used immediately. The reason is that it takes out the need to send and receive function being polymorphic over their parameters. Jude does not mind being more verbose for the sake of more dynamic behavior.

Distributed uses Marshal to serialize messages. The rebuttal was made why Jude does not use Marshal in the Serialization chapter.

Matchers are similar in Jude and distributed. Distributed uses a list of matchers where every case is a matcher, unlike Jude where matcher is type of function. Functions are easier to combine using combinatoric API.

## 4.4 Actor model flavor of Jude

As seen in chapter Flavours of the actor system, there are lots of different actor model designs. Jude took inspiration form all of them, but still, its design lean mostly to the classic actor design. State changes are made by using become with new behavior. The sequential subset is functional. Both mutable state and immutable state are supported by default, but the idiomatic way to change the state is to use become and pass a new state as a parameter.

Active objects as used by Yonezawa guarantee message sending order, which is the main implementation detail that differentiates Jude from AB-CL/1. Usage is different too because only past messages are available in Jude by default. One can build other types of messages by using past message only [28]. Jude did not find this particular feature that useful to justify adding it.

Erlang and its process-based model inspired the looks of the Jude API. It uses similar constructs to provide supervision and name resolving. Additionally, actors in the Erlang are just a function, which holds true to Jude's implementation as well. Differences lie mainly in the backend of both implementations. Erlang isolates every actor to its own lightweight process where no memory is shared. OCaml does not have this kind of capabilities built-in so it would be hard to implement that. Sharing memory in Jude directly is undefined behavior as there is no way to detect that. Jude has only one process which switches actors when they choose to end their computation.

Jude's backend is built on an event-driven library called Libuv. Events are a significant part of its design. The state and behavior of the actor are wrapped in the callback. De facto all actors in Jude work on an event basis. This is where the similarities with Communicating Event-Loops ends. Jude does not provide two types of communication as the E language does. Their internals differs a lot, because E language defines event-loop per actor, whereas Jude has one internal event-loop which processes all events that are sent to it. Vats are comparable to arbiters, but communication to actors does not differ based on their location, as in Vats. This makes Jude more consistent in terms of sending messages.

Chapter 5

# Future Work

The presented Jude actor framework could be extended in many directions. For example, common protocols that are used outside of our actor system are in need, like HTTP or Websocket protocols. These are going to be implemented from scratch or maybe after promisyfing Jude, existent libraries build on top of Async and Lwt libraries can be leveraged. Jude should integrate operating system calls and hide low-level code that is used to communicate with it, behind actors.

Its future enhancements are also going to be influenced by new features implemented in the language. For example, multi-core support is going to be introduced to the language soon [58], which could have shaken the present implementation of the Jude actor model. It is going to introduce algebraic effects [59], where continuation could be implemented not by returning matcher from the function, but by yielding the computation to the scheduler at any given time. By using domains [60], OCaml processes are going to support parallel hardware.

In a broader time span, there is a possibility that modular implicits are going to be implemented [61]. This would make some construct in the library less verbose, thanks to the introduction of ad-hoc polymorphism. It is similar to constructs such as type classes in Haskell [62], or traits in the Rust language [63]. For example `Arbiter.send` function would not need to specify a module, from which serialization functions are taken as seen in the Listing 5.1. Instead, compiler would infer module to use implicitly. Same goes for matchers omitting first-class module that needs to be passed for the deserialization to work.

```
(* Defined in the library *)
module type Binable = sig
  type t [@@deriving bin_io]
end

(* Defined by the user *)
implicit module Greet = struct
  open Bin_prot.Std

  type t = Adioso of string [@@deriving bin_io]
end

(* Sending message of type Greet*)
let () =
  (* Before implicits *)
  Arbiter.send (module Greet) (Adioso "ola!")
  (* After implicits *)
  Arbiter.send (Adioso "ola!")
```

Listing 5.1: Arbiter send using modular implicits

# Conclusion

The goal of this thesis was to design and implement the actor model in OCaml. The result is Jude, an OCaml framework for actor-based concurrency. Next to the core operations such as spawning, messaging, and state-changing, it also supports actor supervision and location transparency. This allows one to develop a fully distributed concurrent applications in Jude using the actor model of computation.

