Continuation-based Mobile Agent Migration[⋆]

Jose A. Ortega-Ruiz, Torsten Curdt[1], and Joan Ametller-Esquerra

Cocoon PMC member,
The Apache Software Foundation,
`tcurdt@apache.org`

Abstract. The purpose of this paper is twofold. On the first hand, we review a novel implementation of explicit continuation handling in Java, *Javaflow*. This development avoids many of the pitfalls of previous attempts of enabling this powerful mechanisim in the JVM. On the second hand, we show how this development is of relevance for the Mobile Agent Systems community. More concretely, continuations are used to model and implement strong agent migration, and the arquitecture and design of a basic MAS based on this paradigm is described. The feasibility of all the proposed solutions is backed by explicit implementations.

Keywords: Mobile Agents, Continuations, Strong Migration, Thread Serialization.

1 Introduction

1.1 Continuations

Continuations and the Continuation Passing Style (CPS) paradigm are a well-known, thoroughly studied subject in the Programming Languages research community[7, 5]. We can think of a continuation as a single functional object containing the state of a (sub) computation. When the object is evaluated, the stored computation is restarted where it left off.

In general, a computation C consists of a set of steps or sub-computations, each one yielding a value which is used in an enclosing step, which in turn will be used by its enclosing sub-computation, and so on until the final result of C is built. In other words, the continuation of a sub-computation S yielding a value V is the computational process that will use V to complete the computation C (which has S as one of its subcomponents). Therefore, S's continuation can be represented as a function K taking a single parameter V. C's result is then just $K(V)$.

Usually, a computation execution flow will run uninterrupted, leaving no trace of its sub-components or its internal continuations, except for its final result and (if the computation is not purely functional) side-effects. But in certain circumstances, we may be interested in interrupting C's execution and restart it later and, possibly, elsewhere.

⋆ This work is licensed under the Creative Commons Attribution 2.5 License. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.5/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Our goal is to obtain an snapshot of the program execution state and context, store it, and resume the computation at a late time or in a different place. This suspended, resumable program state constitutes a continuation.

There is a wide range of problems that can benefit from the ability of dealing explicitly with continuations. Error conditions and similar cases of non-local control flow, for example, are typically implemented (at least at the conceptual level) in terms of continuations[3]. In non-deterministic search programs, a continuation aptly represents a node in the search tree, allowing a streamlined implementation of backtracking strategies. E-commerce multi-step transactions (such as a customer buying a book on-line) can be modelled[9] as a single computation which is interrupted each time the server is waiting for a client's response.

1.2 Strong Migration as a Continuation

In this paper, we shall delve into what, to our view, is a paradigmatic case of the scenario modelled by the continuation concept, namely, strong code migration and its application to mobile agent systems.

During its life cycle, a mobile agent will visit a set of hosts in a network, each of them endowed with the infrastructure necessary to accept the incoming agent, let it access the host's computational resources in a controlled way, and provide the means to forward the agent to the next host[1].

Let us view the whole agent itinerary traversal, and its activities therein, as a single computation. At each host, the agent performs a subcomputation, and suspends its execution until the next node in its itinerary is reached, where the computation is resumed. In general, not only the code, but also the computational context needs to be transported from host to host. In other words, a full-fledged mobile agent system needs strong code migration.

Now, it is clear that each resume/suspend cycle in the agent's itinerary can be easily conceptualized as a serializable continuation. If our system is implemented in a language allowing explicit manipulation of continuation objects, the task of providing robust strong migration capabilities will be greatly simplified. Unfortunately, widespread agent platforms are typically implemented using programming languages, such as Java, with, traditionally, poor support for explicit continuation object handling. But the landscape is changing, and new developments, spurred by the web-based application development community, have brought continuation-aware technologies to the forefront[9].

