Comparison of Active Objects and the Actor Model

Thomas Rouvinez, *Candidate MSc. Computer Science , University of Freiburg,*
Advisor: Anita Sobe, *Post Doc, University of Neuchâtel*

Abstract—This paper compares two patterns of concurrency: the Actor Model and the Active Object Model. The Actor Model is a pattern that decouples function invocation from execution and offers both inherent thread-safety and scalability. The Active Object Model inherits from the Actor Model and as such presents the same major properties. As the Active Object Model inherits from the Actor Model one may think they are equivalent. Throughout this paper we show that both patterns differ in terms of structure design and communication protocols. These differences affect the choice of pattern for specific applications as each pattern has strengths and weaknesses.

Keywords—Actor Model, Active Object, Concurrency, AKKA framework.

1 Introduction

Multi-core enabled machines hit markets in the early 2000s. Nowadays multi-core CPUs have largely replaced single cores. The before common sequential programs were naturally deterministic. They can be read top-bottom while preserving the temporal order of instructions [1]. As a counterpart, determinism leads to bad performance on multi-core architectures. Indeed, operations are forced to execute one at a time and in order, regardless of any opportunities for parallelism. The goal of concurrent programming lies in efficiently using the existing multi-core resources, resulting in improved performance. In turn, exploiting parallelism requires programs to be as scalable as possible.

A program achieves maximum scalability by ensuring that all cores of a CPU are fully loaded throughout its execution. To reach high levels of scalability the span of the code should be minimized. The span represents the minimum sequence of sequential operations the code should perform throughout its execution (serial bottleneck). The potential speedup of parallel programs is directly limited by their sequential portion of code (Amdahls law). A well thought scalable program will automatically perform better regardless of the number of cores provided.

Besides the performance gains parallelism can bring, concurrent programs lead to hazards such as data races, violated consistency, liveness and progress issues caused by multiple threads accessing shared data. Imposing a strict parallel structure to a program helps avoiding such hazards but limits its scalability, hence its performance. Developing concurrent applications remains a challenge requiring deep knowledge of concurrency in hardware as well as in software. Debugging deadlocks or data inconsistencies becomes even harder due to the non-deterministic nature of concurrent programs. To enable faster and safer developments, one solution consists in using inherently concurrent patterns with implicit synchronization mechanisms as conceptualized in the Actor Model [2].

Introduced by Carl Hewitt in 1973, the Actor Model uses message passing in combination with encapsulated states. It is inspired by the multi-agent design of artificial intelligence. Agents are grouped into layers representing different levels of abstraction of the problem to solve. Each agent belongs to one level and may spawn new sub-agents to handle the workload or new tasks. Actors inherit from agents and provide desirable properties like data encapsulation, fair scheduling and location transparency [3]. The Actor Model inspired other patterns like the Active Object Model which inherits most of its properties.

In this paper, we focus on comparing the Actor Model and the Active Object Model. They exhibit desirable properties such as a high capacity to split the workload into lightweight, independent tasks to achieve maximum and automatic scaling. We also explore the different guarantees regarding consistency and concurrency of both models. In our experiments, we will use the AKKA framework as it already supports implementations of both patterns.

This paper is organized as follows: Section III presents both the Actor Model and Active Object patterns. Section IV explains similarities between the two patterns in terms of method invocation, execution and message passing. Structural differences are also discussed. Section V provides implemented examples to highlight some of the differences between the Actor Model and Active Object. Section VI addresses possibilities of mixing both

- *Thomas Rouvinez: thomas.rouvinez@unifr.ch*
- *Dr. Anita Sobe: anita.sobe@unine.ch*

patterns to achieve various combinations of desirable properties in concurrency. Finally, section VII presents the state of the art in bringing concurrency even within actors/servants.

2 PATTERNS OVERVIEW

To understand the differences and similarities between the Actor Model and the Active Object Model, we first introduce their basic principles separately.

2.1 Actor Model

The Actor Model (AM) is a pattern using actors as its universal primitive to perform concurrent computations [4]. Parallel computations are realized by decomposing an intent into subtasks and distributing them over suitable actors. An actor may be considered as a worker that processes messages sent by other actors. To better understand how parallelism is achieved, we will present the structure of actors as well as their communication protocols.

An actor is an object that encapsulates both control and data flow into a single object (see Figure 1). Actors feature three main components: (1) an internal state, (2) a mailbox and (3) a set of functions to query and modify the internal state. The internal state is a combination of variables and objects that represent the knowledge of the actor. This data are encapsulated within the actor and only the actor itself may access it. Actors only have influence on the state of another actor by sending messages.

