Towards Benchmarking Actor- and Agent-Based Programming Languages

Rafael C. Cardoso
FACIN–PUCRS
Porto Alegre - RS, Brazil
rafael.caue@acad.pucrs.br

Maicon R. Zatelli, Jomi F. Hübner
DAS–UFSC
Florianópolis - SC, Brazil
xsplyter@gmail.com
jomi.hubner@ufsc.br

Rafael H. Bordini
FACIN–PUCRS
Porto Alegre - RS, Brazil
r.bordini@pucrs.br

Abstract

Over the past few years there have been several advances in distributed systems, and more recently multi-core processors. Consequently, a natural need for concurrent and parallel programming languages arises. In this paper, we compare some aspects of two concurrency models, Actors and Agents, using benchmarks to evaluate: (i) the communication performance on a concurrency-heavy scenario; (ii) the performance in a scenario with the presence of bottleneck and synchronization problems; and (iii) the reactivity and fairness of the models. We chose Jason, 2APL, and GOAL as the agent-oriented programming languages and Erlang, Akka, and ActorFoundry as the actor-oriented programming languages. Overall, Erlang displayed the best performance of all languages used in this comparison, followed by ActorFoundry, Akka, Jason, 2APL, and GOAL, in this particular order.

Categories and Subject Descriptors I.2.11 [*Distributed Artificial Intelligence*]: Multiagent systems; D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages

General Terms Languages, Experimentation

Keywords benchmark, agents, actors, programming languages, concurrency

1. Introduction

With the recent advances in distributed systems and multi-core processors, there is naturally a greater need for concurrent and parallel programming languages. In this paper, we compare several languages that belong to two inherently concurrent programming models, the Actor- and the Agent-based models. Both models present promising approaches towards taking advantage of multi-core hardware that is widespread nowadays. The Actors model is by definition lighter and, consequently, more efficient, while typical agent architectures lead to a heavier programming paradigm, given that agents are capable of knowledge representation and reasoning to support autonomous intelligent behaviour.

Benchmarking programming languages is an important aspect of research in this area, particularly for those of a young paradigm such as agent-oriented programming. Computer hardware is constantly evolving and new languages are being developed in order to make good use of new hardware architectures. Thus, we hope that by comparing these two models we will be able to establish whether certain scenarios tend to be more appropriate for actors rather than agents and vice-versa, both in terms of naturalness of the paradigm for the development of those scenarios, and in terms of actual performance.

This work was inspired by The Computer Language Benchmarks Game (`http://shootout.alioth.debian.org/`), which provides performance evaluation for approximately twenty four programming languages on various benchmark problems. Although they evaluate the performance on computers with multiple cores, the problems and most of the languages are not appropriate for concurrent programming. A python script is available on their website that does repeated measurements of CPU time, elapsed time, resident memory usage, and CPU load for each core.

In this paper, we analyse the performance of six actor and agent platforms, using three concurrency benchmarks. The first one is the token-ring benchmark, similar to the `threadring` found in The Computer Language Benchmarks Game, where a token has to be passed around a ring of threads; this benchmark was used to compare the communication (message passing) aspect, common to both agent and actor languages. The second one is the chameneos-redux benchmark, also found in The Computer Language Bench-

AGERE!'13, October 27, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2602-5/13/10…$15.00.
http://dx.doi.org/10.1145/2541329.2541339

marks Game, where chamaneos creatures go to a meeting place and exchange colour with a meeting partner; we used it to measure both the performance in the presence of a bottleneck, and the fairness represented by the number of meetings that each "chameneos" had. The third and last benchmark is based on Fibonacci numbers, where clients requests a random Fibonacci number to be calculated by a server; in this scenario we compare the reactivity and internal concurrency of each language model and their platforms.

Unfortunately, not all the experiments finished successfully for 2APL and GOAL, as both were taking too long to finish each benchmark. Jason and the actor languages did not present this problem.

The remainder of this paper is organised as follows. The next section discusses related work. Section 3 gives an overview of each of the programming languages that we compare in this paper. Section 4 shows the description of each benchmark used in the experiments as well as all the results obtained, and the analysis of these results. Finally, we conclude the paper in Section 5.

2. Related Work

There are several benchmarks and evaluations of actor and agent programming languages in the literature, but to the best of our knowledge there is only a few that focus on evaluating both actor and agent models together. A study about several actor-oriented programming languages based on the Java Virtual Machine (JVM) is available in [20]. The study shows a qualitative comparison of actor properties in regards to their execution semantics, communication and synchronization abstractions, and a performance evaluation using three different benchmark problems. Two of the benchmarks are from the Computer Language Benchmarks Game website, the `threadring` and the `chameneos-redux` benchmarks, while the third is a simple implementation of a Fibonacci calculator. The JVM-based languages used in that comparison were: SALSA, Scala Actors, Kilim, Actor Architecture, JavAct, ActorFoundry, and Jetlang. Erlang uses its own virtual machine, nevertheless it was also included in the performance evaluation, as it was the most widely used actor language at the time. The performance results show that Erlang and Kilim had the best performance overall, although Kilim only focuses on providing basic message passing support. While we also have used these three benchmarks, we made several concurrency adaptations to the threadring and the chameneos-redux benchmarks, and had a different focus for the Fibonacci benchmark.

Similarly, in [6] we have a comparison of agent program similarity and time performance between three agent-oriented programming languages: 2APL, GOAL, and Jason. The benchmark used was a Fibonacci calculator, and the results show that the Jason agent was the fastest, followed by 2APL and then GOAL agents.

In [18] a feature model comprised of a collection of general concepts about programming languages appearing in the literature was presented. They validated those concepts against Erlang, Jason, and Java. The result was a feature model of actor, agent, and object programming languages that can be useful to guide programmers, helping them to decide which language is more appropriate for a particular application.

An approach to query caching (memoisation) for agent-oriented programming languages was proposed in [2]. A performance analysis suggests that memoisation can significantly improve the performance of the languages that were used: Jason, 2APL, and GOAL.

