Interfacing Clojure with Pogamut 3 platform

Marcel Gołuński[i], Piotr Wąsiewicz[ii]
Institute of Electronic Systems, Warsaw University of Technology

ABSTRACT

In this paper we present an interface between Pogamut 3 platform and Clojure programming language. Clojure is a state of the art functional language with roots in Lisp. Pogamut 3 is a framework that simplifies creation of embodied agents. Our goal was to introduce Clojure code in our agents logic. Simple emergent behavior of a group of agents was implemented using Clojure code. Performance of execution of Clojure code called from Pogamut platform was measured.

Keywords: bots, agents, Pogamut 3, Java, Clojure, functional programming, Unreal Tournament 2004, benchmark

1. INTRODUCTION

Bot logic can be very complicated and can require a lot of resources. Concurrent programming is a solution that can speed up complex computations. Clojure which simplifies concurrent programming might deal with that level of complexity. In our previous work we have explored the problem of tactical assessment in a squad of intelligent bots [1]. That work covered problems like coordination and communication in a squad of bots, tactical assessment and coordinated movement. Bot logic was implemented plainly in Java using Pogamut 2 platform. After struggling with complexity in the plain Java implementation we have decided that a new functional approach targeted on concurrent programming can be helpful. Therefore we have set off to solve a problem of integrating Clojure code with a new version of testing platform Pogamut 3. Interface design should be flexible and allow for using both Clojure with its functional nature and the imperative object oriented code of Java. One of the main criterion for the Java - Clojure interface was to reuse the existing bot logic code.

2. POGAMUT AND CLOJURE

2.1 Pogamut 3 platform

Pogamut as the authors describe it is a "Java middleware that enables controlling virtual agents in multiple environments provided by game engines" [2]. In other words Pogamut is a Java framework providing an API to control Agents (Bots) in virtual environments and in some cases the game server (virtual environment) itself. Currently three major game engines are supported: Unreal Tournament 2004 (UT2004), Unreal Development Kit (UDK) and DEFCON shown in figure 2. Framework simplifies agent creation. Many complicated actions that bots take in the environment (like pathfinding) can be performed by simple commands. Pogamut provides a plug-in for NetBeans ide which simplifies debugging of the agents. Developer can control the number of agents in the environment, easily remove bots or check bot logs. Pogamut 3 is re-engineered, fully mavenized version of pogamut 2 (which we have used in previous work). It is much more advanced, and offers more features. It's distributed under GNU GPL v3 license. Pogamut was used as a test platform in many research projects including: Genetic Bots [3], StorySpeak [4], Episodic memory for virtual agent [5] and Emohawk [6]. We have used Unreal Tournament 2004 environment as a virtual environment for Pogamut 3. The simplified architecture is depicted on the diagram shown in figure 1.

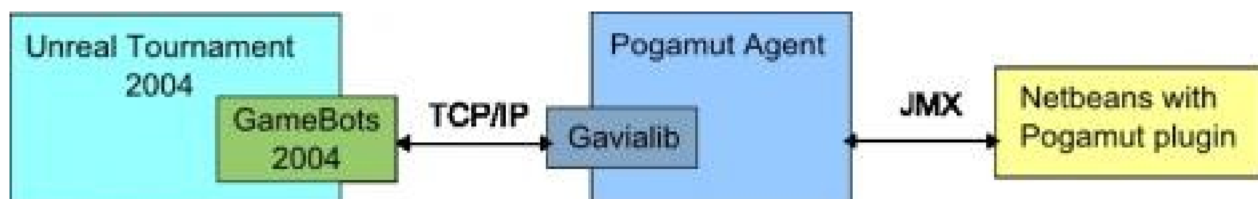

Figure 1. Pogamut 3 architecture (with UT2004) (source: Pogamut documentation).

Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2012,
edited by Ryszard S. Romaniuk, Proc. of SPIE Vol. 8454, 845429 · © 2012 SPIE
CCC code: 0277-786/12/$18 · doi: 10.1117/12.2000228

Unreal Tournament 2004[1] UDK - Unreal Engine 3[2] DEFCON[3]

Figure 2. Game engines supported by Pogamut 3.

2.2 Clojure

Clojure [7] is a relatively young language. It was created by Rich Hickey in 2007. It's one of the youngest languages on the market that has an active community, new concepts and interesting prospects. Compared to popular Scala [8] Clojure is 3 years younger.

Rich Hickey's language is a modern kind of lisp dialect designed for concurrency (Lisp is one of the oldest programming languages still in active use [9]). Lisp-based syntax enables easy to write code that generates code. It is dynamically-typed and provides functional programming style with capabilities of logic programming with its new core logic, which results in compact and "as pure as possible" code. When supplied with a given set of arguments a function will always return exactly the same result.

Clojure is rather impure, but it tries very hard to be functional in most cases this means more robust. In functional programming by avoiding state (and side-effects) the entire system gains the property of referential transparency and a more abstract use of control by using functionals (such as fold, map) rather than explicit looping [10]. State is maintained saved on the stack (like recursive calls), rather than saved on the heap in global variables. Functionals also known as higher-order functions (HOF) take one or more functions as an input and output a function. HOFs (such as partial, comp, juxt complement) makes code more succinct and elegant.