Sending a message is an asynchronous, non-blocking operation for the sender. Messages use arbitrary communication topologies [5] and can contain any type of object. An actor can only interact with another actor if it knows its address. Security is achieved through the scope of addresses an actor is allowed to send messages to. Upon reception of a new message, the actor enqueues it into its respective mailbox. Messages are then dequeued one at a time for processing.

To process the messages, each actor comprises a set of functions. Actors can be described as Turing machines that are programmed to react according to a specific input. The actor progresses as it dequeues messages from the mailbox and matches their type to the corresponding block of statements to execute. When the processing of the current message is finished, the actor will dequeue the next message from the mailbox as long as it still contains messages.

Actors have a defined life cycle: Once spawned, they are started and wait until they receive messages. When an actor is not used anymore, another actor may send a signal to destroy it. Each actor may spawn new actors, which is done for similar reasons as recursive functions call themselves, i.e to reduce the problem until it becomes small enough to be processed. Spawning new actors is

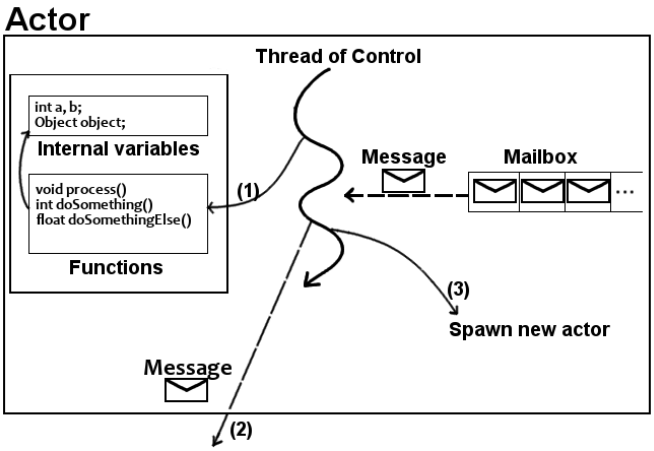

Fig. 1. The structure of an actor. In response to a message, an actor can (1) change its internal state, (2) send a message, (3) spawn new actors or migrate to another host.

an extension mechanism that may be exploited to boost scalability and concurrency. Note also that each part of an actor could be an actor itself. For example, the mailbox could be an actor that stores messages, or the internal state a database actor. We conclude that actors are highly scalable and thread-safe. The AM handles the known concurrency difficulties for the programmers and provides guarantees, which we discuss in section 2.3.

2.2 Active Objects

The Active Objects Model (AOM) is another pattern for concurrency largely inspired by the AM. An active object features private data and methods like any passive object (common object). The difference is that an active object runs in its own thread of control. There is not a single way to implement the AOM. However, most commonly the goal is to decouple method invocation from execution to simplify object access [6]. This property enables the client to continue its work while waiting for an answer. Furthermore, it is not possible that two method calls of one particular active object run at the same time [7][6]. This condition guarantees that the implementation of an active object does not require additional thread-safety mechanisms [7].

To better understand how decoupled method invocation and execution work in active objects, let us review the components of the model as shown in Figure 2. The first essential notion is that the client and the active object queried run in separate address spaces and threads. We distinguish the client space from the active object space. In figure 2 the limit between the two spaces is shown by the vertical bar below the proxy.

In the *client space* we find the **client** and the **proxy**. The proxy is an interface object between the client space and

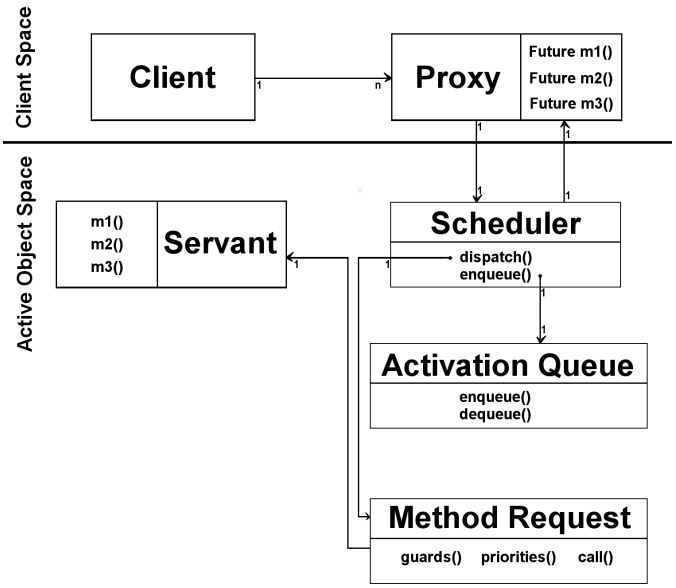

Fig. 2. Architecture of an active object [6].