The simpAL agent-oriented programming language was proposed in [24], with the purpose of integrating autonomous and reactive behaviour; the authors claim that this is a problem not easily achievable with object-oriented or actor-oriented languages. The benchmark scenario is that of an agent reacting to a message by incrementing a counter and printing a message, comparing simpAL with Jason, Erlang, and ActorFoundry. When considering the experiments with no I/O (i.e., no printing), the results show simpAL as the fastest followed by Erlang, ActorFoundry, and then Jason. Otherwise Erlang is the fastest, followed by ActorFoundry, simpAL, and then Jason.

An implementation of Jason in Erlang, eJason, was presented in [15]. It makes use of the lightweight processes available in Erlang in order to allow the creation of a much higher number of agents then previously possible in Jason. The experiments involved a simple counter scenario, and a greeting scenario where greeting messages are printed on the console by each agent. The results show eJason going as high as 800,000 agents in the first scenario, and 300,000 for the second, while still maintaining a low execution time. Jason stayed in the limit of 10,000 agents in both scenarios. The downside of eJason is that only a small subset of Jason is currently implemented.

Finally, there is our own previous work [10, 11] in benchmarking communication for actor- and agent-based languages. We benchmarked Jason, Erlang, and Scala Actors, using the `threadring` scenario, from the Computer Language Benchmarks Game website, as a simple non-concurrent scenario, and two variations of it: a concurrent variation where 50 tokens are passed concurrently and a variation to test reactivity by adding a second type of token in the ring. Our results encouraged us to continue and extend that work, as Jason performed surprisingly close to the actor languages, even surpassing them in some cases. We also briefly benchmarked the agent-oriented programming language JACK Intelligent Agents [9] in our previous work, but the results obtained were not encouraging for benchmarks that had significant concurrency, since JACK uses mainly one thread, for security purposes. Therefore, JACK is not included in this paper, and instead we added two other agent-oriented programming languages, 2APL and GOAL, and another actor-oriented programming language, ActorFoundry. Besides the addition of these languages, we also used new benchmarks to cover different metrics and features.

3. Background

In this section, we give a brief overview of the programming languages that we chose for this comparison work. We only present basic information about each language, such as the runtime environment and communication features that might help understanding the experiments. However, we assume some familiarity of the reader with the actor and agent paradigms. In the actor model [1], an actor is a lightweight process that does not share state with other actors and communicates by asynchronous message passing through mailboxes. In the agent model [28], which is an extension of the actor model, an agent is a more complex entity. The agent languages we used are based on "practical reasoning" (i.e., logic-based reasoning about the best action to take at the given circumstances) [7, 8].

Jason [7] is a platform for the development of multi-agent systems based on the AgentSpeak language, initially conceived by Rao [23] and inspired by the Belief-Desire-Intention (BDI) architecture. The AgentSpeak language was later much extended in a series of publications by Bordini, Hübner, and colleagues, so as to make it suitable as a practical agent programming language. Communication in Jason is based on the speech-act theory [27], which considers messages as actions that affect the mental state of the agent that is receiving the message. Jason is implemented in Java, thus its programs run on a JVM, allowing support for user-defined "internal actions" that are programmed in Java and run internally within an agent rather than change the environment model, as normal actions do in agent programs.

2APL [14] is also a BDI-based language, implemented in Java, for the development of multi-agent systems. It is an extension of the original version of 3APL [17], which was focused on single agent programming. 2APL separates the multi-agent and individual agent concerns into two levels: the multi-agent level — providing constructs in terms of a set of individual agents and a set of environments; and the individual agent level — providing constructs to implement cognitive agents. Communication in 2APL is also based on speech acts and therefore considered actions.

GOAL [16] is not entirely based on the BDI model like the other two agent-oriented programming languages; it is influenced by the UNITY [13] language. In UNITY a set of actions executed in parallel constitutes a program, however whereas UNITY is based on variable assignment, GOAL uses more complex notions such as beliefs, goals, and agent capabilities. Message passing is based on mailboxes, each message that arrives is inserted as a fact in the receiver agent message base. GOAL's agent communication is also based on speech acts, represented by moods: indicative, declarative, and interrogative.

Erlang [4], acronym for Ericsson language where it was developed, is a functional language supported by an extensive library collection named OTP, originally an acronym for Open Telecom Platform before 2000 when Erlang became open source. The Erlang Run-Time System (ERTS) application is responsible for low-level operations, including the Erlang virtual machine called Bodgan's Erlang Abstract Machine (BEAM) [3]. Communication between processes — the concurrency model usually referred by Erlang users is the process model, but it corresponds directly to the actor model — is based on asynchronous message passing; if a message matching the pattern is found in the queue, it is processed and its variables instantiated before the expressions in the body are evaluated. Functions are also defined by pattern matching and expressions as usual in functional languages. For further details, we refer the interested reader to [12, 21].

Akka can be used either as a library for Scala or directly with Java. In this paper we chose to use the Akka library for Scala as it provides a syntax that is similar with other common actor programming languages. Scala is considered a multi-paradigm language, as it combines features of object-oriented and functional programming languages. Its programs run on a JVM, so it has direct integration with Java, allowing the use of existing Java code within Scala systems [22]. Many libraries have been developed for it, such as Scala Actors (currently to be deprecated) and Akka: two libraries that provide concurrent programming based on actors for Scala programming. It is important to note that the actors used in the scenarios of this paper are event-based actors (default in Akka), and not thread-based actors. For a more in-depth reading of Scala, we suggest [26] and, more specifically for Akka, [29].

ActorFoundry [5] is a JVM-based framework for actor-oriented programming that allows actor programs to be written with a familiar Java syntax. It performs internally a continuation-passing style transformation using the bytecode post-processor from the Kilim [25] actor programming language, and maps M actors to N native threads where $M \gg N$. As in the previous actor-oriented programming languages, actors in ActorFoundry communicate mainly by asynchronous message passing.