Testing is far more effective, even though testing for one set of inputs says nothing at all about behavior with another set of inputs and for example in a pure functional language it will always be safe to evaluate all arguments to a function in parallel [11]. Data is immutable; after creation, it can't be changed and new versions with modifications are created with a persistent way (efficient memory sharing with old ones). Clojure enables creating of late bindings, macros (operations on a program code), lazy sequences e.g. infinite mathematical sequence items are computed on demand, they are available as a code to run.

Clojure runs on JVM, Java 5 or greater (There are also CLR[4] and JS[5] implementations). It implements Software Transactional Memory (STM), in which every transaction is atomic and isolated. Thus, it works on data copy and intermediate states are not visible to other transactions. There is no need to lock data, when mutable reference types to immutable data (Refs) can be changed inside an STM transaction. If an attempt is made to read or modify a Ref that has been modified in another transaction that has committed since the current transaction started (a conflict) the current transaction will retry up to 10000 times. This means it will discard all its in-transaction changes and return to the beginning of transaction body (such as dosync) [11,12]. Only after finishing transaction all changes made by this transaction are written into shared memory in an act of committing - making changes permanent. Agent Refs are used to run tasks in separate threads that typically do not require coordination.

Compared to Java Clojure code is 'elegant' and succinct (elegant part depends on the point of view of course). To illustrate that claim lets analyze simple factorial function definition example presented on listing 1.

---

[1] Screenshot from Unreal Tournament 2004 game, copyright Epic Games, Digital Extremes, Atari.
[2] Screenshot from Unreal Tournament III game, copyright Epic Games, Midway Games.
[3] Screenshot from Defcon game, copyright Introversion Software.
[4] Implementation for Common Language Runtime called ClojureCLR
[5] Implementation for Java Script called ClojureScript

```
1.  (defn factorial [n]
2.    (apply * (range 1 (inc n))))
```

Clojure implementation

```
1.  public static int factorial(int n) {
2.          int ret = 1;
3.          for (int i = 1; i <= n; ++i) ret *= i;
4.          return ret;
5.  }
```

Java implementation

Listing 1. Factorial implementation.

Clojure code in listing 1 consists of ~40 characters while Java code counts ~80 characters. In Clojure implementation of factorial is completely different than in imperative languages. First (the most inner brackets) n is incremented by 1 (inc n). Then range function generates a lazy sequence of numbers that starts from 1 and ends with n (the end parameter of range is exclusive that's why inc function is used). Example sequence generated by (range 1 6) looks like '(1 2 3 4 5). Finally apply function is used and applies * to the argument list. In our example the result will be equal to 120.

3. INTERFACING POGAMUT WITH CLOJURE

3.1 Overview

Pogamut bots must be run as java threads. Pogamut bot class extends `UT2004BotModuleController` class and implements agent methods like the main logic loop method shown in listing 2.

```
 1.  @Override
2.   public void logic() throws PogamutException {
3.       if (isInitOk()){
4.           mediator.execute();
5.       }
6.       else{
7.           //some logic initialization that can't be done in prepareBot() Method
8.           //(e.g. Requires Agent Body)
9.           setInitOk(true);
10.      }
11.  }
```

Listing 2. Sample logic() method implementation.

Main program execution originates from Java and given that fact Clojure code has to be called from Java as shown in figure 3, not the other way around.

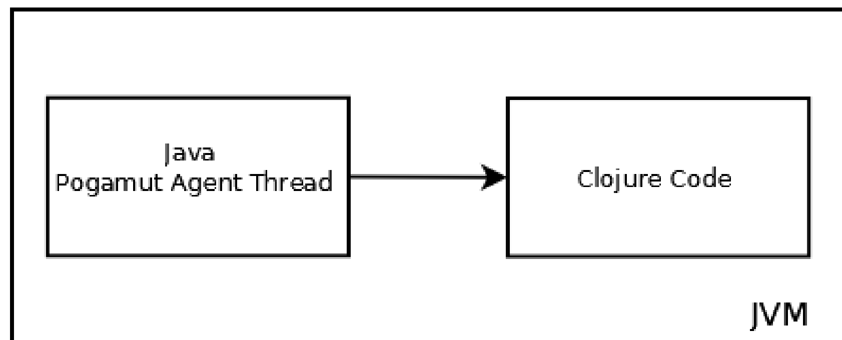

Figure 3. Java 'calls' Clojure Code.

Pogamut 3 is fully mavenized therefore addition of Clojure jar library to Pogamut project requires proper dependency declaration in project pom.xml file. Dependency used in our project is shown in listing 3.

```
1.   <dependency>
2.       <groupId>org.clojure</groupId>
3.       <artifactId>clojure</artifactId>
4.       <version>1.4.0</version>
5.   </dependency>
```

Listing 3. Clojure dependency for Pogamut 3 MVN project.