Scheme[11] was the first programming language providing explicit, first-class continuations, followed by Smalltalk and ML. On the other hand, programs written using the CPS paradigm allowed the use of continuation objects even in languages, such as Lisp, without such explicit support[6]. But none of these languages has mainstream status. Today's most popular implementation language is, arguably, Java, especially in the case of distributed applications. The vast amount of libraries available, and its excellent support for networking and multithreading make Java a very attractive platform for agent systems implementation.

[1] We call such infrastructure an agent platform.

An autonomous agent execution flow is naturally mapped to a thread running within the platform's virtual machine (JVM). Therefore, if we are going to represent our agents execution in terms of continuation objects, we must augment the JVM with the ability of suspend and serialize threads, an build on that ability to provide full-blown, serializable continuations.

1.3 Contributions of this paper

As we will see in section 2, augmenting the current JVM implemetations with continuation support is non-trivial. There, we present a new library, *Javaflow*, which extends the JVM with thread serialization and explicit continuation support.

Javaflow is being developed by one of the authors under the umbrella of the Apache Software Foundation and its *Jakarta Commons* project[4]. In that context, the focus has centered until now in using continuations to ease the development of web-based applications. The second contribution of this paper consists of applying these new developments to a totally new domain, namely, the implementation of Java-based mobile agent systems. Section 3 describes the design and implementation of a mobile agent platform offering strong migration services and built on top of the *Javaflow* framework. As we shall see, the ability of using continuations (both conceptually and as an explicit implementation technique), results in a simple, lightweight and elegant arquitecture for our platform.

The article closes with a conclusions section, which also provides an overview of future developments.

2 Java Continuations with Javaflow

The *Javaflow* library is one of the more recent *Jakarta Commons Sandbox* components. It evolved out of Apache Cocoon, which already features continuation based web applications development in javascript for about two years now. But due to reasons like type-safety, IDE support, performance or company standards, a Java implementation has been a requested feature and naturally a good fit to the Cocoon project. So far, the Java virtual machine does not support native continuations although it has lately become one of the most requested features of the Java avant-gardes. Javaflow tries to fill this gap. Since Java continuations are not Cocoon specific, Javaflow has been moved over to Jakarta Commons to provide easy integration and code reuse for other projects.

2.1 The Goal

The code of a Java program represents the full flow of the application. However, in some scenarios it can make sense to split up the application execution into smaller chunks. For example, the execution of a web application is distributed and controlled over a number of HTTP requests; and the need is obvious when our goal is to implement a mobile code system. The application is resumed on every HTTP connection, or after each code migration.

Using Java threads in such scenarios as a naive implementation is not scaleable. Besides, it lacks the requisite reentrancy properties. These requirements lay out the goal for the Javaflow project. Without introducing major code changes, the execution state of a Java thread needs to be captured. While code and data migration is well supported in Java, thread migration is not. So the execution state information has to be extracted differently. Javaflow aims to separate code and data of a thread execution and therefore provide migration of Java threads.

2.2 The Challenge

In Java, every thread has its own program counter and a frames stack. As depicted in the left hand side of figure 1, every method invocation pushes a frame onto the stack. A frame holds the local variables and an operand stack for the method execution. Returning from the method, the frame is being removed from the frames stack and execution is continued inside the new context. In combination with the program counter, this stack of frames provide the execution state information that needs to be captured on thread suspension and restored on resuming it. Saved inside an object, this is what we referred to as a Java continuation.