4. Experiment Results

The machine used to run the experiments was an Intel Xeon Six-Core E5645 CPU @ 2.40GHz (6 physical, 12 logical cores with HyperThreadring), with 12GB of DDR3 1333 MHz RAM, 1TB hard disk drive, running Ubuntu 12.10 64 bits; the versions of the languages used were Jason 1.3.10, 2APL 2010-11-16, GOAL 20130516v5876, Erlang R16B01 erts 5.10.2, Akka 2.1.4 for Scala 2.10.2, and ActorFoundry 1.0; the additional software used were Java OpenJDK 64-Bit Server VM, Java 1.7.0_21, and Python 2.7.3. We used

Java 6 for the experiments in ActorFoundry as its the only Java version currently supported, and used Java 7 for all the other languages that required Java. The benchmarks described in the following sections focus on the message passing aspect of communication, testing the support for asynchronous message passing, as well as the concurrency, fairness, and reactivity present in each platform; features that are essential for both actor- and agent-based languages.

Each program is run as a child-process of a Python script using the popen function to create a pipe between the script and the program. We used this script for the token-ring and the chameneos-redux benchmarks, and took three measurements with it: CPU load for each core, elapsed time, and resident memory. The script measures core usage through the GTop library on Unix systems, taking the CPU-idle and CPU-total values before forking the child-process and after it exits; the percentage represents the time that a core was not-idle. For measurements of elapsed time, it uses `time.time()` to get the time before forking and after exiting. Resident memory is measured by sampling GLIBTOP_PROC_MEM_RESIDENT for the program every 0.2 seconds. All the values shown below were collected through five repeated measurements of each program with each configuration; the results represent the turn with lowest (best) value of elapsed time among the five different runs. We did not use these performance metrics for the Fibonacci benchmark, as performance is not the focus in that scenario. We also measured source code size for all three benchmarks; source code size was measured by removing the comments then applying GZip compression (using the `--fast` parameter). The results of source code size can be found at the end of this section, after the analysis of the Fibonacci benchmark.

4.1 Token Ring

This scenario is similar to one in our previous work on benchmarking actor and agent languages [11], except that this time there is a larger focus on concurrency rather than just a variation on number of token passes. In this version of the benchmark, T tokens are available to be passed concurrently N times through a ring of "workers" (i.e., agents, processes, or actors, depending on the language). Each program in this scenario should:

- create 500 workers, named 1 to 500, and each linked to its successor;
- worker 500 should be linked back to worker 1, forming an unbroken ring;
- initially pass a token to the worker assigned by the distribution equation below;
- each worker passes the token to its neighbouring worker (i.e., the worker it is linked to);
- the program halts when all T tokens have been passed, between any two workers, N times each.

At the start of a run the tokens are distributed through the ring using the following equation:

$$W = I * (WT/T)$$

where W is the worker which will receive the current token, I is the number of the current token being assigned, WT is the total number of workers, and T is the total number of tokens. Each of these T tokens have to be passed N times, and because neither agents nor actors share state, a counter worker is needed for counting the tokens that have finished: this is necessary because in order for the Python bencher script to run repeated measurements it needs the programs to halt. We ran experiments for this scenario with N fixed at 500,000 and five different configurations for T, starting from 1 token (not concurrent) and adding 250 tokens with each configuration, arriving at a total of 1,000 tokens (2 tokens per worker) in the last configuration. This was done in order to check how well the languages handle an increasing concurrency factor, and its impact on multi-core processors. In summary, we have the following configurations: $T = 1$; $T = 250$; $T = 500$; $T = 750$; and $T = 1,000$.

The results for this benchmark can be seen in the following graphs[1]. Figure 1 presents the measurements of elapsed time in seconds based on the values in Table 1. Figure 2 shows the memory results, and in Figure 3 we have the CPU load results.

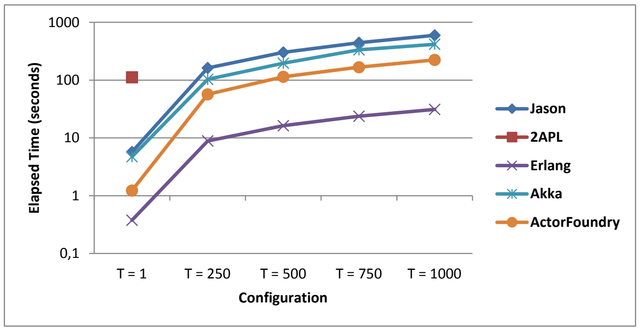

Figure 1. Elapsed time for the token-ring benchmark.

As we previously stated in the introduction, it was not possible to complete all experiments when using 2APL and GOAL. 2APL could not run any configuration besides the first one, always terminating with a Java exception indicating that the garbage collector overhead limit had been exceeded. We made several trials, changing some parameters of the JVM, for example increasing the heap size, but to no avail. By using the flag `-XX:-UseGCOverheadLimit` we managed to ignore the warning generated by the exception, but the benchmark execution would carry on for several hours (more than 8 hours) without concluding, and Ubuntu would display several overheating warnings for the cores.

[1] The elapsed time graphs presented in this paper are in logarithmic scale of base 10, and the memory usage graphs are in logarithmic scale of base 2.

Tokens	1	250	500	750	1,000
Jason	5.724	162.476	302.386	444.015	597.024
2APL	112.078	—	—	—	—
Erlang	0.377	8.888	16.273	23.768	31.156
Akka	4.727	103.21	196.375	333.929	418.273
ActorFoundry	1.229	56.749	114.221	167.199	223.918

Table 1. Elapsed time in seconds for the token-ring benchmark.