Clojure library is available in maven central repository. We have used version 1.4.0 of Clojure library.

3.2 Agent architecture

To satisfy our requirements regarding the possibility of reusing the existing bot logic code, and possibility to easily introduce new Clojure code, we have utilized Modular Horizontal Layered architecture (MHL). MHL is a variation of popular horizontal layered architecture widely employed in agent systems. MHL consist of layers and mediators.

Each layer represents a behavior, simple or complex. Each layer should be implemented independently from other layers. Existing bot logic can be wrapped in a single layer and executed as in standard scenario, or can be decomposed and put into several layers. Each layer has access to agent components: memory, communication module, agent body, etc.

Mediators execute layers in a conditional manner. Dependent on virtual environment state and/or bot state given layer or layers will be executed. Mediators can execute multiple layers (sequentially to avoid concurrency problems when accessing and modifying bot memory). Mediator is an entry point for agent logic.

Both mediators and layers are programmed as classes implementing specific Java interface. Therefore layer sets and active mediator can be easily changed at runtime. The use of abstraction (interfaces) allows for simple mediator change (implementation of *Strategy* design pattern) which in turn supports simple and effective way to change bot behavior on the top level of bot logic.

Figure 5 depicts MHL architecture. In figure 4 UML class diagram of mediator and layer interfaces as well as sample implementing classes is presented.

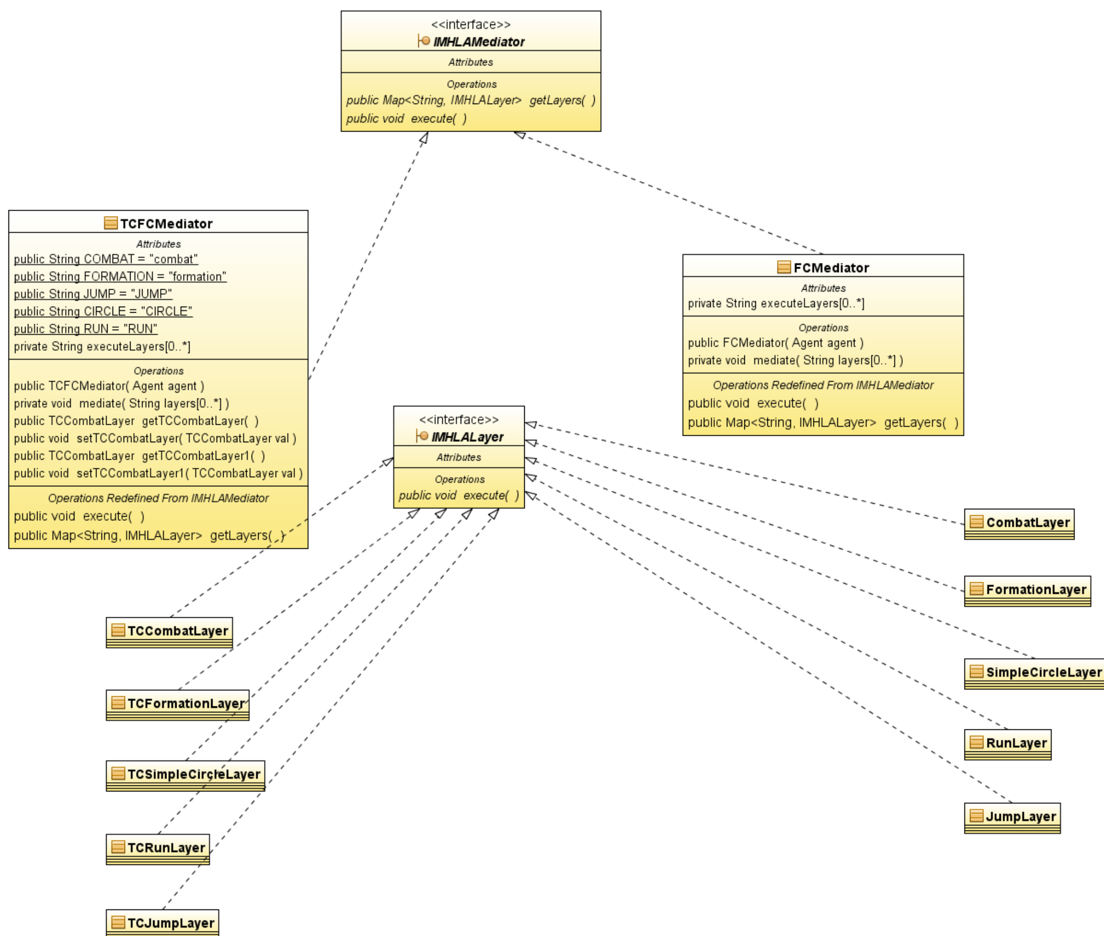

Figure 4. UML class diagram of sample Mediators and Layers of MHL architecture.

