D actor framework: actor-d
Distributed System Project Final Report

Alex Speranza
alex.speranza@studio.unibo.it

December 2021

The library `actor-d` aims to be a basic actor framework based on message passing for the `D` programming language, as there are no other up to date packages with the same purpose.

The project will focus on providing a simple actor abstraction capable of performing the actions of `send` (to both local and remote actors, with the focus mainly on the latter), `receive`, `create` and `become`, while also decoupling actors from threads and optimizing the required resources.

1 Introduction

1.1 D programming language

> "*D* is a general-purpose systems programming language with a *C*-like syntax that compiles to native code." [10]

This language is born as a *C++* redesign, but the development drifted into a new language[19].

> "The design goals of the language attempted to combine the performance and safety of compiled languages with the expressive power of modern dynamic languages." [19]

D is a medium-level (of abstraction) language as it mixes aspects of both low-level and high-level worlds (non-exhaustive list):

- Low level:
  1. pointers and pointer arithmetic;
  2. low-level data aggregation (`structs` & `unions`);
  3. imperative-only style is allowed;
  4. memory allocation and deallocation can be manually managed;

5. inline assembler[9];
6. compilation produces native code.

The *D* compilation includes a linking phase, which allows to use external library functions[11], but not all of them work out-of-the-box and need port.

- High level:
  1. object oriented components (`class`es, `interface`s, `Object`);
  2. automatic garbage collection;
  3. code is divided in modules and packages (no more header files);
  4. reflections (with `traits`);
  5. `mixin`s;
  6. error handling with try/catch blocks;
  7. functional programming.

D supports 5 main programming paradigms[18]:

1. **imperative**: *C*-like programming, with the addition of a `foreach` looping construct and nested functions;
2. **object-oriented**: Java-like inheritance, reflections with `trait`s and `TypeInfo`s;
3. **functional**: *D* allows the use of function literals (lambdas) and higher order functions;
4. **concurrent**: the standard library of *D* contains functions to start parallel and concurrent computations and tasks, while the type system ensure that data sharing can be detected;
5. **metaprogramming**: `template`s, `tuple`s, compile-time function execution (CTFE) and string `mixin`s allow the generation of *D* code at compile time;

1.2 DUB - D package manager

DUB is the official package manager for the
textittexttttD language and it is distributed along the major *D* compilers (*DMD* and *LDC*). It allows to deeply configure a *D* project aspects, among all:

- project metadata: name, description, author, license;[8]
- dependencies (both published on a registry and local);[7]
- platform-specific configurations;[7]
- pre/post commands;[7]
- source paths/files;[7]

- compiler flags.[7]

The main commands are:

- `build`: compiles the code and generate an executable;
- `run`: builds the project and start the executable;
- `test`: build the projects and the unit tests and run them.

Published packages can be searched on `https://code.dlang.org/`.

1.3 Actor model

> "The actor model is a mathematical theory that threats 'Actors' as the universal primitives of digital computation." [13]

In the most widely known and accepted formalism, actors are active agents whose control flow and data flow are tied together and whose behavior can be defined in terms of sending massages to actors.[4]

Messages are the unit of communication, and they trigger a defined behavior on the receiving actor. The basic actions an actor can perform, while receiving messages, are[13]:

- send messages to actors he knows the address of;
- create new actors;
- define how to handle the next message it receives.

Actors can be implemented as objects (with the same meaning in 'object-oriented programming') with an abstraction of a mailbox and an abstraction of a control flow thread (i.e. actors may not have their own mailbox or thread, but to them it appears so).

The key principle of the actor model are[16]:

- **pure reactive behavior**: actors work only when receiving a message;
- **state encapsulation**: an actor cannot directly access the inner state of other actors;
- **macro-step semantics**: when an actor receives a message, it execute the corresponding computation completely before handling another message;
- **fairness in message delivering and processing**: a message sent to an actor is eventually delivered and processed by the actor;
- **location transparency**: to send a message, the actor needs just the identity of the receiver and not its location.

1.4 Tasks

Generally, `Task`s are the fundamental unit of work, which can be executed in parallel with other `Task`s[1].

In *D*, `Task`s are provided by the *D* standard library in the package `std.parallelism`. They can be executed directly in a new thread or submitted to a `TaskPool` for execution. A `TaskPool` abstracts a task queue and its worker threads to efficiently map a large amount of tasks to a smaller number of threads. There can be more than one `TaskPool` in a program, for example they could have different priorities for CPU scheduling. Each worker thread is bound to a task queue (and thus to only one `TaskPool`). The task queue contains `Task`s that have been submitted to the pool and await execution; a worker thread executes the `Task` in front of the queue, if the queue is not empty, else it sleeps.[17]

`Task`s are designed to be one-shot terminating actions, so in a `Task` there cannot be an infinite loop or an unsignaled wait, as it would prevent the `Task` termination and the `TaskPool` would not execute the next `Task`.