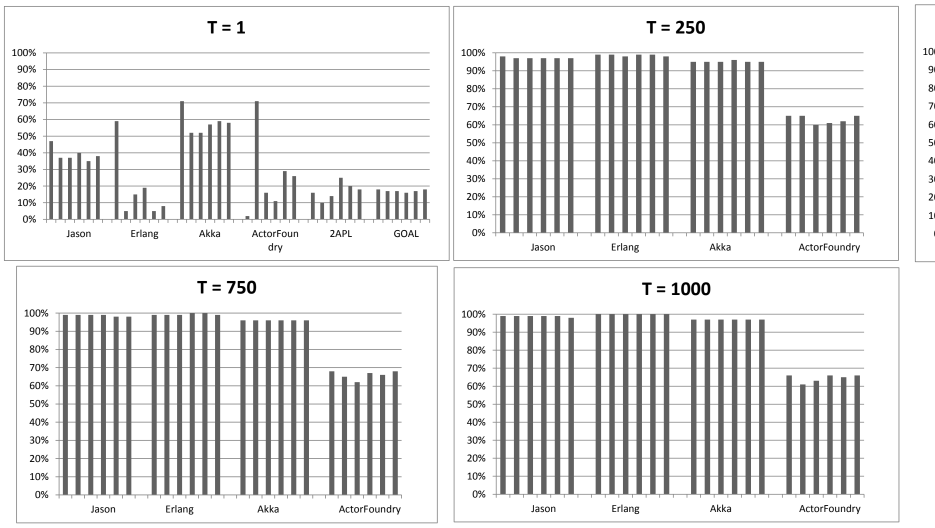

Figure 3. Core load for the token-ring benchmark.

Using 500 workers caused a heavy impact on performance in this benchmark for GOAL, to the point where it would run for more than 4 hours in the first configuration, with only one token. Thus we were not able to run the experiments using GOAL with the same configurations used in the other languages. Just to present some values, it took 328 seconds to finish the experiment with only 50 workers, 1 token, and 50,000 token passes, which is a much lower configuration than those that we used for the other languages. In regards to memory and core load for this lower configuration, GOAL used 250MB and an average of 17% respectively.

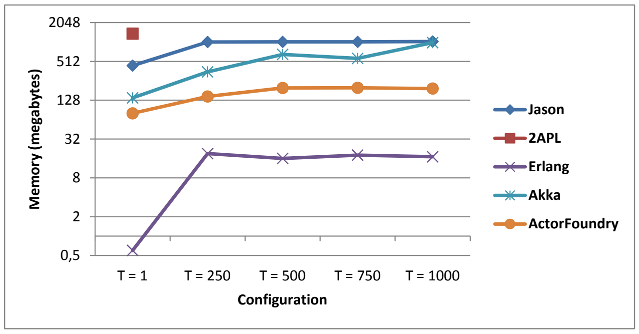

Figure 2. Memory for the token-ring benchmark.

That garbage collector exception found in 2APL occurs if more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, then an `OutOfMemoryError` will be thrown. This feature is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small. Although we cannot attest to what specifically caused this behaviour, the nature of the experiments indicate that the problem is related to the use of multiple tokens, i.e. increasing the concurrency factor of the scenario. As each token has to be passed N times, we also have a much larger pool of messages to go through, totalling $T*N$ messages, which should also be considered as a potential cause for this problem.

With GOAL, the performance bottleneck appears be the number of workers, as it takes a long time just to start the workers and form the ring. GOAL had the highest elapsed time between all of the languages, even when using a lower configuration and only one token. Both 2APL and GOAL use, besides the JVM, a Prolog interpreter that could have caused a heavy impact on performance. 2APL uses the JIProlog interpreter and GOAL uses the SWI-Prolog interpreter.

The elapsed time results for the token-ring benchmark show Erlang as the one with the best results (i.e., lower elapsed time), followed distantly by ActorFoundry, and then Akka and Jason close together. All languages had the expected peak in elapsed time when moving from the first configuration (1 token) to the second (250 tokens), with a close to linear increase for the rest of the configurations. We can easily separate Erlang into one group, and the other three languages into another one which are orders of magnitude slower for this benchmark. Interestingly, these three languages use a JVM, while Erlang has its own VM. This suggests that the VM could be responsible for the performance gap, as it is responsible for the garbage collector, thread optimisation, memory usage, core distribution, and many other performance factors.

Considering the memory results, we have a much smaller peak when moving from the first configuration to the second. It may seem that there is a big peak for Erlang, but moving from 0.5MB to 28MB is not really a lot. Increasing the concurrency factor, i.e. the number of tokens, did not seem to impact the memory used by each language except for that initial peak. Although this is a somewhat expected behaviour for the actor languages, the agents are much heavier entities and with more tokens passing through the ring it would mean that more intentions are generated during each agents reasoning cycle, and therefore should have an increase in memory usage as the number of tokens that are passed concurrently increases. A possible solution involves changing the heap size used by the JVM, which may improve performance as the garbage collector would waste less time emptying the heap, and less time would be spent in paging.

In regards to core usage, both ActorFoundry and Erlang had optimal use of the cores for the first configuration; as only one token is passed, there is no concurrency and therefore mostly one core should be used. When moving to concurrency-heavy configurations, ActorFoundry did not manage to keep up with the other languages, presenting only an average usage of the cores while Jason, Erlang, and Akka used the cores to their maximum. Although ActorFoundry's performance on elapsed time was not bad by any means, the core usage indicates that it could do even better if it made more appropriate use of the available cores. Both 2APL and GOAL had low core usage across all six cores in the configurations that they were able to finish; this could be another determining factor for their low performance.

4.2 Chameneos Redux

This benchmark is also from The Computer Language Benchmarks Game website, which is an adaptation of the full version found in [19]. In our version of this benchmark we varied the number of chameneos creatures instead of the number of meetings. The general idea of the benchmark is that given a population of C chameneos creatures, each creature lives lonely in the forest but at certain times it goes to a meeting place where it meets one other creature, and both mutate before returning to the forest. Each program in this scenario should:

- create C differently coloured (blue, red, or yellow), differently named, chameneos creatures;
- each creature will repeatedly go to the meeting place and meet, or wait to meet, another creature;
- both creatures will change colours to the complement of the colour of the chameneos that they just met;
- write all of the complements between the three colours (blue, red, and yellow);
- write the initial colours of all chameneos creatures;
- after N meetings, for each creature write the number of creatures it met and spell out the number of times that it met a creature with the same name (should be zero);
- the program halts after writing the sum of the number of meetings that each creature had (should be $2 * N$).