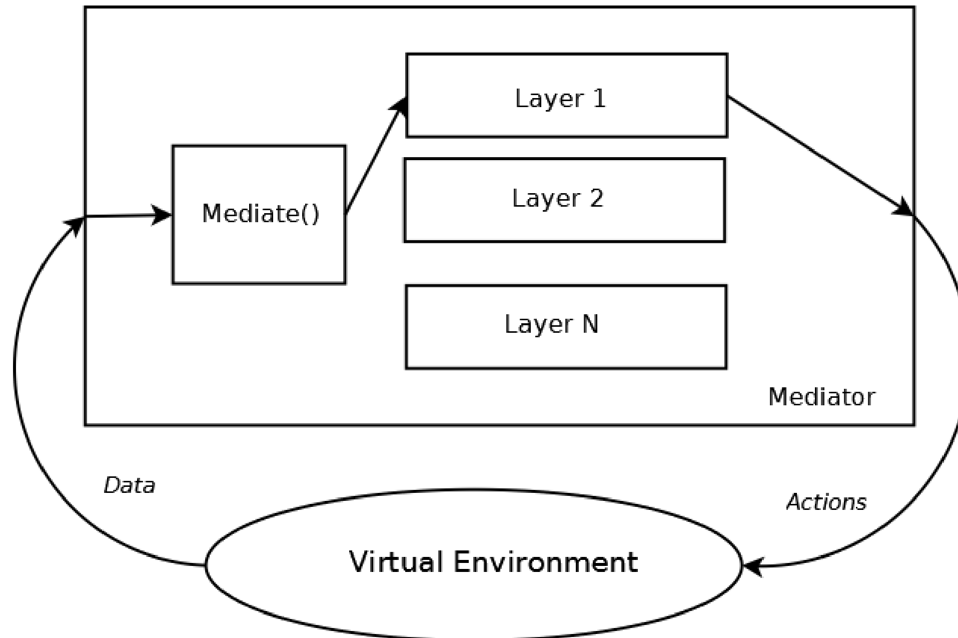

Figure 5. Modular Horizontal Layered agent architecture.

3.3 Calling Clojure code from Java

Java code can be invoked from Clojure as well as Clojure code can be invoked from java [13]. Clojure code can be called from java in three different ways. The first way involves compiling Clojure code into a java class. The other two ways use evaluation method of clojure.lang.RT. The latter two methods might be considered as one but there is a slight difference between both uses.

1. Gen-class – compiling Clojure code into a java class

In a standard case (eg. when using REPL[6]) Clojure compiles code at runtime, therefore compiling Clojure code before it is run is called ahead-of-time (AOT) compilation. Any Clojure namespace can be AOT-compiled [13]. Take a look at listing 4. There are two chunks of code. Clojure source code and Java code that invokes the Clojure source code. As shown in the listing import statement in the Java code uses the namespace declared in Clojure code snippet, but before Java code can be executed the required class must be compiled. To compile Clojure namespace into a Java class user must use Clojure and the *compile* function: `(compile namespace-name)`. After java class has been created Java code from listing 4 should execute properly.

```
     ;clojure code
1.    (ns pl.pw.edu.eiti
2.     (:gen-class
3.       :name pl.pw.edu.eiti
4.       :methods [#^{:static true} [calculate [int int] double]]))
5.   ;(some clojure code)

     // java invocation
1.   import pl.pw.edu.eiti;
2.   double  calc = eiti.calculate(5, 3));
```

Listing 4. Clojure code and Java invocation (gen-class).

[6] Read–Eval–Print Loop. In REPL user may enter Clojure expressions, which are evaluated and the results of that evaluation are displayed to the user.

2. Evaluation (clojure.lang.RT) – loading clj file

Instead of compiling Clojure code to a Java class using gen-class construct, it's possible to use Clojure's Java API, which consists of static methods of the classes clojure.lang.RT, clojure.lang.Compiler, and clojure.lang.Var [13]. clojure.lang.RT is the Clojure runtime class. It allows to load resource files (files with Clojure code and extension .clj) and obtain a Var[7] object. When a Var object for a defined function is available we can invoke that function using invoke method of the Var class. Sample code that shows loading a clj file and invoking loaded function is presented in listing 5.

```
     ;clj file with Clojure code
1.   ; multiply.clj
2.   (ns myspace)
3.   (defn multiply [a b]
4.          (* a b))

     // Java invocation
1.   RT.loadResourceScript("multiply.clj");
2.   Var v = RT.var("myspace", "multiply");
3.   Object result = v.invoke(5, 6);
```

Listing 5. Clojure code and Java invocation (clj file).

3. Evaluation (clojure.lang.RT) – loading 'inline' Clojure code

A slightly different approach (which we call 'inline') uses clojure.lang.Compiler class to load Clojure code directly from a java String object. After loading Clojure code we must also obtain the Var object using runtime (RT), and invoke defined function using invoke method of Var class. Appropriate code is shown in listing 6.

```
1.   String str = "(ns myspace) (defn multiply [a b] (* a b))";
2.   Compiler.load(new StringReader(str));
3.   Var v = RT.var("myspace", "multiply");
4.   Object result = v.invoke(5, 6);
```

Listing 6. Clojure code and Java invocation (inline).