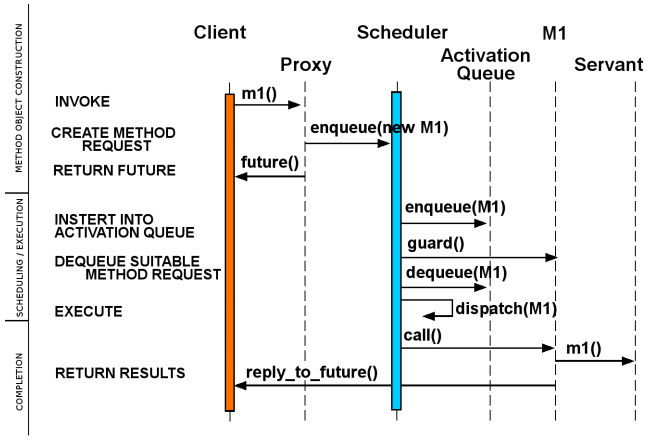

Fig. 3. Workflow of a request with an active object [6].

the active object space. The client can call a function in the same way as calling a function on passive objects. The proxy takes care of translating the function call and its parameters into a method request and forwards it to the **scheduler** for asynchronous execution.

The proxy also immediately returns the address to a future to the client. A future is a reserved placeholder for the results of the active object. When the active object finished the computation, it fills the results into the shared future. To know if the results have arrived, the client can test the state of the future. A future can be in pending, succeeded or error state and may change its state only once in its lifetime [7].

- Pending. Default state when the future is created. No results have been put into the future yet. It also means that no error has happened yet.
- Succeeded. The results have been computed and successfully stored in the future.
- Error. Futures also work as error handling mechanisms. If errors happen during computation in the active object, this state will be enabled and will report the error.

Note also that futures support synchronous waiting, synchronous waiting with timer or asynchronous answers.

Moving to the *active object space*, we find on the frontline the **scheduler**. An active object contains one scheduler that manages one **activation queue** of pending method requests enqueued by the proxy. It is the very mechanism that ensures the client space is decoupled from the active object space. The scheduler not only manages the activation queue, but also selects which method request is eligible for dequeuing and execution. This selection is based on multiple criteria like ordering of method requests and synchronization constraints. To ensure all synchronization constraints are met before dequeuing and executing a method request, the scheduler uses **guards**.

Guards check for execution order, potential writing operation hazards and the current state of the active object to know if the function called by the method request can be executed. Once a method request in the activation queue is validated by the guards, the scheduler dequeues and dispatches it to an available **servant**. A servant implements one or many methods referenced in the proxy. It defines the behaviour and state to be modeled as an active object [6]. In other words, the servant will execute the actual computation and return the results to the already established future. This concludes the execution of a method request using the active object model.

2.3 Pattern guarantees

The AM is message-based, asynchronous and can create new actors at will. Such properties require guarantees and inconsistency for robust information integration [4]. As the AOM largely inherits from the AM, the following guarantees are common to both models. To avoid repeating the same concepts, we will focus on the AM.

A message sent should be received. The AM uses best-effort message delivery. This means that each message has the same priority, no actor gets any guarantees in terms of quality of service. A message is delivered zero times or once (at-most-once[1]). The asynchronous

1. http://doc.akka.io/docs/akka/2.3.0-RC1/general/message-delivery-guarantees.html

communication between actors combined with the lack of guarantee of message delivery can make it difficult to know if a message is lost or still in transit. Moreover, actors can be distributed and moved over multiple machines in a network. In this context, issues such as transient or permanent packet losses or node failures may occur. The AM does not specify the specific handling of lost messages and this responsibility falls to the different implementations. The literature usually considers the environment to be theoretically fully stable and tends to simply ignore the issue [8][4], except for the work presented in [9] where the authors propose the Transactor Model.