1.4.1 Usage

The module `std.parallelism` contains the definitions for managing `TaskPool`s and `Task`s[17].

A `Task` can be created with its `struct` constructor or with the helper function `task`; the created `Task` can be executed in a new thread with the corresponding method `executeInNewThread` or submitted to a `TaskPool` with `TaskPool.put`.

To get a lazily instantiated `TaskPool`, the module provide the function `taskPool` but, if a deeper configuration is needed, the `TaskPool` constructor can be used.

2 Goal/Requirements

2.1 Premise

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [2].

As the project consists in the implementation of an actor framework based on message passing for the D language, henceforth in this document the terms "project", "library" and "framework" SHALL be considered synonyms.

2.2 Requirements

The library MUST provide a base extensible component for actors, which SHALL implement the three basic actions: `send`, `create`, `become`. The framework is REQUIRED to allow sending a message to a remote actor, i.e. an actor not located within the same execution context (so the two communicating actors are not sharing memory),

and moreover, the location of the receiving actor SHALL be abstracted and SHALL be transparent to the user [13].

The project SHOULD optimize the resources used by the implementation. Also, the framework SHOULD decouple actors and logical threads, to provide the most lightweight implementation of the former as possible.

2.3 Scenarios

The users of the library are expected to import the library, extend the basic actor component and, for each extension, define a reaction to messages, which need to be registered in the system (only if they are sent to or received from a remote actor).

The library is advisable for every problem solvable with an actor-based architecture.

2.4 Self-assessment policy

2.4.1 Quality assessment

The project quality will be assessed by analyzing the design of the framework: a good design must respect the requirements while being easily extensible and using the least amount of resources as possible.

2.4.2 Effectiveness assessment

The library effectiveness will be measured by counting the number of steps required to setup an application which uses the actor model, that is, the less steps are needed, the more effective the framework is.

3 Requirements Analysis

3.1 Implicit hypothesis

1. Messages will be eventually delivered, but there cannot be any assumption on the arrival order and delay.
2. An actor can only handle one message at one time, so incoming messages should be stored for later handling.

3.2 Implicit requirements

1. The library SHALL define the concepts of "message", "behavior", "system".
2. The project SHALL provide an abstraction for an actor address.
3. The framework MUST be able to marshall and unmarshall messages for remote communications.

4. Given hypothesis (2) (3.1), every actor SHOULD have a personal mailbox to enqueue received messages.

5. Actors can change their reactions to messages; so messages which cannot be handled by the current actor's behavior SHALL be enqueued in another queue of unhandled messages, which will be reconsidered on the next actor's behavior change.

3.3 Non-functional requirements

- Every communication and processing operation SHOULD happen in the least time possible.
- The library SHOULD optimize all the used resources. Also, to allow the presence of numerous actors, the framework SHOULD avoid tying actors to threads.

As the actor model unifies object-oriented (OO) programming and concurrency, the OO approach is the best suited for this project as it allows to define a set of predefined basic operations and let the user of the library extend the classes to their need, while the concurrent aspect will be provided by the library itself internally, by using the tools of *D*.

4 Design

4.1 Structure

As the model and the library are based on message passing, the most important entity to represent is the message itself (figure 1).

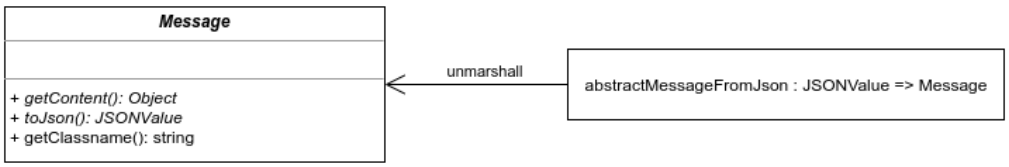

Figure 1: Message class diagram

The abstract class `Message` is the base for all exchanged messages.
Also, for messages which need to be sent remotely, an unmarshaller must be defined, which will convert a `JSONValue` into a `Message`.

The other most important entity is the actor (figure 2).

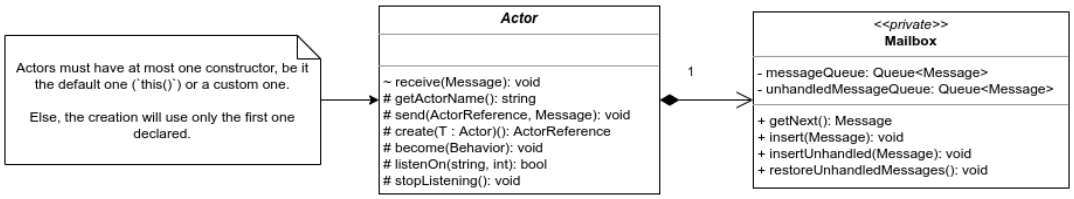

Figure 2: Actor class diagram

As actors handle one message at a time, they have a `Mailbox` to store received-and-to-be-processed messages and unhandled messages (received messages which cannot be handled by the actor's current behavior).