Note that the inline approach allows for (simple) dynamic generation of Clojure code inside a Java code followed by instant execution of created Clojure functions. Important thing to highlight at this point is that Clojure is just a compiler not an interpreter [13], which means that Clojure code is compiled to bytecode and only then passed to JVM for execution.

As we have stated before second and third method both using clojure.lang.RT are very similar, but from our perspective they present different opportunities. For example using a clj resource file allows for dynamic substitution of a large script file, but it requires that Java code reloads the resource on file change. Using inline Clojure invocations allows for dynamic creation of code (on-the-fly) and instant execution.

3.4 Sample Bot Logic

In our bot's logic sample implementation we have used clj scripts and inline invocations to invoke Clojure code. Clojure works very swiftly with seqences and collections. Locations of bots in a group is a collection of points in three-dimensional space. Point is a collection of doubles in Cartesian coordinate system (As represented in Pogamut framework). Therefore as a sample implementation we have used functions operating on locations treated as collections. As an example consider listing 7 depicting the calculation of mass center of agents. Also the calculation of location of agent being the nearest or the farthest to the mass center is implemented, with several helper functions.

We have successfully used Clojure functions to implement simple emergent behavior of bots. Bot behavior is plausible and Clojure code works well. Code samples from listing 7 shows that some operations on data can be implemented in a much more straightforward way in Clojure than in Java. Programming in Clojure requires completely different approach. For a Java (or any other imperative language) programmer, Clojure might be very hard to understand at the beginning, but the pros of introducing Clojure are compelling.

---

[7] Var in clojure is a name bound to value (also a name bound to function). Whenever a function is defined it is added as a Var to the global environment. After it is added, it is available for reference within the same environment [13].

```
1.   ; geometry.clj
2.   (ns geometry)
3.
4.   (defn center [xc yc]
5.       (let [x (vec xc) y (vec yc)]
6.           (map #(/ (reduce + %) (count %)) [x y])))
7.
8.   (defn getpairs [arg1]
9.       (loop [sumpa [] xyc arg1]
10.          (let [xyr (rest xyc)]
11.              (if (some #(= [] %) [xyc xyr])
12.                  sumpa
13.                  (recur (conj sumpa [(first xyc) (first xyr)]) (rest xyr))))))
14.
15.  (defn mean [arg]
16.    (/ (reduce + arg) (count arg)))
17.
18.  (defn middle[arg]
19.      (flatten (let [vpairs  (getpairs (vec arg))
20.                     [x y] (apply map vector vpairs)]
21.                     (map mean [x y]))))
22.
23.  (defn pairs2middle[arg]
24.      (flatten (let [vpairs arg
25.                     [x y] (apply map vector vpairs)]
26.                     (map mean [x y]))))
27.
28.  (defn vlen [a b]
29.      (let [[x1 y1] a [x2 y2] b]
30.          (Math/sqrt (reduce + (map #(* % %) [(- x1 x2) (- y1 y2)])))))
31.
32.  (defn nearestb [vp]
33.      (let [mp (pairs2middle vp)
34.            dp (map #(vlen mp %) vp)
35.            di (map-indexed vector dp)
36.            vi (map-indexed vector vp)
37.            mi ((comp first flatten)(filter #(= (reduce min dp) (second %)) di))]
38.           (second (first (filter #(= mi (first %)) vi)))))
39.
40.  (defn farthestb [vp]
41.      (let [mp (pairs2middle vp)
42.            dp (map #(vlen mp %) vp)
43.            di (map-indexed vector dp)
44.            vi (map-indexed vector vp)
45.            mi ((comp first flatten)(filter #(= (reduce max dp) (second %)) di))]
46.           (second (first (filter #(= mi (first %)) vi)))))
```

Listing 7. Sample Clojure fuctions: bots mass center, nearest and farthest bot to the mass center.

4. PERFORMANCE

To measure Clojure vs Java performance[8] we have used two simple code fragments presented in listings 8 and 9.

First function (listing 8) calculates the sum of elements in an integer collection.

Second function (listing 9) inverts the order of elements in the collection before selecting only even elements and calculating a sum.

[8] In this benchmark performance of Clojure code called from Java (compiled to bytecode and passed for execution to JVM) was tested not the REPL invocations. Memory testing was also done in that scenario, meaning that Clojure had to convert Java collection representations to internal Clojure collection representations.

```
1.  (defn test1 [arg]
2.      (reduce + arg))
```

```
1.  public long test1(List coll){
2.      long out = 0;
3.      for (Integer i : coll){
4.          out += i;
5.      }
6.      return out;
7.  }
```

Listing 8. Benchmarking - first function implemented in Clojure (left) and in Java (right).

```
1.  (defn test2 [arg]
2.      (reduce +
3.          (filter even? (reverse arg))))
```

```
1.  public long test2(List coll){
2.      long out = 0;
3.      Collections.reverse(coll);
4.      for (Integer i : coll){
5.          if  (i % 2 == 0)
6.              out += i;
7.      }
8.      return out;
9.  }
```

Listing 9. Benchmarking - second function implemented in Clojure (left) and in Java (right).