The thesis briefly introduced general concurrency concepts and classic concurrency. Next, the actor model was described with examples of flavors existing in the field. Background needed to understand the thesis was wrapped up by familiarizing readers with the OCaml language. When the background needed for the work was established, the actor model was analyzed to define features. The library was then designed to meet those criteria. The design was then converted to the implementation with descriptions of problems that were needed to overcome. The resulting actor library was at the end compared to its alternatives.

This thesis gives a taste of designing and implementing libraries for a functional programming language. One should additionally take a look at other concurrency models and libraries (like Lwt or Async) used by the OCaml and learn more about this interesting language. The same applies to the actor model. For almost fifty years it helped to shape the concurrent world. Remember that labor becomes easier when the right tool (or abstraction) is used for the job.

# Bibliography

1. SUTTER, Herb. The Free Lunch Is over: A Fundamental Turn toward Concurrency in Software. *Dr. Dobb's journal.* 2005, vol. 30, no. 3, pp. 202–210.
2. *Successful Companies Use Erlang and Elixir* [online]. 2018 [visited on 2020-06-02]. Available from: `https://web.archive.org/web/20200221090407/https://codesync.global/media/successful-companies-using-elixir-and-erlang/`.
3. *Who Is Using Orleans? | Microsoft Orleans Documentation* [online]. 2020 [visited on 2020-06-02]. Available from: `https://web.archive.org/web/20200122150922/https://dotnet.github.io/orleans/Community/Who-Is-Using-Orleans.html`.
4. ARMSTRONG, Joe. Erlang - A Survey of the Language and Its Industrial Applications, pp. 8.
5. BEN-ARI, M. *Principles of Concurrent Programming.* 2nd ed. Addison-Wesley, 2006. Prentice Hall International Series in Computer Science. ISBN 978-0-321-31283-9. Available also from: `https://books.google.cz/books?id=oP-2hpMEdb8C`.
6. DIJKSTRA, E. W. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM* [online]. 1965, vol. 8, no. 9, pp. 569 [visited on 2020-06-02]. ISSN 0001-0782. Available from DOI: `10.1145/365559.365617`.
7. JONES, Simon Peyton. Beautiful Concurrency. *Beautiful Code: Leading Programmers Explain How They Think.* 2007, pp. 385–406.
8. BUSTARD, David W. *Concepts of Concurrent Programming.* CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1990. Available also from: `https://resources.sei.cmu.edu/asset_files/CurriculumModule/1990_007_001_15815.pdf`.

9. RAYNAL, M. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32027-9. Available also from: `https://books.google.cz/books?id=0a1AAAAAQBAJ`.

10. TANENBAUM, Andrew S.; BOS, Herbert. *Modern Operating Systems*. Pearson, 2015. ISBN 978-0-13-359162-0. Available from Google Books: `9gqnngEACAAJ`.

11. SOTTILE, M.J.; MATTSON, T.G.; RASMUSSEN, C.E. *Introduction to Concurrency in Programming Languages*. CRC Press, 2009. Chapman & Hall/CRC Computational Science. ISBN 978-1-4200-7214-3. Available also from: `https://books.google.cz/books?id=J5-ckoCgc3IC`.

12. HOARE, C A R. Communicating Sequential Processes, pp. 260.

13. AGHA, Gul A. *ACTORS - a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1990. MIT Press Series in Artificial Intelligence. ISBN 978-0-262-01092-4.

14. HEWITT, C; BISHOP, P; STEIGER, R. A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI3. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. 1973, pp. 235–245.