The `Mailbox` class is private, because other entities do not need to know how the system stores the messages.

To react to messages, `Actor`s must define their `Behavior` (figure 3).

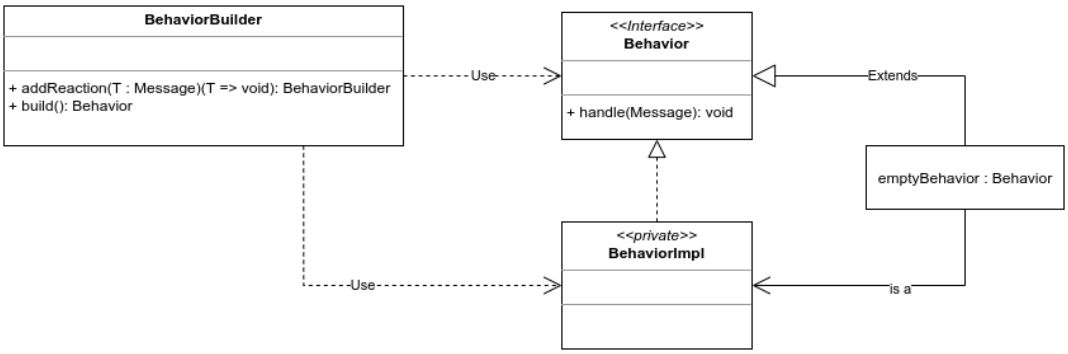

Figure 3: Behavior class diagram

`Behaviors` can and should be created by using the given builder class.

Actors, at the creation and if not specified otherwise in their extensions, have an empty behavior (figure 3, `emptyBehavior`), which cannot handle any `Message`.

The best way to interact with `Actor`s is through `ActorReference`s, which abstract the location of the `Actor` itself (figure 4).

`ActorReference`s can be obtained by creating a new actor with `ActorSystem.create`, `Actor.create` or `ActorSystem.getRemoteReference`. `Actor`s should be always handled through references.

To glue all the framework aspects, the `ActorSystem` singleton has been designed to handle actor creation, message marshalling and unmarshalling, socket listening and logging (figure 5).

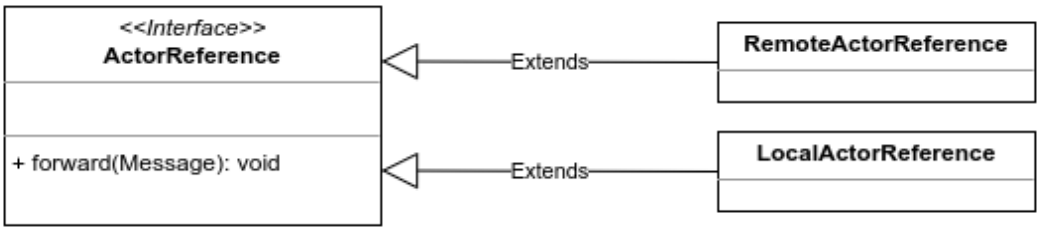

Figure 4: Reference class diagram

The system itself has been divided into smaller components, each of which handles a specific category of tasks to minimize responsibilities of the system class itself.

Package subdivision (figure 6) has been defined also to exploit the `package` visibility modifier of the language (see section 5)[12].

4.2 Behaviour

Actors are the only entity with a meaningful state, which is shown in figure 7.

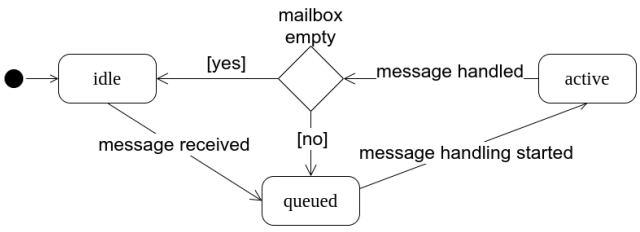

Figure 7: Actor state diagram

4.3 Interaction

Locally connected actors interact directly (through their references), as shown in figure 8, where references have been abstracted away. Figure 8 represents the situation implemented in the example code for local actors.

Communication between remotely connected actors needs to perform some extra steps (see figure 9), as the message must be sent through an internet interface. The figure shows the implementation example for remote actors.