Benchmark consisted of series of function invocations (listings 8, 9) taking randomly generated collections of integers as input parameters. Size of the collection was increased, starting from 10^3 and ending with 10^7 elements. The pseudorandom generation was done using `nextInt(n)` method of `java.util.Random` class.

In figure 6 we exhibit code execution time of the first benchmarking function shown in listing 8. Please note that the time axis in figure 6 is given in a logarithmic scale. Java code executes much faster than the corresponding Clojure code. Observe that the second invocation of Clojure function (for a slightly different data set) is faster than the first invocation. For large sets of data it's even 1.5 times faster.

In figure 7 we present code execution time of the second benchmarking function shown in listing 9. Time axis in figure 7 is also given in a logarithmic scale. Java code executes much faster than the corresponding Clojure code, but it executes more than 2.0 times longer than the code from listing 8. Clojure code of the second benchmarking function executes much longer than the code of the first benchmarking function. For large data sets (10^7 elements) it's even one order of magnitude longer. The second invocation of Clojure function (for a slightly different data set) tends to be even more than 1.5 times faster than the first invocation, but due to logarithmic nature of the graph in figure 7 it can't be easily observed for large data sets.

In figure 7 memory footprint[9] of Clojure code execution is shown. Please note that the memory consumption axis is given in a logarithmic scale. For a large set of data (10 millions integers) additional 120MB (for first function) and 360MB (for second function) of memory is allocated by the JVM to perform calculations. In case of Java code no additional memory was allocated.

Please observe that for data sets with order of magnitude of 4 or less (num of elements $\leq$ 10000) time and memory consumption of Clojure code implementing benchmarking function in listings 8 and 9 is negligible.

Note that Clojure code was executed on a single processor, without taking advantage of concurrent nature of the language. The main strength of this contemporary language is distributed execution. It means that for large data sets, data is partitioned into smaller parts, and distributed between nodes to execute appropriate function 'remotely'. Then results are gathered together to calculate final result.

Due to hardware limitations we didn't conduct appropriate tests to measure performance in distributed environment. In that case results might be completely different in favor of Clojure performance.

Managing concurrency in Java is a hectic and difficult task. Most of existing solutions are based on locking and are vulnerable to deadlocks, and error prone.

[9] Memory benchmark is very naïve. It presents additional memory allocated by JVM to execute code fragment rather than explicit memory usage. Obtaining explicit memory usage for our scenario might be difficult in the JVM environment.

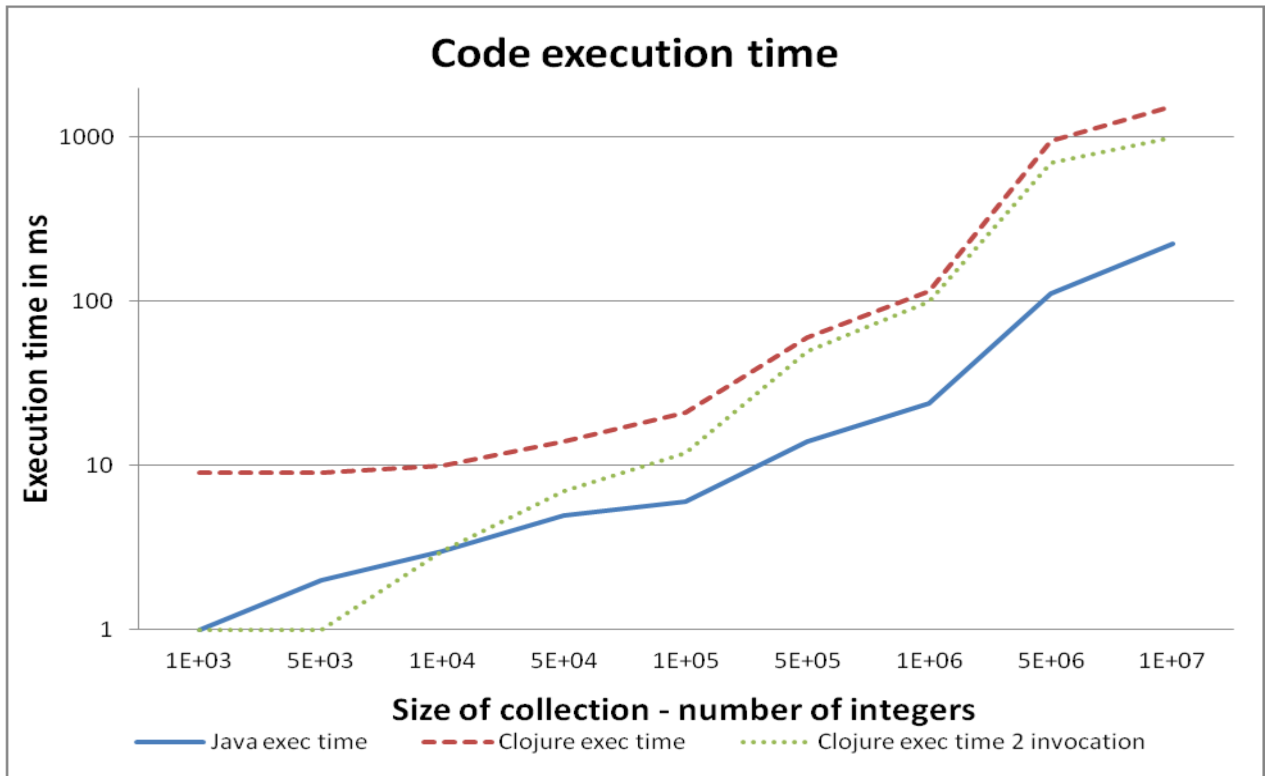

Figure 6. Code execution time for the first benchmarking function shown in listing 8.