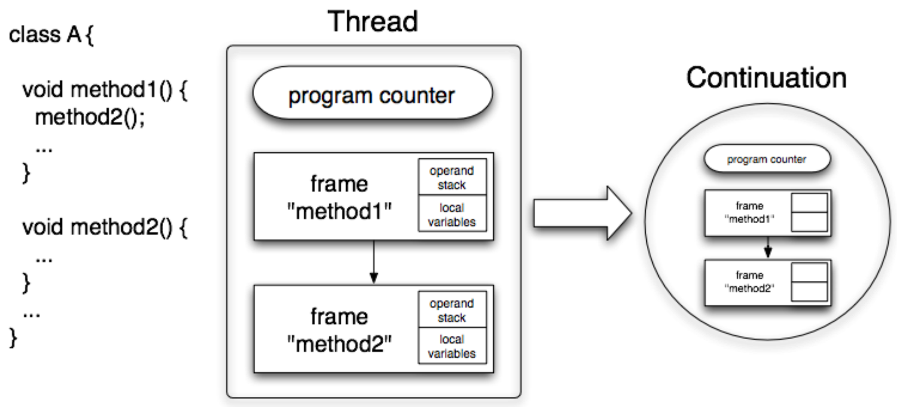


Fig. 1: Capturing Java control flow

2.3 Possible Implementation Strategies

Unfortunately, the Java Virtual Machine (JVM) does not expose access to any of those execution state information easily. As it stands, there are only four possible ways of implementing continuations with the current java virtual machine, which have been studied in the literature:

1. Modifying the JVM[1, 8, 10]

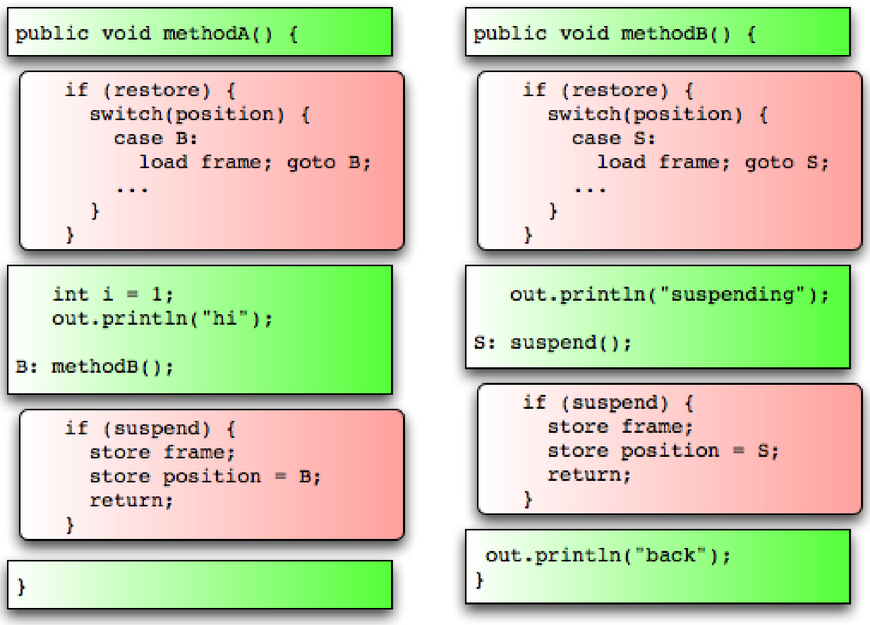
2. Hook into an existing JVM though the JIT or JVMDI interface[14]
3. Rewriting at the byte code level[12]
4. Rewriting at the source code level[2]

Were the modification of the JVM done (or accepted) by Sun and reflected in the Java API, the above list would reflect the order of preference for a clean solution. But, from an external developer's standpoint, the first two approaches require an exacting knowledge of JVM internals. A second drawback is that these solutions are non-portable. As for the last of the above approaches, one of its downsides is that not all virtual machine instructions are available at the source code level.

Thus, we have opted for the byte code rewriting approach. The discontinued *Brakes* research project already provided an implementation based on Java byte code rewriting[12]. Their results formed the roots for the Javaflow project.