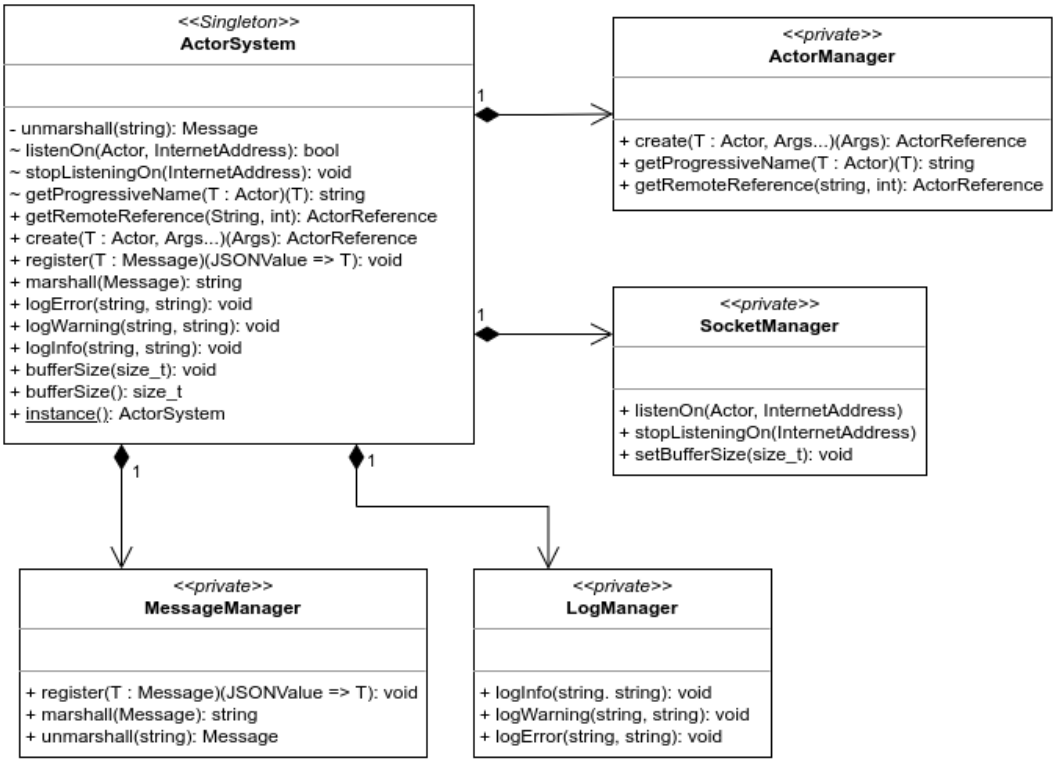

Figure 5: System class diagram

5 Implementation Details

The build tool *DUB* allows to generate a documentation by following the official specification with the command `dub build --build=docs`[3].

The generated documentation (for version `v1.0.0`) is hosted on Github Pages.

5.1 Message passing reimplemented

Even if *D* supports message passing with functions in the `std.concurrency` package, they are related to local-only threads, so for this framework their adoption was not feasible, as its focus is on remote communication and decoupling actors from threads. The reimplementation of the message passing mechanism used Object-Oriented Programming to abstract the location of actors and to allow a seamless message send to both remote and local actors. Also, the new implementation offered the possibility to unbind actors and threads, as explained in 5.3.

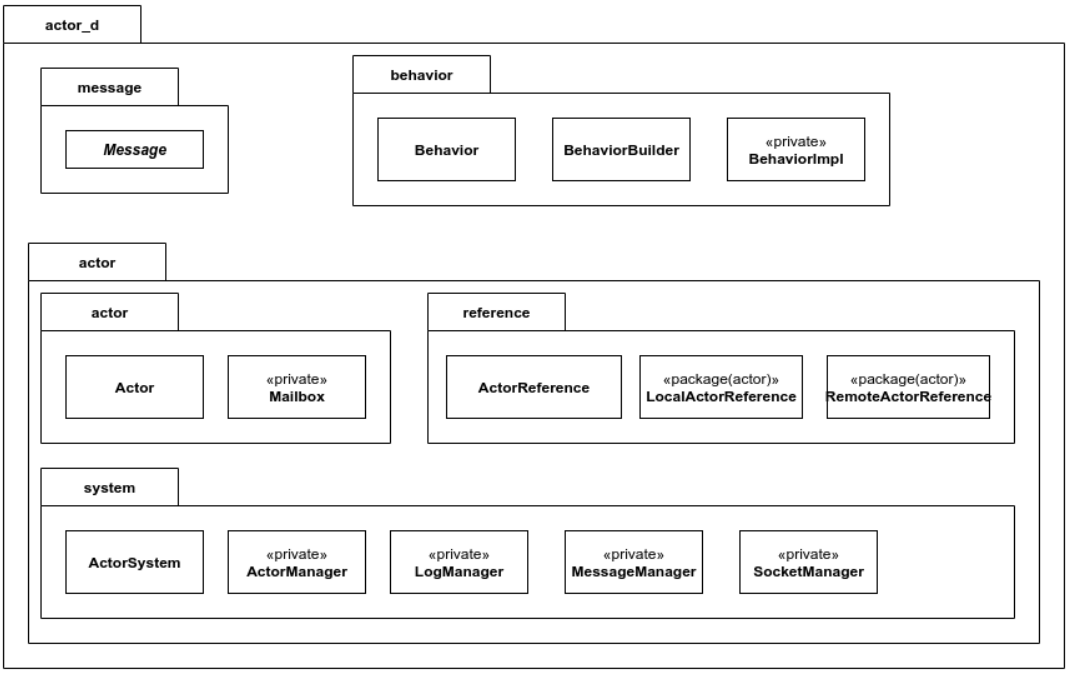

Figure 6: Package diagram

5.2 Mutual exclusion

As a lot of structures needs to be accessed by concurrent control flows, they have been guarded by using mutex variables, which are available in the *D* package `core.sync.mutex`[15].