15. LIEBERMAN, Henry. *A Preview of Act 1*. 1981. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB.

16. HEWITT, Carl; MEIJER, Erik; SZYPERSKI, Clemens. The Actor Model (Everything You Wanted to Know, but Were Afraid to Ask). 2012. Available also from: `https://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask`.

17. HOARE, C. A. R. Communicating Sequential Processes. *Communications of the ACM* [online]. 1978, vol. 21, no. 8, pp. 666–677 [visited on 2020-04-25]. ISSN 0001-0782. Available from DOI: `10.1145/359576.359585`.

18. GERRAND, Andrew. Share Memory by Communicating. *The Go Blog*. 2010. Available also from: `https://blog.golang.org/codelab-share`.

19. *Distributed Haskell* [online] [visited on 2020-05-06]. Available from: `https://github.com/haskell-distributed`.

20. *Prototypes vs Classes Was: Re: Sun's HotSpot* [online] [visited on 2020-06-04]. Available from: `http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html`.

21. KAY, Alan C. The Early History of Smalltalk. In: *History of Programming Languages—II*. 1996, pp. 511–598.

22. HALLIDAY, Leigh. *Ruby Metaprogramming - Method Missing* [online] [visited on 2020-04-28]. Available from: `https://www.leighhalliday.com/ruby-metaprogramming-method-missing`.

23. THE PHP DOCUMENTATION GROUP. *PHP - Manual* [online]. 2020 [visited on 2020-05-29]. Available from: `https://www.php.net/manual/en/language.oop5.overloading.php#object.call`.

24. DE KOSTER, Joeri; VAN CUTSEM, Tom; DE MEUTER, Wolfgang. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* [online]. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 31–40 [visited on 2020-04-28]. AGERE 2016. ISBN 978-1-4503-4639-9. Available from DOI: `10.1145/3001886.3001890`.

25. AGHA, Gul. Concurrent Object-Oriented Programming. *Communications of the ACM* [online]. 1990, vol. 33, no. 9, pp. 125–141 [visited on 2020-04-28]. ISSN 0001-0782. Available from DOI: `10.1145/83880.84528`.

26. BONÉR, Jonas. *Introducing Akka - Simpler Scalability, Fault-Tolerance, Concurrency & Remoting through Actors* [online]. 2010 [visited on 2020-04-29]. Available from: `http://jonasboner.com/introducing-akka/`.

27. YONEZAWA, Akinori; BRIOT, Jean-Pierre; SHIBAYAMA, Etsuya. Object-Oriented Concurrent Programming in ABCL/1. In: AGHA, Gul; IGARASHI, Atsushi; KOBAYASHI, Naoki; MASUHARA, Hidehiko; MATSUOKA, Satoshi; SHIBAYAMA, Etsuya; TAURA, Kenjiro (eds.). *Concurrent Objects and Beyond: Papers Dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday* [online]. Berlin, Heidelberg: Springer, 2014, pp. 18–43 [visited on 2020-04-29]. Lecture Notes in Computer Science. ISBN 978-3-662-44471-9. Available from DOI: `10.1007/978-3-662-44471-9_2`.

28. YONEZAWA, Akinori; BRIOT, Jean-Pierre; SHIBAYAMA, Etsuya. Object-Oriented Concurrent Programming in ABCL/1. In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* [online]. Portland, Oregon, USA: Association for Computing Machinery, 1986, pp. 258–268 [visited on 2020-05-26]. OOPSLA '86. ISBN 978-0-89791-204-4. Available from DOI: `10.1145/28697.28722`.

29. MILLER, M. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control 2006. *Johns Hopkins: Baltimore, MD.* 2006, pp. 302.

30. CLEBSCH, Sylvan; DROSSOPOULOU, Sophia; BLESSING, Sebastian; MCNEIL, Andy. Deny Capabilities for Safe, Fast Actors. In: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 2015, pp. 1–12.

31. MINSKY, Yaron; MADHAVAPEDDY, Anil; HICKEY, Jason. *Real World OCaml: Functional Programming for the Masses*. " O'Reilly Media, Inc.", 2013.