No message order guarantee. Messages in the AM are fully dependent on the host transport layer concerning message delivery. Therefore, there is no guarantee on when a message will be received by an actor. When considering two actors, messages exchanged between the first and the second should not be received out of order. But if two actors send a message at the same time to a third actor, there is no way to know in which order they will be received. Actors do not necessarily receive messages in the same order as they were sent [4]. Finally, there is no guarantee either on the message dispatch order [4]. It is important to differentiate receive order and dispatch order: there is no guarantee that a message will be dispatched right after it is received or in any limited number of steps.

Unbounded concurrency. The AM was directly inspired by the multi-agent system. According to the principle of subsumption, a problem is modeled as an intent the agents are trying to resolve. The intent is divided into levels of abstraction until parts of the problem are small enough to be handled by a single agent. The actors are no different and can arbitrarily spawn new actors, yielding an unbounded concurrency.

Inherent concurrency. The AM allows only one message to be processed at a time to avoid concurrent hazards. The usual explicit synchronization mechanisms like locks or semaphores are no longer required as only a single thread controls the whole actor. This provides an easy way to implicitly deal with concurrent accesses on data.

3 Comparison between the Actor Model and Active Objects

The original AM defines only the concepts and guidelines any implementation should respect. This leaves room for adaptations and extensions. In this section, we go through the similarities and differences between the AM and AOM.

3.1 Structure

Structurally, the AM and AOM use the same concepts and components. Indeed, in the AOM the client can be considered as an actor sending messages to other actors. The proxy is replaced by the scope of addresses in each actor, as an actor may send messages only to known addresses. Once a message is received, both models store them in queues (mailbox, activation queue respectively) where they await to be processed. In both cases the messages in the queues are tested and dequeued once their synchronization requirements are met. A dequeued message is then processed by the actor, servant respectively. Computation results are then returned if need be to the client, requesting actor respectively. The AM uses messages and the AOM futures for this purpose.

The first difference lies in how these components are organized. The AM encapsulates in each actor the equivalent of the scheduler, activation queue, servant and proxy. The AM does not require a proxy to interface two actors exchanging messages. Since each actor knows a definite set of addresses where it is allowed to send messages to, it is assured the receiving actor has the means to process the messages. The AOM relies on the proxy to select an active object that features the right set of methods to process the request. This implies that the proxy must be aware and keep track of all the available active objects and their state. The AM leaves this responsibility to each actor as they are self aware and responsible for querying the right actors. In case of failure of the proxy, no work can be distributed anymore in contrast to actors that can reorganize themselves and spawn new actors to replace the failed ones. Another structural difference lies in the queues used to store pending messages. The AM encapsulates an unbounded mailbox in each actor. To achieve the same functionality, the AOM relies on the activation queue which maintains a bounded queue of pending method requests. With unbounded queues the AM guarantees that messages received will be stored and eventually processed at some point in time. At this point different strategies can be implemented: Either wait for the active object's activation queue to free up one slot or spawn a new active object to handle the method requests the original active object cannot accept. If the proxy waits on an active object, it means the caller is blocked.

As mentioned above, the AOM allows to arbitrarily spawn new active objects. This ability is intended as an extension mechanism for the AM and provides an insight on how high scalability is achieved. Scalability is easier with the AM as any actor can be a proxy and distribute workload to other actors. Most implementations of the AOM rely on a thread pool with a limited number of servants. Note that both models try as much as possible to reuse existing actors/active objects rather than creating new ones.

3.2 Messages and communication

In section 3.1 we linked the various components from both models and showed how the same tasks are

accomplished in different ways. One notable difference lies in messaging and returning computed results. The communication protocol in the AM relies only on messages carrying information from one actor another. Upon reception the actor reads the message content and executes the corresponding block of statements. A message triggers function calls, coroutines, resource seizures, scheduling, synchronization, continuous evaluation of expressions, etc. [8]. The semantics of the messages do not have any restrictions and are independent from the sender [4]. The AOM, however, is using only method requests and futures as message types. Method requests carry contextual information such as parameters and context for the execution of the selected method. They have defined semantics and are not user generated. Hence, the AM offers freedom of message semantics whereas the freedom in the AOM is limited. The same applies to returning computed results: The AM will return results by returning a message whereas the AOM will return a future. Indeed, futures can be used to take care of error handling. In case of an error occuring during execution, the active object changes the state of the future to Error and stores the cause. In contrast, the AM is based on the subsumption principle and usually detects a problem when an actor's state differs from its intention (problem to solve). In case of an error, an error message is sent through the layers of actors (based on the abstraction level of the problem) until it reaches the actor with best knowledge on how to manage the error.