Examples of said structures are the actor's mailbox and state (with two separate mutexes).

When a snippet of code contains method calls like `lock_nothrow()` and `unlock_nothrow()`, then the block between the two calls is guarded by at least a mutex.

5.3 Actor control flow abstraction

The actor control flow has been abstracted by using `Task`s (see 1.4).

The `TaskPool` is configured automatically and lazily by the getter `taskPool`, which, on the first call, creates a new `TaskPool` with a number of worker threads equals to the number of CPUs of the system minus 1, which is left to the other operations; in this framework, the spare thread is assumed to be used as the socket-listening one (see 5.7.2).

The design of `Task`s as terminating algorithms led to tracing the actor state (see 5.4) and the following snippet of code:

```
actorActivityMutex.lock_nothrow();
if (mailbox.empty)
```

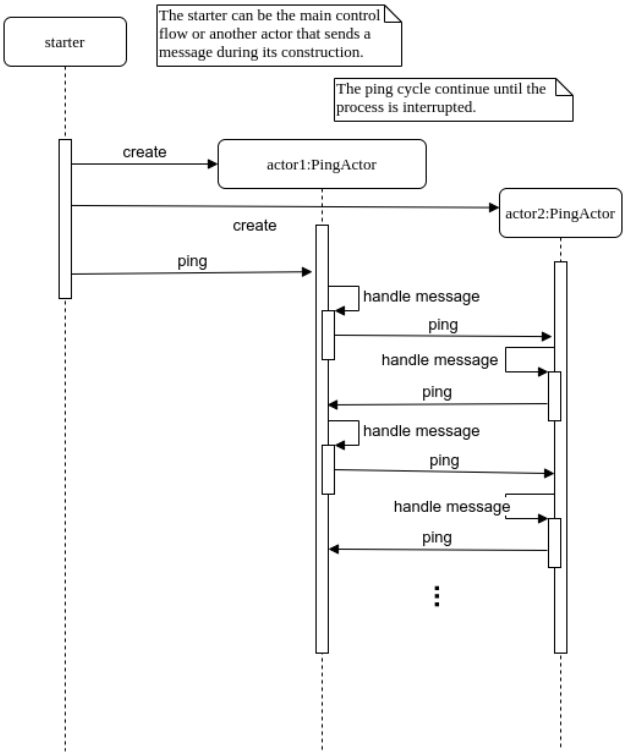

Figure 8: Local actors sequence diagram

```
{
    this.activityStatus = ActorActivityStatus.idle;
}
else
{
    this.activityStatus = ActorActivityStatus.queued;
    taskPool.put(task(&handleNextMessage));
}
actorActivityMutex.unlock_nothrow();
```

This code checks if the actor mailbox is empty and, if not, submit a new `Task` to handle the next message (also in any case updates the actor state), thus implementing the actor event loop.

5.4 Actor status

The actor status is traced to avoid handling more than one message at a time.

The actor status can assume 3 values:

1. **idle**, if no message is being handled and no `Task` has been submitted;