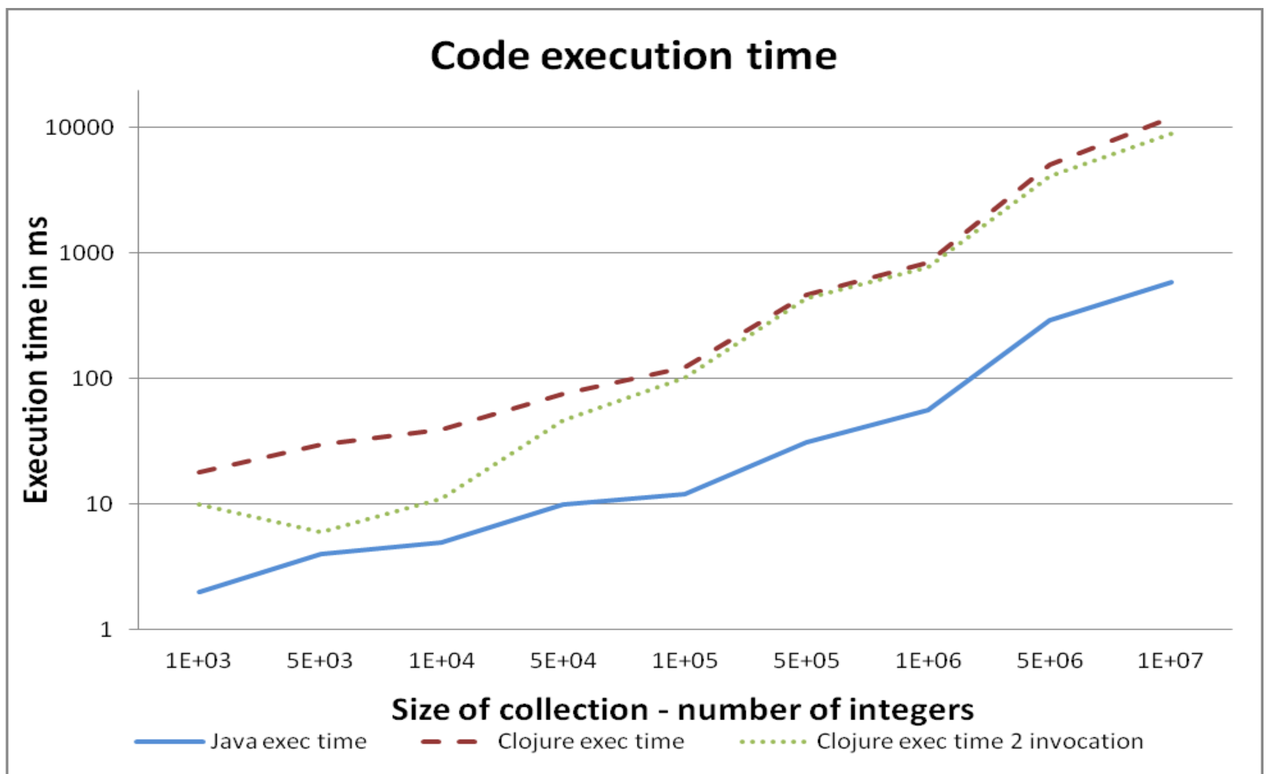

Figure 7. Code execution time for the second benchmarking function shown in listing 9.