Hence, the semantics for messages differ considerably. The AM offers freedom, which can cause the following problems: (1) message size, (2) lack of standards in message semantics. The AM heavily relies on messages passing and as such generates much traffic. To avoid congestion and global slowdown, the message size can be reduced to a minimum, storing only vital information (application and implementation dependent). In the AM message types already convey information about the block of statement to be triggered. For example a printer actor could directly print any incoming message of type string without further testing on it. In contrast to the AOM where the whole context is passed as an argument within a message.

Then with freedom of message semantics comes flexibility. The programmers are allowed to develop their AM systems as close as possible to their existing business systems. The gain, in terms of integration, is instant as no interfacing is required. Messages can be built according to existing data structures to further enable compatibility. Therefore, the complexity of the system is reduced. This flexibility also causes potential inconsistencies and higher responsibilities for the programmer. To avoid these inconsistencies and relieve the programmers, the messages can be standardized. The AOM uses strict standardization of messages; the method requests are validated by the proxy and must respect the function prototypes the active object can handle. Moreover, futures represent a strict structure with defined states. Standardization reduces the degree of freedom for development purposes, but enables cross compatibility with different systems. The AOM allows any passive object to query the proxy of an active object and as such calls for standardized function prototypes. Even if standardization removes all freedom of message semantics, it does not affect the types of arguments passed to the active object. Standards are therefore easier to respect and enforce good development practices.

Different semantics and purposes translate into different communication protocols. The AM allows any actor to send messages asynchronously. Also, a message is delivered at-most-once, which can be compared to the guarantees provided by the UDP protocol. This approach takes advantage of the tradeoff between message overhead and reliability of transmission, yielding a lightweight and fast communication protocol between actors. We previously explained that the AOM's messages were of two types: method requests and futures. If the proxy fails to deliver the method request to the scheduler, no actual computation is performed. Similarly, if the active object fails to deliver or update the future, the client will never get an answer and waits indefinitely. Thus, messages in the AOM require acknowledgements of delivery. Acknowledgements are usually associated with the TCP networking protocol. We can compare futures to TCP acknowledgements as the proxy automatically returns a future for each method request. Moreover, if a failure occurs during the execution of a method request, the future can be used by the client as means to trigger a retransmission of the function call on the proxy.

3.3 Concurrency

From a concurrency point of view, both models share the same goal: Providing an inherently thread-safe and concurrent execution of programs. Concurrent execution is achieved through decoupling method invocation and execution; inherent thread-safety by processing at most one message at a time. The AM handles method invocation by sending messages to other actors. The message is eventually received and processed. By allowing such a loosely-coupled method invocation, the caller is never blocked. In the AOM, the status of the request is accessible by probing the future returned by the proxy. Method requests should not block an active object [6].

4 Pattern Illustration

In this section we provide the traditional "Hello world" program implemented with the AM and the AOM to illustrate both models and some of the differences we highlighted in section 3. We use Akka[2], an open-source

2. http://akka.io/

toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM. Akka offers its framework in Scala[3] and in Java. We chose Akka because it allows to implement both the AM and AOM.

4.1 Implementation of the Actor Model

In Listing 1, we present the implementation of an actor printing any messages it receives to the standard output.

```
public class PrinterActor extends UntypedActor {
  @Override
  public void onReceive(Object msg) {
    if(msg instanceof(TextMessage)){
      System.out.println(TextMessage.getText());
      getSender().tell(new TextMessage("Done"),
          getSelf());
    }
    else{
      unhandled(msg);
    }
  }
}
```

Listing 1. Printer Actor.

To print a "Hello World!" with the AM, we need to send a specific message to an actor that will recognize its type and perform the desired action. We use an UntypedActor class to create the Printer Actor without any pre-defined behaviour. The only mandatory function is "onReceive", triggered each time a message is dequeued from the mailbox. In case the message is not recognized, it is dropped by the "unhandled" function. From now on, any actor that knows the address of the printer actor can send a message of type "TextMessage" as, for example, presented in Listing 2.

```
public class HelloActor extends UntypedActor {
  @Override
  public void preStart() {
    // Create the printer actor.
    final ActorRef printerActor = getContext().
        actorOf(Props.create(PrinterActor.class), "
        printerActor");

    // Send it a message to be printed.
    printerActor.tell(new TextMessage("Hello World!"
        ), getSelf());
  }
  @Override
  public void onReceive(Object msg) {
    if (msg instanceof(TextMessage)) {
      // When the printer actor is done, stop this
      // actor and with it the application.
      getContext().stop(getSelf());
    }
    else
    {
      unhandled(msg);
    }
  }
}
```

Listing 2. Hello Actor.