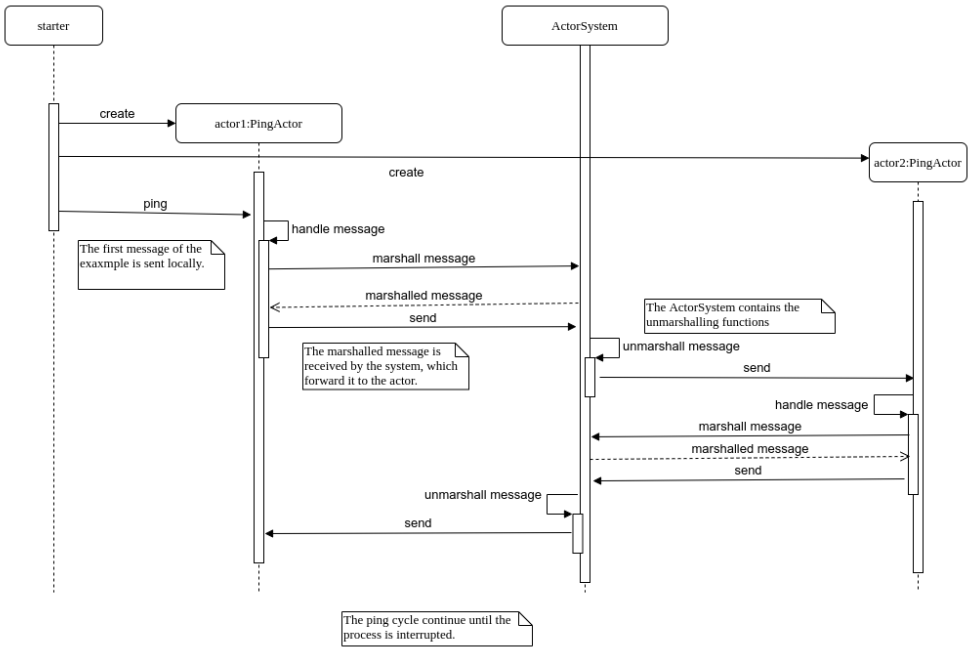

Figure 9: Remote actors sequence diagram

2. **queued**, if no message is being handled but one `Task` has been submitted;
3. **active**, if a message is being handled.

A new `Task` is submitted to the pool only if the actor is (or will be) idle and the mailbox is not empty, which occurs:

- After handling a message (see 5.3).
- After receiving a new message:

```
this.mailbox.insert(message);
actorActivityMutex.lock_nothrow();
if (this.activityStatus == ActorActivityStatus.idle)
{
    this.activityStatus = ActorActivityStatus.queued;
    taskPool.put(task(&handleNextMessage));
}
actorActivityMutex.unlock_nothrow();
```

- After changing behavior, as old unhandled messages are reconsidered:

```
this.currentBehavior = behavior;
this.mailbox.restoreUnhandledMessages();
if (!this.mailbox.empty)
{
    actorActivityMutex.lock_nothrow();
    if (this.activityStatus == ActorActivityStatus.idle)
    {
        this.activityStatus = ActorActivityStatus.queued;
        taskPool.put(task(&handleNextMessage));
    }
    actorActivityMutex.unlock_nothrow();
}
```

5.5 Message content is Object

The *D* language distinct primitive types from aggregate types (`struct`s and `union`s) and from `class` types.

As they have no common ancestor, only one of these category could have been used.

It has been chosen to use `class` types, as they can wrap easily both primitive types and `struct`s/`union`s, so the base type of the message content is `Object`.

5.6 Message de/serialization format

The chosen format of serialized messages is *JSON*. This choice has been driven by the will to use a standard (and not create a new custom format) and the availability of the module `std.json`[14] in the *D* standard library, which makes working with *JSON*s straightforward.

5.7 Sockets

5.7.1 Socket type

As the actor model requires the certainty of the message arrival, the sockets used are `TcpSocket`s. They are all set to be non-blocking to avoid making the `Task`s hang.[5]

5.7.2 Socket listening

An `Actor` can request to listen to a specific `address - TCP port` pair which is translated internally in a socket, and the `ActorSystem`, if possible, binds the new socket to the requesting `Actor`. `Actor`s can listen to only one socket at one time, so the number of open sockets won't exceed the number of actors. This mechanism opens a new socket for each `Actor` who wants to listen to incoming remote messages.

Reading sockets are watched for readiness by a separate thread.

The thread is started by the `ActorSystem` when a new socket has been requested and there are no other listening socket, and it stops when all the active connections are closed.

When a socket is ready, a new `Task` is submitted to handle the incoming message, if any.

`Actor`s, as they can ask to listen to an interface, can also ask to stop listening to it.

The socket watching is continuous and without sleeps, so the thread is always working.

5.8 Package visibility

Some implementation classes and methods (for example `ActorReference` implementations and `Actor.receive` method) are visible only to their package (`actor_d.actor`, see packages in figure 6).

This allows other specific classes to see them, but not subclasses (with `protected` visibility) and other files (with `public` visibility), as they are needed only to perform basic internal operations, and there are other methods to "access" them, respectively by the `ActorSystem` actor creation and remote reference retrieval (which both returns an `ActorReference`) and the `LocalActorReference.forward`.

5.9 No `interface` for `Message`s and `Actor`s

Both `Message` and `Actor` classes use methods based on their `.classinfo`, which behaves differently whether is is called on `interface`s or `class`es: on the former it returns the `TypeInfo` object of the interface, while on the latter it returns the `TypeInfo` object of the most derived class [6]. `Message`s have a method which returns their name based on the class name, which is used to unmarshall them in a remote communication, which is done automatically by the `Actor System` on reception; `Actor`s, on construction, ask their name (see 5.10) to the `ActorSystem`, which uses the `TypeInfo` object to retrieve the name of the most derived class.

So, for this two classes, `interface`s would have broken this automatism, so their definitions had to use `abstract class`es.