A broker worker had to be created in order to represent the meeting place and simulate the meetings. We ran experiments for this scenario with N fixed at 500,000 meetings and three different configurations for the number of chameneos creatures (C), in order to test how well the languages handle an increasing concurrency factor when dealing with a small bottleneck, such as the Broker (i.e., the meeting place), in this scenario. The configurations used are: $C = 5$; $C = 50$; and $C = 500$.

The results for this benchmark are shown in the following graphs. Figure 4 presents the measurements of elapsed time in seconds based on the values in Table 2. Figure 7 shows the memory results, and in Figure 5 we have the CPU load results for each of the six cores. Besides these measurements, another important aspect to report in this scenario is the fairness of the workload distribution between workers for each language, which in this case is represented by the number of meetings that each chameneos (i.e., worker) had at the end of a run. To represent this data we make use of boxplot graphs[2] in Figure 6, and present the standard deviations in Table 3.

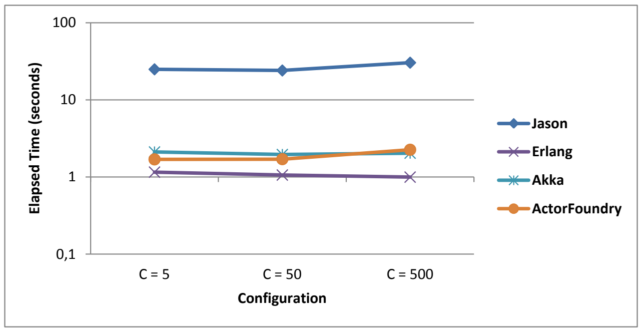

Figure 4. Elapsed time for the chameneos-redux benchmark.

[2] In a boxplot, the bottom of the box represents the first quartile, the top of the box represents the third quartile, the band inside the box represents the median, and the bottom and top whiskers represent, in the way we used here, the minimum and maximum of all the data, respectively.

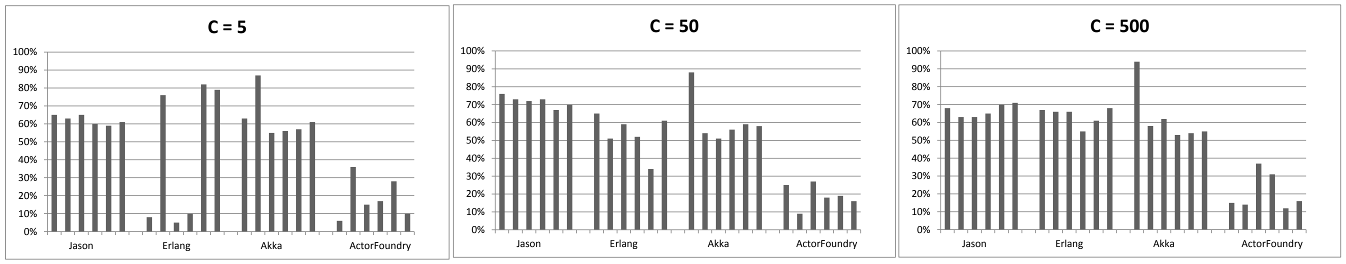

Figure 5. Core load for the chameneos-redux benchmark.

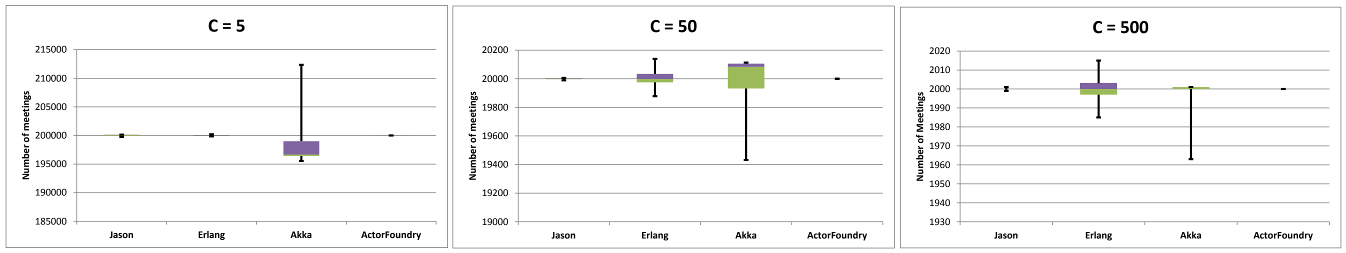

Figure 6. Boxplot for the numbers of meetings of each worker in the chameneos-redux benchmark.

The same behaviour as in the previous benchmark appeared again for 2APL and GOAL, as both were unable to run the same experiment configurations as the other languages. Their programs would either end in an exception, or keep running for too long a period of time. In 2APL we obtained the same exception as before, "GC overhead limit exceeded", and had to lower the configurations in order to obtain the following results: for 5 chameneos and 50,000 meetings, the result was 33 seconds of elapsed time, 1040MB of memory used, and an average 50% of core load; for 50 chameneos and 50,000 meetings, the result was 33 seconds of elapsed time, 1552MB of memory used, and an average 49% of core load; and for 500 chameneos and 50,000 meetings the results was 56 seconds of elapsed time, 1716MB of memory used, and an average 40% of core load.

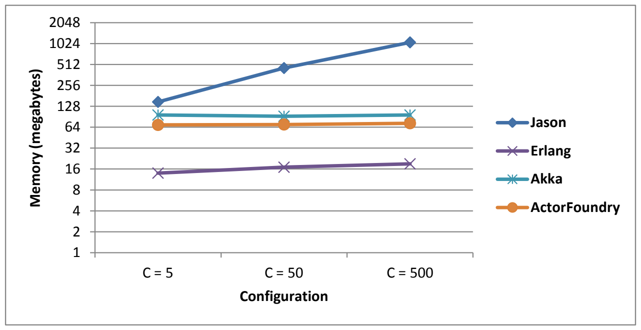

Figure 7. Memory for the chameneos-redux benchmark.