3. http://www.scala-lang.org/

When the message carrying the string to be printed is processed by the Printer Actor, it will return another message to the Hello Actor. Upon receiving this message, the Hello Actor knows his message has been printed and we stop it. Finally, we need a system to query the Hello Actor and launch the whole process to display the message. Listing 3 presents the creation of such a system and the creation of the whole program. To shut down the system, the function "getContext().system().shutdown()" is used. It is called from a Terminator Actor upon receiving the corresponding message.

```
public class Main{
    public static void main(String[] args) {
        ActorSystem system = ActorSystem.create("
            testSystem");
        ActorRef helloActor = system.actorOf(Props.
            create(HelloActor.class), "helloActor");
    }
}
```

Listing 3. Actor system and launch of the program.

Through this example, the sending of messages and spawning of new actors has been demonstrated. The Hello Actor needs a Printer Actor to fulfill its purpose and since none is already existing, it creates a new one. In the next section, we present the same program but using active objects.

4.2 Implementation of the Active Object Model

In Akka the AOM is implemented with Typed Actors. To make a "Hello World!" program with the AOM, we need a proxy("Printer") that features a callable function for the client.

```
public interface Printer{
    Future<String> printNonBlocking(String msg);
    Option<String> printBlocking(String msg);
}

class PrinterImpl implements Printer{
    private String name;

    public void PrinterImpl(String name){
        this.name = name;
    }

    public Futures<String>printNonBlocking(String
        msg){
        System.out.println(msg);
        return Futures.successful("Worked");
    }

    public Option<String> printBlocking(String msg){
        System.out.println(msg);
        return Option.some("Worked");
    }
}
```

Listing 4. Implementation of the servant.

We declare an interface and the implementation of that interface to realize the proxy and the servant (see Listing 4). In the main class of Listing 5, set up the active object and call the respective function.

```
public class Main{
    public static void main(String[] args){
        ActorSystem system = ActorSystem.create("
            testSystem");

        // Create a new Active Object (first arg for
        // the proxy, second for the servant).
        Printer helloWorld = TypedActor.get(system).
            typedActorOf(new TypedProps<PrinterImpl
            >(Printer.class, PrinterImpl.class));

        // Call asynchronously on the Active Object.
        Future<String> resultNB = helloWorld.
            printNonBlocking("Hello World!");

        // Call synchronously on the Active Object.
        Option<String> resultB = helloWorld.
            printBlocking("Hello World!");
    }
}
```

Listing 5. Implementation of both proxy's interface and servant.

Note that we once use an asynchronous function call, and once we use a synchronous function call. This calling differentiation is the only visible difference between active and passive objects.

5 Possibilities of mixing the AM and the AOM

All parts of the AM and the AOM may be built from their own base elements (actors and active objects). E.g., the mailbox of an actor can be an actor by itself and the activation queue might be an active object. By pushing this property to its limits, an actor could entirely be made of other actors, which in turn could be made of multiple actors (the same goes for the active objects). Provided the architecture and properties of each model is respected, mixing both models at the core level could be done. We discuss in the following paragraphs a few theoretical possibilities to mix both models in order to potentially reduce their respective disadvantages.

As seen previously, one of the major differences between the two models is the way messages are stored within an actor and an active object. The active objects use the activation queue which is a bounded buffer. In the case this buffer is full, further messages cannot be stored and will be lost or immediately return a future with a failure notice (depending on the implementation). The actors on the other hand have unbounded buffers to store incoming messages. Assuming an active object uses an actor to replace its activation queue, no method request from the proxy would be rejected. With the fairness of the actors in scheduling, all messages would be processed eventually. However, the AM uses asynchronous message passing with no restriction on message reception order [4]. Therefore messages would have to be sorted by time stamps. Note that this solution could be memory efficient on smaller devices as it would give the choice between creating a new active object and keeping the current number of active objects to handle more requests.

Another modification to the AOM would be to use an actor to replace the proxy, in combination with an actor for the activation queue. The proxy in the AOM formats the method requests specifically and leave a low margin for freedom. Even though the parameters captured in the method request might be any kind of objects, using the freedom of message semantics of the AM could offer more possibilities. The method requests captures the context of the invocation of a method. In other words it encapsulates into a message which function should be called on the servant with which parameters. Having clearly defined method requests enforces conformity of function calls but functions with numerous parameters are bad practice. With the freedom of message semantics from the AM, we could imagine creating a single context object and pass it to the scheduler. Note also that is only the proxy is converted to an actor, it would also solve the problem of the bounded activation queue. However it would be harder to implement since it requires managing priorities and delays of messages.