5.10 Actor naming

Actors, on creation, asks a name to the `ActorSystem`, which is generated progressively in the order of the creation itself.

It is used to specify which actor prints a given message.

6 Self-assessment / Validation

To validate the project, it will be used the policies specified in section 2.4.

The framework design is simple and modular. Also the implementation has a more-than-average optimization, as it uses full concurrency and only a thread for socket lis-

tening, but it might be optimized further by activating the connection acceptance only when there is an incoming connection request (now it uses a busy waiting mechanism).

The framework is effective because, as seen in the examples in section 8, a basic communicating actor system can be achieved with a few steps and lines of code.

To check the correctness of the produced library there have been developed some unit test (which can be launched with `dub test`) for the base components and operations:

1. Behavior building.
2. Behavior usage.
3. Mailbox methods.
4. Message de/serialization.

To validate the correct behavior of the framework, the two developed examples are used.

7 Deployment Instructions

To run `actor-d` examples:

1. Install *DMD* or *LDC2* (*D* official download page).
2. Change directory to the example you want to run (`local-actors` or `remote-actors`).
3. Run `dub run`.
4. The examples will go on indefinitely, to stop them press `CTRL-C`.

To use `actor-d` as a library, follow this steps:

1. Install *DMD* or *LDC2* (*D* official download page).
2. Create a new Dub project with `dub init` (configure it as you like).
3. Search for `actor-d` in the official package registry:
   - If you find it there, then add it to your project dependencies with `dub add actor-d`.
   - If you do not find it in the registry, then clone the git repository and run in the project folder `dub add-local <path-to-git-repo>`, where `<path-to-git-repo>` is the path of the cloned repository.
4. Now you can `import actor_d` in your code and start using it.

8 Usage Examples

The two example projects are reported below, and the common code between the two examples is in its own section, as it contains the basic imports and declarations.

8.1 Common code

```
import std.stdio;
import std.json;
import std.conv : to;
import core.thread.osthread : Thread;
import core.time : seconds;
import std.parallelism : taskPool;

import actor_d;

class Integer
{
    int value;
    alias value this;

    this(int i)
    {
        this.value = i;
    }
}

class PingMessage : Message
{
private:
    Integer content;

public:
    this(int value)
    {
        this.content = new Integer(value);
    }

    override Integer getContent() const
    {
        return cast() this.content;
    }

    override JSONValue toJson() const
    {
        JSONValue messageAsJson = [ "value": this.content.value ];
        return messageAsJson;
    }
}
```

```
PingMessage jsonToPingMessage(JSONValue json)
{
    return new PingMessage(json.object["value"].get!int);
}
```

8.2 Local actors

```
class PingActor : Actor
{
public:
    this(Behavior behavior)
    {
        this.become(behavior);
    }
}

void main()
{
    ActorReference actor1;
    ActorReference actor2;
    Behavior pingBehavior1 = new BehaviorBuilder()
                                .addReaction!PingMessage((ping) {
        writeln("Actor1: ping " ~ ping.getContent.value.to!string);
        Thread.sleep(seconds(1));
        actor2.forward(new PingMessage(ping.getContent.value + 1));
    }).build();
    Behavior pingBehavior2 = new BehaviorBuilder()
                                .addReaction!PingMessage((ping) {
        writeln("Actor2: ping " ~ ping.getContent.value.to!string);
        Thread.sleep(seconds(1));
        actor1.forward(new PingMessage(ping.getContent.value + 1));
    }).build();
    actor1 = ActorSystem.instance.create!PingActor(pingBehavior1);
    actor2 = ActorSystem.instance.create!PingActor(pingBehavior2);

    actor1.forward(new PingMessage(0));
}
```

8.3 Remote actors

```
class PingActor : Actor
{
private:
```

```
    ActorReference remoteActor;
    Behavior behavior;

    void handlePingMessage(PingMessage ping)
    {
        writeln("Actor: ping " ~ ping.getContent.value.to!string);
        Thread.sleep(seconds(1));
        /* Uncomment if you want to stop the pinging after
         * a defined number of pings.
        if (ping.getContent.value > 2)
        {
            this.stopListening();
        }
        */
        this.send(remoteActor, new PingMessage(ping.getContent.value + 1));
    }

public:
    this(ushort port, ushort remotePort)
    {
        this.behavior = new BehaviorBuilder()
            .addReaction!PingMessage(&(this.handlePingMessage))
            .build();
        this.become(behavior);
        this.listenOn(port, "localhost");
        this.remoteActor = ActorSystem
            .instance
            .getRemoteReference("localhost", remotePort);
    }
}

void main()
{
    /// Create the actors from the ActorSystem instance.
    ActorReference actor1 = ActorSystem
        .instance
        .create!PingActor(cast(ushort)10000u, cast(ushort)10001u);
    ActorReference actor2 = ActorSystem
        .instance
        .create!PingActor(cast(ushort)10001u, cast(ushort)10000u);

    import std.functional : toDelegate;
    /// Register the unmarshaller for the excahnged message types.
    ActorSystem.instance.register!PingMessage(toDelegate(&jsonToPingMessage));
```

```
    /// Change the message buffer size (defaults to 1024).
    ActorSystem.instance.bufferSize = 16;

    /// Starts the message chain.
    actor1.forward(new PingMessage(0));
}
```

9 Conclusions

The developed library provides a basic implementation for an actor model where:

- Actors can communicate with other actors located both locally and remotely, transparently.
- Actors can change behavior, and store messages unhandled by their current behavior for future handling.
- Actors can create other actors and can be created by the system.