In GOAL, it was even more difficult to choose the values for the lower configuration because, as in the previous benchmark, changing the number of workers proved to cause a huge impact on performance, so we had to lower considerably the number of meetings in order to increase the number of chameneos. The configurations and results are as follows: for 5 chameneos and 50,000 meetings the result was 310 seconds of elapsed time, 171MB of memory used, and an average 17% of core load; for 50 chameneos and 5,000 meetings the result was 195 seconds of elapsed time, 268MB of memory used, and an average 17% of core load; and for 500 chameneos and 500 meetings the results was 632 seconds of elapsed time, 1252MB of memory used, and an average 16% of core load.

Chameneos	5	50	500
Jason	24.882	24.034	30.326
Erlang	1.157	1.061	0.997
Akka	2.11	1.956	2.038
ActorFoundry	1.693	1.705	2.257

Table 2. Elapsed time in seconds for the chameneos-redux benchmark.

Chameneos	5	50	500
Jason	167	5	0.347
Erlang	144	50	5.059
Akka	7,021	154	3.637
ActorFoundry	3	0	0

Table 3. Standard deviations for chameneos-redux.

The analysis of the observed behaviour of 2APL and GOAL in the token-ring benchmark also applies here. We can see that this time 2APL used the available cores a little better, albeit still low when compared to the other languages. GOAL stayed below the 20% mark for core-load average, as in the previous benchmark, indicating that it has high amounts of CPU idle time. Regarding memory usage, 2APL seems to use a lot of memory even for simpler configurations; this memory problem may be the cause of the GC overhead exceptions that we encountered when trying to run heavier configurations. As for the memory usage in GOAL, it had similar results to Jason, even though the number of meetings were a lot lower per configuration for the same number of workers (chameneos). Increasing the number of workers makes a higher impact on memory usage than increasing the number of meetings for the agent programming languages, which is expected as each agent has a complex internal architecture.

Our initial idea for this benchmark was to increase the number of chameneos up to a point where the elapsed time would start to increase as well with each configuration, because of the bottleneck that the broker (meeting place) causes. We were not able to increase that value further as 2APL and GOAL had problems even with the normal configurations. With only three configurations and a max of 500 chameneos, there was almost no increase in time for any of the other languages in the graph. Jason did not present similar results as in the previous benchmark, that is, its performance was not as close to the actor languages that use JVM. This suggests that, performance-wise, this benchmark may be a more appropriate scenario for actor languages instead of agent languages.

Jason also does not scale very well in regards to memory in this benchmark, although in this case it was to be expected, as generally creating more agents will result in an increase on memory usage, as we previously mentioned. The actor languages managed to stay constant in their memory usage, although they may have executed too fast to even make use of the garbage collector.

ActorFoundry had an even lower use of the available cores, across all three configurations, than in the previous benchmark. Jason and Akka had an even distribution of core load, while Erlang started using only three cores for the first configuration and balanced out in the other two configurations. Except for Jason, we cannot attest how core usage may have influenced the performance of elapsed time, as the actor languages may have executed too fast for our current configurations. However, the core load results suggest the presence of a small bottleneck, as it appears that in each language there were similar core load results across all three configurations.

Regarding the boxplot graphs and the standard deviation results, we can observe optimal performance in all configurations by ActorFoundry, where basically almost every chameneos had the same number of meetings, followed by an excellent display by Jason, that had only a small number of chameneos with more/less meetings than the other ones. Erlang had good performance for the first configuration, but an average performance for the other two, indicating that performance appears to get worse as the number of chameneos increase. Akka had the worst performance in regards to fairness (i.e., how many other creatures each chameneos met). It had an asymmetrical distribution of the number of meetings, which varied either with a few very high values (first configuration) or very low values (in the other two configurations).

4.3 Fibonacci

In this scenario we analyse the reactivity aspect of communication, using an implementation of a Fibonacci number calculator as a server. The fibonacci worker that acts as a server takes requests from C clients, with each client making R requests (one at a time for each client) to solve an nth Fibonacci number; each such n is as a random number from 0 to M, where M is the maximum value that can be calculated for each language, before it takes too long too wait (i.e., around 10 minutes). Each program in this scenario should:

- create the Fibonacci ”server”;
- create C clients;
- create a manager worker, responsible for writing the results in a file;
- from the total of R requests made by each client, send to the server one pending request to calculate the nth Fibonacci number, with n a random value between 0 and M;
- each client sends the time it waited for the answer for that request to the manager;
- the program halts when all R requests from each client have been completed.

The implementation should be as simple as possible, preferably using a regular recursive solution, as we are not trying to measure how quick these languages can calculate a Fibonacci number, but rather how quick they can react when they receive requests for calculating a small Fibonacci number among larger requests that will take a lot of time. That is, we want to measure the *responsiveness* of the Fibonacci server to all the clients' requests. Therefore, the clients should start a timer just before they send a request to the server, and stop the timer as soon as they receive that particular result. After receiving the result, they should send the time it took to get the answer to the manager, who will then write it in a file along with the Fibonacci number calculated in that particular request. To summarise, the behaviour we expect as ideal in this benchmark is to quickly solve small Fibonacci numbers, so as to not let larger requests slow down the calculation of all the other smaller (hence quicker to calculate) requests.

The results shown in Figure 8 present the different behaviours that we obtained using different approaches with each language. For these experiments, R (total number of requests) was set to 50 and C (total number of clients) to