The AOM is not the only model that could benefit from mixing with the AM. Indeed the AM also has its weaknesses that could be improved by using concepts of the AOM. The general asynchronicity in the AM makes it difficult to track messages. This is even more evident upon tracking an expected answer when a task is completed by another actor. For this purpose the AOM uses futures. They are a convenient mechanism because they work as independent rendezvous points. In the AM, futures are not implemented, but messages sent upon completion and any type of object can be returned (the Akka framework implements futures in the AM). It could be an error message, a result, etc. However, in case the actor carrying out the task fails, there is no real way to handle this failure. With an actor playing the role of a future, timeouts can be used and it provides some sort of acknowledgement for a successful reception of the message. It does not break the UDP type of communication but could affect the performance. More messages mean more overhead and if tasks execute very quickly it could lead to congestion. But for longer tasks or tasks that require guarantees it could improve reliability and uniforms the delivery of results delivery.

As interesting as using futures in the AM, futures in the AOM could be implemented with actors. The main reason behind it would be that once the results (or errors) have been taken by the client, the future could automatically end itself. Indeed an actor has a specific life-cycle and upon destroying itself it could ease the work of garbage collection. Garbage collection is a costly process and easing it would result in better performances.

6 Conclusion

The Actor Model (AM) and Active Object Model (AOM) are often mistaken as being the same and hence would share the same properties. We have presented both models in terms of structure, communication protocols and concurrency applications. From these overviews emerged fundamental differences. On the first hand the AM offers more freedom of message semantics, less overhead due to the UDP style communication protocol and more execution control. This allows for programmers to focus on their experience and knowledge toward software design rather than concurrency technicalities. The AM takes advantage of the multi-agent system to perform computations. The AOM on the other hand follows the same design principles as the AM, but fails to achieve the same level of freedom. Indeed, the AOM constrains message types and works with acknowledgments, leading to performance slowdown. The AOM, however, offers mechanisms to probe the linked future for checking the current state of the execution.

By identifying the differences, we also highlight strengths and weaknesses of both models. The AOM turns out to be easier to implement in an already existing system as any passive object can be turned into an active object by wrapping it in a proxy. To build efficient systems, the AM is the best choice as it forces the whole system to be designed with concurrency in mind. No solution out-of-the-box exists. However, the AM and the AOM are a first step towards simplifying concurrent programming.

7 Outlook

Software agents are used in many fields such as ubiquitous computing and sensor networks. Both of these fields of research focus on the evolution of networks going from a small number of powerful entities to a vast amount of devices. These devices share issues such as data transfer and energy consumption as they run on batteries. Using the AM in such networks could offer advantages like the ability to migrate from one node to another, while conserving its current internal state; or allow a wide range of different devices to work together and share a common language. For sensor networks the AOM could offer a way to manage queries going on the network with the sink acting as a proxy. Research on agents running through sensor networks has already been done, but the topic is still new and widely unexplored. For ubiquitous computing the AM would probably suit better as we face a more disparate and quickly evolving type of network. The goal of ubiquitous computing lies in communicating with as many devices as possible. If all devices could talk to each other, it would result in far more efficient means of aggregating data and reduce the number of required transmissions to deliver this data. For many devices with limited computational power and embedded energy, it is beneficial to avoid transmission congestion. Moreover, the known addresses of the actors could be used to control the range and enforce access policies between the devices. Also with smaller devices real parallelism is not achieved. Instead, concurrency is simulated by event-based actions and asynchronous method call initiation and return of calls. TinyOS, for example, uses an AOM to deal with its concurrent parts. Research could be done to verify if using an AM could lead to higher energy saving / computational efficiency.

Another hot field of research focuses on saving energy. The hardware has evolved toward more efficient CPUs with improved photo-lithography processes and automatically varying their frequencies. However hardware-only solutions cannot achieve the goals of energy efficient computing alone. The software also plays a huge role in global energy consumption. An efficient program will need less time to execute and save energy compared to a program requiring more time for the same computations. Another fact in energy consumption shows that a CPU running on lower frequencies improves the energy consumption to computation performed ratio. However running at lower frequencies also means that programs executions will take more time. Distributing the workload on many cores or machines running on lower CPU frequencies could yield lower consumption and fast computations at the same time. It would be interesting to test whether it is energetically more efficient to use the AOM to distribute a workload on slower cores than running the whole process on a very fast CPU.