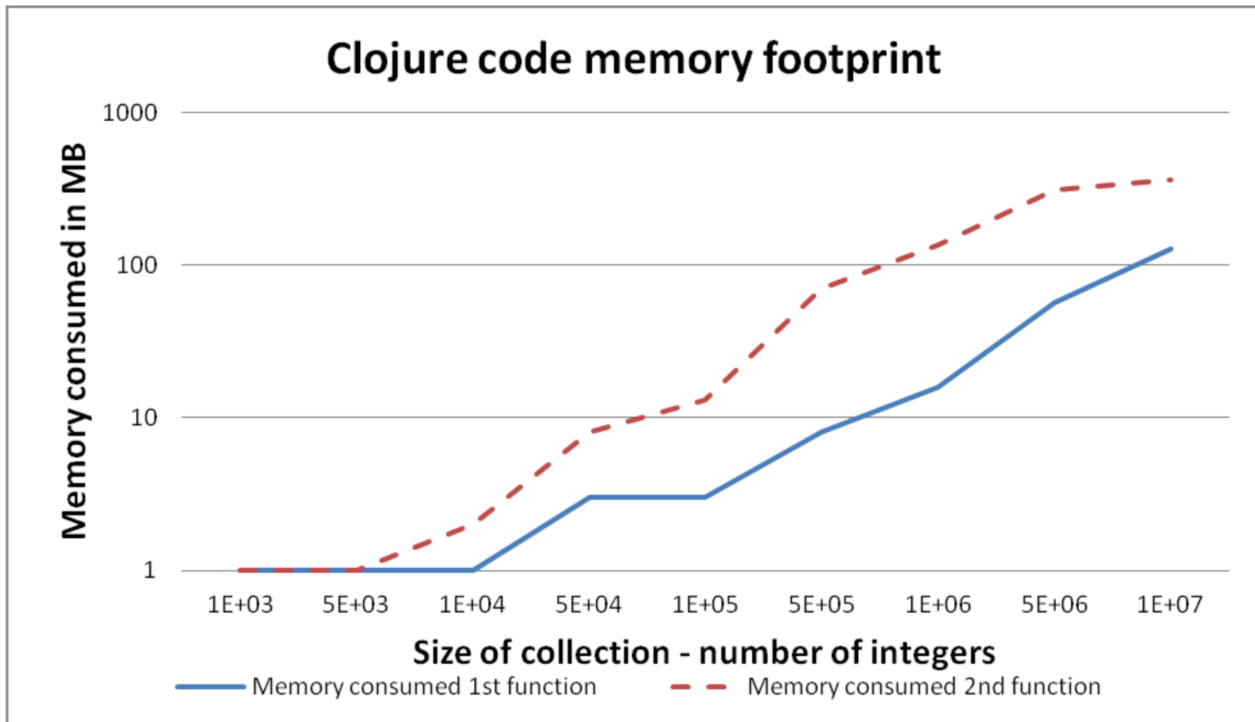

Figure 8. Memory footprint of Clojure code for both benchmarking functions.

5. SUMMARY

Clojure can be integrated with Pogamut platform and we can effectively implement Clojure code fragments in our agent body. Compared to java performance single core execution of Clojure code is ineffective, but the main advantage of Clojure rests in concurrent execution of code. Due to hardware limitations we couldn't inspect distributed scenarios to benchmark potential performance. Given the size of large collections Clojure performance (as for dynamically compiled and executed language) seems to be good even on a single processor. In a large project with lots of bot logic simplicity and clarity of code can be a great advantage for a programmer. Also more advanced features of Clojure might perform much better. Benchmarking functions used are well suited for Java language and that should be also taken into consideration.

The great advantage of Clojure code is that it can be switched and changed at runtime. In case of clj files, complicated functions representing complex bot behavior can be switched instantly, of course each time new clj resource file must be loaded. We have tested that feature and it works really well. In case of inline Clojure code execution, Java can generate different Clojure code based on some conditions and execute that code, it also gives great opportunities for the AI algorithms.

Advanced functions of Clojure like Agents or STM weren't used. They also can prove useful and shall be utilized in our future work during creation of more advanced Clojure bot logic.

REFERENCES

[1] Gołuński M., Wąsiewicz P. "Tactical assessment in a squad of intelligent bots", Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments, Proc. SPIE 7745 (2010).

[2] Gemrot, J., et al.: "Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In: Agents for Games and Simulations", LNCS 5920, Springer, 1-15 (2009).

[3] Kadlec R. "Evolution of intelligent agent behavior in computer games", Master Thesis, Charles University in Prague (2008).

[4] Gemrot J, "Behavior Coordination of Virtual Characters", Diploma Thesis, Charles University in Prague (2009).
[5] Burkert O. "Connectionist Model of Episodic Memory for Virtual Humans", Master Thesis, Charles University in Prague (2009).
[6] Bída M, "Artificial emotions in computer games", Master Thesis, Charles University in Prague (2009).
[7] Halloway S. [Programming Clojure], Pragmatic Bookshelf (2009).
[8] Odersky M, Altherr P., et al. "An Overview of the Scala Programming Language", EPFL Technical Report IC/2004/6 (2004).
[9] Rathore A. [Clojure in Action], Manning Publications (2011).
[10] Moseley B., Marks P., "Out of the Tar Pit", <http://web.mac.com/ben_moseley/frp/paper-v1_01.pdf> (2006).
[11] Volkmann R. M., „Clojure - Functional Programming for the JVM", Partner Object Computing, Inc. (OCI), <http://java.ociweb.com/mark/clojure/article.html> (last update 10 June 2012)
[12] Volkmann R. M., „Software Transactional Memory", Partner Object Computing, Inc. (OCI), <http://java.ociweb.com/mark/stm/article.html> (last update 4 September 2009)
[13] VanderHart L., Sierra S., [Practical Clojure], Apress (2010).

[i] marcel.golunski@gmail.com, mgolunsk@elka.pw.edu.pl
[ii] pwasiewi@gmail.com, pwasiewi@elka.pw.edu.pl