32. LEROY, Xavier; DOLIGEZ, Damien; FRISCH, Alain; GARRIGUE, Jacques; RÉMY, Didier; VOUILLON, Jérôme. *The OCaml System Release 4.09: Documentation and User's Manual*. 2019. Available also from: `https://hal.inria.fr/hal-00930213`. Intern report. Inria.

33. *Janestreet/Ppx_let* [online]. 2020 [visited on 2020-05-29]. Available from: `https://github.com/janestreet/ppx_let`.

34. *Concurrency, Parallelism, and Distributed Systems* [online] [visited on 2020-05-01]. Available from: `https://ocamlverse.github.io/content/parallelism.html`.

35. DOLAN, Stephen; WHITE, Leo; MADHAVAPEDDY, Anil. Multicore Ocaml. In: *OCaml Workshop*. 2014, vol. 2.

36. VOUILLON, Jérôme. Lwt: A Cooperative Thread Library. In: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML* [online]. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 3–12 [visited on 2020-04-26]. ML '08. ISBN 978-1-60558-062-3. Available from DOI: `10.1145/1411304.1411307`.

37. *Async* [online] [visited on 2020-05-01]. Available from: `https://opensource.janestreet.com/async/`.

38. EPSTEIN, Jeff; BLACK, Andrew P.; JONES, Simon L. Peyton. Towards Haskell in the Cloud. In: CLAESSEN, Koen (ed.). *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. ACM, 2011, pp. 118–129. Available from DOI: `10.1145/2034675.2034690`.

39. *Libuv/Libuv* [online]. 2020 [visited on 2020-05-13]. Available from: `https://github.com/libuv/libuv`.

40. MARATHE, Nikhil. *An Introduction to Libuv*. 2015. Available also from: `http://nikhilm.github.io/uvbook`.

41. BACHIN, Anton. *Aantron/Luv* [online]. 2020 [visited on 2020-05-30]. Available from: `https://github.com/aantron/luv`.

42. MCCARTHY, John. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*. 1960, vol. 3, no. 4, pp. 184–195.

43. RIVEST, Ronald; DUSSE, S. The MD5 Message-Digest Algorithm. 1992.

44. FEISTEL, Horst. Cryptography and Computer Privacy. *Scientific American*. 1973, vol. 228, no. 5, pp. 15–23. ISSN 00368733, 19467087. ISSN 00368733, 19467087. Available from JSTOR: `24923044`.

45. *Gocircuit/Circuit* [online]. 2020 [visited on 2020-05-13]. Available from: `https://github.com/gocircuit/circuit`.

46. HALLER, Philipp. Isolated Actors for Race-Free Concurrent Programming. 2010. Available from DOI: `10.5075/epfl-thesis-4874`.

47. GRAY, Jim. Notes on Data Base Operating Systems. In: *Operating Systems, an Advanced Course*. Berlin, Heidelberg: Springer-Verlag, 1978, pp. 393–481. ISBN 3-540-08755-9.

48. AKKOYUNLU, Eralp A; EKANADHAM, Kattamuri; HUBER, Richard V. Some Constraints and Tradeoffs in the Design of Network Communications. In: *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*. 1975, pp. 67–74.

49. LAMPORT, Leslie. Time, Clocks, and the Ordering of Events in a Distributed System. 1978, vol. 21, no. 7, pp. 8.

50. ARMSTRONG, Joe. *Making Reliable Distributed Systems in the Presence of Sodware Errors*. 2003. The Royal Institute of Technology Stockholm, Sweden.

51. DEERING, Steve E. Host Extensions for IP Multicasting [RFC 1112]. 1989, no. 1112. Available from DOI: `10.17487/RFC1112`.

52. *Uv_async_t — Async Handle — Libuv Documentation* [online] [visited on 2020-05-30]. Available from: `http://docs.libuv.org/en/stable/async.html`.