Finally, development carried in the recent years lead scalar CPUs to work in combination with vector CPUs. Vector CPUs are designed to handle massive numbers of concurrent threads supported by specific hardware architectures. Most vector CPUs are in fact GPUs (graphical processing units) as producing images for digital screens requires each pixel to be computed and displayed independently. The video game industry largely uses GPUs for real-time rasterization but many applications such as physics simulations or any program based on matrix operations require much parallel computations. In this context libraries like NVIDIA's CUDA[4] or OpenCL[5] have emerged. They aim at enabling parallel programming on heterogeneous systems. Most desktop and mobile computers embed a CPU and a GPU. By using the GPU to perform massively parallel computations, specific parts of specific programs can achieve up to 200 times the performances achieved only on scalar processors. The AM is a perfect way to convey and distribute work on multiple heterogeneous machines. It could integrate the ability to take advantage of GPUs power for vector operations and improve global performance. For this an actor could implement special messages that inform the

4. http://www.nvidia.com/object/cuda_home_new.html
5. https://www.khronos.org/opencl/

actor if a computation can be performed using a vector processor. There could also be specialized actors whose functions only run on vector processors. Such actors would only be spawned on compliant machines.

8 Concurrent processing of messages

Research on the AM and AOM mainly focuses on improving the performance by attempting to remove the biggest bottleneck: Processing only one message at a time. Multiple solutions have been found, we present three of them.

The first article [10] combines the AM with the Async-Finish Model (AFM). The AFM is a variant of the Fork-Join Model and can create lightweight tasks to manage synchronization constrains among these tasks. To allow the processing of multiple messages concurrently, a first message is processed. This message will be attached to a finish scope. All tasks that run during this scope will be asynchronously processed in the actor and all join when the finish scope's time is over. Futhermore the option to pause / resume the processing of an actor is included. This allows to pause the actor (only keeps on receiving messages), and when resumed will finish the current processing and then wait (as in the initial state of the actor). The AFM simplifies termination detection and allows arbitrary coordination between tasks.

The authors of the second work [11] implement Parallel Actor Monitor (PAM). A PAM is a scheduler "expressing a coordination strategy for the parallel execution of messages within a single actor" [11]. PAM schedules the processing of multiple messages to a pool of threads that are then allowed to access the internal state of a given actor. If the scheduler allows it, new messages can be dispatched before a first batch of selected messages has completed.

Hayduk et al. propose in [3] to add support for speculative concurrent execution in actors using Transactional Memory (TM). The block of statement triggered by each message is transformed into a transaction atomically executed. Multiple transactions can run concurrently and rollback if a synchronization constraint is not met. The transaction is then rolled back and restarted. Rollbacks allow to reinforce the robustness of the AM. With transactions running concurrently, scalability is improved. They implement this solution by adapting the Akka framework and show that the overhead of using transactions is hidden by the improved message processing throughput. This implementation extends to the AOM since Akka implements the AOM with actors.

References

[1] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[2] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[3] Y. Hayduk, A. Sobe, D. Harmanci, P. Marlier, and P. Felber, "Speculative concurrent processing with transactional memory in the actor model," in *Principles of Distributed Systems*. Springer, 2013, pp. 160–175.

[4] C. Hewitt, "Actor model of computation: scalable robust information systems," *arXiv preprint arXiv:1008.1459*, 2010.

[5] W. J. Wang, "The actor model," 2006, accessed: 2014-04-24. [Online]. Available: http://wcl.cs.rpi.edu/salsa/tutorial/salsa-1_1_0/node8.html

[6] R. G. Lavender and D. C. Schmidt, "Active object–an object behavioral pattern for concurrent programming," 1995.

[7] T. Gurock, "Active objects and futures: A concurrency abstraction implemented for c and .net," 2007.

[8] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[9] J. Field and C. A. Varela, "Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments," in *ACM SIGPLAN Notices*, vol. 40, no. 1. ACM, 2005, pp. 195–208.

[10] S. M. Imam and V. Sarkar, "Integrating task parallelism with actors," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 753–772.

[11] C. Scholliers, É. Tanter, and W. De Meuter, "Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model," *Science of Computer Programming*, vol. 80, pp. 52–64, 2014.