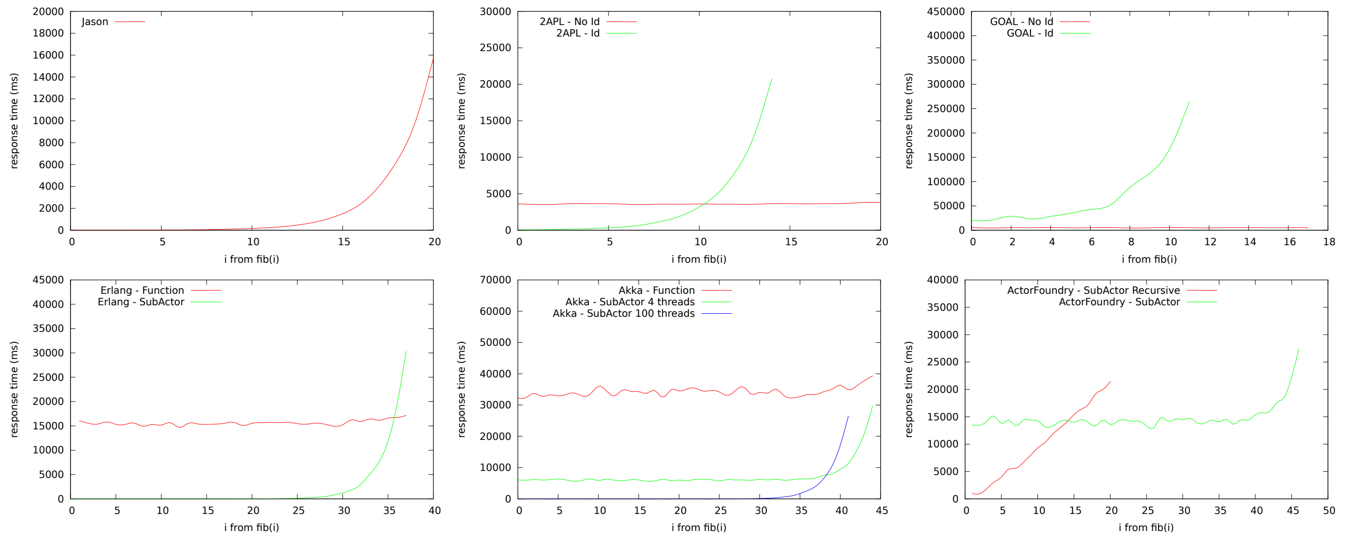

Figure 8. Results for each language in the Fibonacci benchmark.

100, resulting in 5,000 points for the graphs in Figure 8. For GOAL we had to lower the number of points to 1,000, setting R to 10 instead of 50. Below we will discuss how we tried to achieve the behaviour that we expected as ideal for each language and the results from the graphs.

In Jason there is a natural solution to achieve the expected behaviour for this benchmark, while all other languages had multiple solutions, none of which appears to be as natural as in Jason. Besides the concurrency between agents, Jason also has internal concurrency in each agent through the notion of intentions. Because of this internal concurrency present in Jason, each request made by a client to calculate a Fibonacci number will generate a new intention on the server. These intentions will then be executed concurrently, thus ensuring that small Fibonacci numbers are calculated fast and before larger Fibonacci numbers. That is an important feature of Jason, and one that is useful in scenarios where reactivity is needed. Although other languages also have some degree of reactivity, they often have to be simulated or worked around, and as such they do not appear as natural.

To simulate this behaviour in the other agent languages, namely 2APL and GOAL, a special version using identifiers (IDs) for each request had to be implemented, so that goals could be pursued concurrently. We had to add an ID for each new request, in order to identify which one was being calculated, otherwise the server would not know who to respond to if it had, for example, two requests (goals) for `fib(10)`. In 2APL, we were able to do that without much effort. Besides the increase in the code length, there was also a decrease in performance, but 2APL managed to achieve the type of behaviour that we expected for this benchmark. In GOAL, however, this approach did not result in the behaviour that we hoped for, but it was somewhat closer to it, even though performance was significantly compromised in the process.

In Erlang, using recursive functions resulted in exactly the opposite behaviour we were looking for: large Fibonacci number requests were holding up the solution of the smaller ones. The version with subactors — despite this being an obvious approach for the actor paradigm, it does not appear as natural as the one in Jason, as actors do not have internal concurrency — leads to the expected behaviour. In that version, the server creates a subactor for each request that arrives, and the subactor becomes responsible for solving that particular request.

With Akka, using recursive functions resulted in the same behaviour as Erlang with functions but, unexpectedly, using subactors in Akka did not initially result in the same behaviour as subactors in Erlang. We then altered the configuration for the number of threads used, changing it to the maximum number of subactors possible at any given time (which in our scenario is 100: 1 subactor per request, 100 clients, each client makes one request at a time), we then managed to achieve the expected behaviour. Such configurations are found in the *application.conf* file; we changed *parallelism-factor* and *parallelism-max* to 100 for the default Akka dispatcher, *fork-join-executor*.

We used two different subactor versions for ActorFoundry, one where a subactor is created for every request from a client (same as Erlang and Akka subactor versions), and another where a subactor is created for every request (from a client, server, or another subactor). The latter was adapted from the source codes available with the ActorFoundry distribution, and we refer it from now on as the recursive subactor version. Despite its unexpected linear growth, as shown in the graph, this recursive subactor version had poor performance in this benchmark because a very high number of subactors had to be created. When dealing with a lower number of requests, this recursive subactor version can be very efficient. The version with simple subactors had a similar behaviour to the one presented by Akka when using 4 threads, but as far as we know ActorFoundry has no similar setting for the number of threads, and thus could not achieve that ideal behaviour that we were looking for.

The values of M (maximum Fibonacci number to be calculated) for each language were as follows: in Jason, 20; in 2APL with IDs, 13; in 2APL with no IDs, 20; in GOAL with IDs, 12; in GOAL with no IDs, 18; in Erlang with subactors or functions, 38; in Akka with subactors and 100 threads, 42; in Akka with functions or subactors and 4 threads, 45; in ActorFoundry with subactors, 46; and in ActorFoundry with recursive subactors 20.

Finally, we show the results for source code size in Table 4. Although there is in general some controversy about whether any metrics other than source lines of code (SLOC) are good code-complexity metrics, we decided to measure the source code size by compressing it with `gzip`. The authors are not expertly familiar with the syntax and the best coding practices for all the languages used in this paper, so perhaps compression may help to alleviate some of these factors (e.g., ignoring line breaks that expert programmers of a particular language would normally do). The results for the alternative implementations (i.e., those that did not achieve the expected behaviour) for the Fibonacci benchmark are as follows: 2APL with no IDs, 568 bytes; GOAL with no IDs, 797 bytes; Erlang with functions, 710 bytes; Akka with functions, 856 bytes; ActorFoundry with recursive subactors, 1157 bytes.