53. DIMINO, Jérémie. *Dune Documentation*. Available also from: `https://dune.readthedocs.io/en/stable/overview.html`.

54. *Logs / Erratique* [online] [visited on 2020-05-30]. Available from: `https://erratique.ch/software/logs`.

55. S.T. *Essdotteedot/Distributed* [online]. 2020 [visited on 2020-05-23]. Available from: `https://github.com/essdotteedot/distributed`.

56. *Erlang – Supervisor* [online] [visited on 2020-05-23]. Available from: `https://erlang.org/doc/man/supervisor.html`.

57. *Introduction to Actors • Akka Documentation* [online] [visited on 2020-05-31]. Available from: `https://doc.akka.io/docs/akka/current/typed/actors.html`.

58. SIVARAMAKRISHNAN, K. C.; DOLAN, Stephen; WHITE, Leo; JAFFER, Sadiq; KELLY, Tom; SAHOO, Anmol; PARIMALA, Sudha; DHIMAN, Atul; MADHAVAPEDDY, Anil. Retrofitting Parallelism onto OCaml [online]. 2020 [visited on 2020-05-27]. Available from arXiv: `2004.11663 [cs]`.

59. DOLAN, Stephen; ELIOPOULOS, Spiros; HILLERSTRÖM, Daniel; MADHAVAPEDDY, Anil; SIVARAMAKRISHNAN, K. C.; WHITE, Leo. Concurrent System Programming with Effect Handlers. In: WANG, Meng; OWENS, Scott (eds.). *Trends in Functional Programming*. Cham: Springer International Publishing, 2018, pp. 98–117. ISBN 978-3-319-89719-6.

60. *Ocaml-Multicore/Domainslib* [online] [visited on 2020-05-31]. Available from: `https://github.com/ocaml-multicore/domainslib`.

61. WHITE, Leo; BOUR, Frédéric; YALLOP, Jeremy. Modular Implicits. *Electronic Proceedings in Theoretical Computer Science* [online]. 2015, vol. 198, pp. 22–63 [visited on 2020-05-27]. ISSN 2075-2180. Available from DOI: `10.4204/EPTCS.198.2`.

62. WADLER, Philip; BLOTT, Stephen. How to Make Ad-Hoc Polymorphism Less Ad Hoc. *[No source information available]*. 1997. Available from DOI: `10.1145/75277.75283`.

63. KLABNIK, Steven; NICHOLS, Carol. *Traits: Defining Shared Behavior - The Rust Programming Language* [online] [visited on 2020-05-31]. Available from: `https://doc.rust-lang.org/book/ch10-02-traits.html`.

Appendix **A**

# Acronyms

**API** Application Programming Interface

**AST** Abstract Syntax Three

**CD** Compact Drive

**CLE** Communicating Event-Loops

**CSP** Communicating Sequential Processes

**FCM** First Class Modules

**FFI** Foreign Function Interface

**FP** Functional Programming

**GIL** Global Interpreter Lock

**HOM** Higher Order Modules

**IO** Input/Output

**JSON** Javascript Object Notation

**JVM** Java Virtual Machine

**OPAM** OCaml PAckage Manager

**OS** Operating System

**PID** Process ID

**SRP** Single Responsibility Principle

**TCP** Transmission Control Protocol

**UDP** User Datagram Protocol

| | |
|---|---|
| **DNS** | Domain Name Server |
| **IP** | Internet Protocol |

# Appendix B

# Contents of enclosed CD

```
README.md ........................ the file with CD contents description
src .................................. the directory of source codes
  jude ..................................... implementation sources
    example ........................... implementation examples
    lib ....................................... library source codes
    lib_test ...................................... library tests
    scripts ................................ scripts used for CICD
    README.md ........................... README of the library
  thesis ............ the directory of LaTeX source codes of the thesis
text ....................................... the thesis text directory
  BP_Vardanjan_Narek_2020.pdf ....... the thesis text in PDF format
```