Messages exchanged by the framework are serialized and deserialized by using the *JSON* standard format, so it should have a small and smooth learning curve.

The framework also decouples actors from threads, as the message handling is delegated to tasks, this allows the use of actors in lightly-equipped systems, with just a little delay in message processing (due to task queuing).

The library provides a simple logging interface, too.

9.1 Future Works

A future work can optimize the number of sockets required by 2 communicating remote `ActorSystem`s: now every `Actor` who listen to remote messages occupies a socket; this can be optimized further by having just one open socket which receives all the messages, and dispatching them internally by using the already-existing `Actor` name. The expected changes are mostly inside the `ActorSystem` implementation and they barely the interfaces and extensible classes, but to fully exploit the new system, `Actor`s may probably slightly change their implementation, by saving the socket address and the sender name in a particular data structure.

Other than that, the library can be enhanced with some small refinements, like more tests and other speed and hardware-resources optimizations.

9.2 What did I learned

The implementation of the actor model taught me how to code non-blocking cycling control structures (specifically a `while`, in actor message handling); how to abstract the location of an interlocutor and the communication channel between them, while

also handling the marshalling and unmarshalling of dynamic messages; how to decouple objects from hardware resources and how to optimize the available resources.

In a more general optic, working on this project allowed me to learn how to design an extensible framework based on some preset indications and requirements. I also had to try to impersonate a potential user and think about what he could want to do or what reaction he would expect from the system.

10 Stylistic notes

- **Bold text** is reserved for keywords or key concepts;
- `Monospace text` defines CLI commands, codes blocks or in-code function names;
- *`Monospace italic text`* indicates names of softwares or languages;

References

[1] Ananth Grama et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley, 2003. ISBN: 0201648652. URL: http://books.google.com/books?hl=en&lr=&id=B3jR2EhdZaMC&oi=fnd&pg=PR17&dq=Purdue+University+Grama&ots=uCCIvLGaHJ&sig=B72Ct5UjD1UI60-JkXLuYbiJr1A.

[2] Scott Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. 1997. URL: https://www.ietf.org/rfc/rfc2119.txt.

[3] *Build Types*. URL: https://dub.pm/package-format-json.html#build-types.

[4] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: (1973). URL: https://www.ijcai.org/Proceedings/73/Papers/027B.pdf.

[5] Christopher E. Miller, David Nadlinger, and Vladimir Panteleev. *std.socket*. URL: https://dlang.org/phobos/std_socket.html.

[6] D Language Foundation. *.classinfo Property*. URL: https://dlang.org/spec/property.html#classinfo.

[7] D Language Foundation. *Dub: advanced usage*. URL: https://dub.pm/package-format-json.html.

[8] D Language Foundation. *Dub: getting started*. URL: https://dub.pm/getting_started.

[9] D Language Foundation. *Inline Assembler*. URL: https://dlang.org/spec/iasm.html.

[10] D Language Foundation. *Introduction*. URL: https://dlang.org/spec/intro.html.

[11] D Language Foundation. *Linkage Attribute*. URL: https://dlang.org/spec/attribute.html#LinkageAttribute.

[12] D Language Foundation. *Visibility Attribute*. URL: https://dlang.org/spec/attribute.html#VisibilityAttribute.

[13] Carl Hewitt. "Actor Model of Computation: Scalable Robust Information Systems". In: (2015). URL: https://arxiv.org/abs/1008.1459v37.

[14] Jeremie Pelletier and David Herberth. *std.json*. URL: https://dlang.org/phobos/std_json.html.

[15] Sean Kelly. *core.sync.mutex*. URL: https://dlang.org/phobos/core_sync_mutex.html.

[16] Alessandro Ricci. *Module 3.2 - Actors*. URL: https://www.unibo.it/it/didattica/insegnamenti/insegnamento/2020/412598.

[17] David Simcha. *std.parallelism*. URL: https://dlang.org/phobos/std_parallelism.html.

[18] Wikipedia. *D - Programming paradigms*. URL: https://en.wikipedia.org/wiki/D_(programming_language)#Programming_paradigms.

[19] Wikipedia. *D (programming language)*. URL: https://en.wikipedia.org/wiki/D_(programming_language).