Benchmark	Token	Chameneos	Fibonacci
Jason	415	1074	501
2APL	1034	1711	656
GOAL	1038	1687	929
Erlang	400	726	710
Akka	423	1221	856
ActorFoundry	636	1266	1043

Table 4. Compressed source code size for the benchmarks, in bytes.

In regards to the results for source code size, overall Jason and Erlang presented the lowest (best) size among all the languages. In the Fibonacci benchmark, Jason had the shortest source code, while Erlang did best in the chameneos-redux benchmark; both languages has similarly good results for the token-ring benchmark. Overall 2APL and GOAL had the worst results while Akka and ActorFoudry stayed in between.

The interested reader can check the source code for all benchmarks and languages used in this work at `https://github.com/rafaelcaue`.

5. Conclusion

In this paper, we compared three agent-oriented programming languages, Jason, 2APL, and GOAL, along with three actor-oriented programming languages, Erlang, Akka, and ActorFoundry, using three distinct benchmarks: the token-ring benchmark, with a non-concurrent configuration and multiple concurrency-heavy configurations for measuring communication performance; the chameneos-redux benchmark, a concurrency scenario for measuring fairness and performance in the presence of a bottleneck; and a Fibonacci benchmark, a scenario focused on measuring reactivity. Although the comparison of different programming models is a hard task, even for related models with many similarities such as the actor and agent models, we were able to show that Jason can follow closely the performance of the JVM-based actor languages, even surpassing them in some of the comparison criteria, despite belonging to a "heavier" paradigm.

The criteria mentioned above relate to some of the features found in programming languages. Although we only focused on those features in this paper, there are many others that could be taken into consideration when evaluating a programming language, such as: security, software quality, exception handling, library support, expressiveness, etc.

Future work includes developing new benchmarks for measuring more specific metrics of actor and agent languages, and including even more languages and comparison criteria in the experiments. We also plan to compare more qualitative aspects of such languages, with specific benchmarks for that.

The agent languages we compared so far demonstrated very different behaviours where performance was considered, which is to be expected since these languages are relatively new when compared to their actor counterparts. A more in-depth comparison focusing only on the agent languages would make it easier to focus on metrics that are specific to the agent model, and that may lead to more complete experiment sets for those languages. In order to support the work on benchmarking agent languages, we have developed a website (`http://www.inf.pucrs.br/maop.benchmarking/`) to serve as a repository of benchmarks specifically designed for comparison of agent programming languages. Benchmarking agent programming languages can sometimes lead to performance improvements in the respective platforms, and is an important step towards mainstreaming of the agent programming paradigm.

Acknowledgments

We are grateful for the support given by CAPES and by CNPq (grant numbers 140261/2013-3, 306301/2012-1, and 308095/2012-0).

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] N. Alechina, T. Behrens, K. Hindriks, and B. Logan. Query Caching in Agent Programming Languages. In *Proceedings of ProMAS-2012, held with AAMAS-2012*, pages 117–131, Valencia, Spain, June 2012.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, Sept. 2010.

[5] M. Astley. The Actor Foundry: A Java-based Actor Programming Environment. Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998.

[6] T. M. Behrens, K. Hindriks, J. Hübner, and M. Dastani. Putting APL Platforms to the Test: Agent Similarity and Execution Performance. Technical Report IfI-10-09, Clausthal University of Technology, 2010.

[7] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[8] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, 2009.

[9] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java. AgentLink News, Issue 2, 1999.

[10] R. C. Cardoso, J. F. Hübner, and R. H. Bordini. Benchmarking Communication in Agent- and Actor-Based Languages (Extended Abstract). In *Proceedings of the AAMAS '13*, pages 1267–1268, Saint Paul, Minnesota, USA, 2013.

[11] R. C. Cardoso, J. F. Hübner, and R. H. Bordini. Benchmarking Communication in Agent- and Actor-Based Languages. In *Proceedings of the EMAS '13, held with AAMAS-2013*, pages 81–96, Saint Paul, Minnesota, USA, 2013.

[12] F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[13] K. M. Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[14] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008.

[15] Á. F. Díaz, C. B. Earle, and L.-A. Fredlund. eJason: an implementation of Jason in Erlang. In *Proceedings of ProMAS-2012, held with AAMAS-2012*, pages 7–22, Valencia, Spain, June 2012.

[16] K. V. Hindriks. Programming rational agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer, June 2009.

[17] K. V. Hindriks, F. S. De Boer, W. Van Der Hoek, and J.-J. Ch. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, Nov. 1999.

[18] H. Jordan, G. Botterweck, M.-P. Huget, and R. Collier. A feature model of actor, agent, and object programming languages. In *Proceedings of AGERE!'11*, pages 147–158, New York, NY, USA, 2011.

[19] C. Kaiser and J.-F. Pradat-Peyre. Chameneos, a concurrency game for Java, Ada and others. In *Computer Systems and Applications. ACS/IEEE International.*, 2003.

[20] R. K. Karmani, A. Shali, and G. hind. Actor frameworks for the JVM platform: a comparative analysis. In *PPPJ '09*, pages 11–20, New York, NY, USA, 2009.

[21] M. Logan, E. Merritt, and R. Carlsson. *Erlang and OTP in Action*. Manning, Nov. 2010.

[22] M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[23] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proc. 7th MAAMAW*, pages 42–55, 1996.

[24] A. Ricci and A. Santi. Programming abstractions for integrating autonomous and reactive behaviors: an agent-oriented approach. In *Proceedings of AGERE! '12*, pages 83–94, New York, NY, USA, 2012.

[25] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP '08*, pages 104–128, Berlin, Heidelberg, 2008.

[26] J. Suereth. *Scala In Depth*. Manning Publications Co., 2012.

[27] R. Vieira, Á. F. Moreira, M. Wooldridge, and R. H. Bordini. On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language. *J. Artif. Intell. Res. (JAIR)*, 29:221–267, 2007.

[28] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10:115–152, 1995.

[29] D. Wyatt. *Akka Concurrency*. Artima Incorporation, USA, 2013.