2.4 Capturing and Restoring the Execution State

The basic idea of the code rewriting approach is to instrument the class with new instructions providing a higher control of the execution flow. Figure 2 exemplifies Javaflow's rewriting process for a suspend/resume scenario.



```
public void methodA() {

    if (restore) {
      switch(position) {
        case B:
          load frame; goto B;
        ...
      }
    }

   int i = 1;
   out.println("hi");

B: methodB();

    if (suspend) {
      store frame;
      store position = B;
      return;
    }

}
```

```
public void methodB() {

    if (restore) {
      switch(position) {
        case S:
          load frame; goto S;
        ...
      }
    }

   out.println("suspending");

S: suspend();

    if (suspend) {
      store frame;
      store position = S;
      return;
    }

 out.println("back");
}
```

Fig. 2: Code rewriting in Javaflow

When resuming a suspended thread, it is not possible to directly restore the frames stack and then jump to the position where execution was left off. Therefore, the thread has to quickly re-run through the complete call hierarchy to return to the desired position on a resume. While most of the previously executed code can be skipped, executing the invoke instructions will build the required stack of frames. Since the inserted code will also load the content of the frame from the continuation, the original state is restored once the marked program position is reached. Capturing works in a similar way. Once the suspend is initiated, all methods will return immediately while capturing the frame information up until the start of the call hierarchy. The suspend method is special in a way that it controls the flow of application. It initiates the suspend and resets the restore mode.

2.5 The API

The Java byte code rewriting is done either dynamically through a classloader or at build time. It also includes a per-method call stack analysis, which is required to store the frames. The most important API functions are `startWith` and `continueWith`. The `ContinuationContext` provides runtime information that is outside of the scope of a continuation. Once a continuation has been started, `continueWith` provides polling access to the program execution flow. This process is shown in the code skeleton depicted in figure 3.

```
ContinuationContext context =  new ContinuationContext();
Continuation firstContinuation =  Continuation.startWith(methodName, context);
    ...
Continuation nextContinuation =
        Continuation.continueWith(firstContinuation, context);
```

Fig. 3: Continuation usage

As long as all objects on stack are serializable the continuation is serializable. That means that Javaflow can be readily used when strong migration is at stake. In the following sections, we show how to take advantage of this key Javaflow functionality to the problem of strong code agent migration.

3 Mobile Agents using Javaflow

As we have seen in section 2, the Javaflow library extends the JVM platform with new thread handling primitives (resume/suspend) and the ability to capture and serialize a thread execution state in the form of explicit continuations. This section describes how we have taken advantage of this new functionality to layout the architecture of a simple (yet powerful) mobile agent system supporting strong migration. We have also developed a proof-of-concept realization of the proposed architecture, in order to assess the technical feasibility of continuation-based agent platforms implemented on top of Java/Javaflow.

As already noted, continuation-based implementations are being widely used in application servers to solve the problems on navigation flow control introduced by the HTTP protocol stateless nature. We propose using continuations to save the execution state of mobile software agents. Mobile Agents [13] are software entities able to suspend their execution and resume it in a different host. The agent's code, data and saved state are transmitted through the network to the destination host where its execution is to be resumed. In contrast to mobile code used in distributed operating systems for load balancing purposes, mobile agents migration is initiated by the agent itself, without direct intervention of by the underlying operating system.

Our goal is thus designing and implementing a basic mobile agent system providing strong agent migration. Strong migration implies the transmission the data, code and *state* of the agent. The latter aspect is probably the more challenging from a development standpoint. We shall see how continuation objects provide an especially simple way of satisfying this requirement.

3.1 Strong versus Weak Migration

Most mobile agent systems implemented in Java provide what is known as *weak migration*. This kind of migration, easy to implement, consists of moving only the agent's data and code but not the state. This approach has a direct impact in agent design, because the agent itself must be coded as a state machine in order to be able to save the state at application level.

From our point of view, a mobile agent system with strong migration in conjunction with continuations provides an elegant way of implementing mobile agents and applications without artificial limitations to application design. Furthermore, the effort of reusing existing code to make it mobile is clearly reduced when strong (as opposed to weak) migration is available.

3.2 The Platform

The mobile agent platform we have implemented is in a very early stage of developement, but its purpose at this moment is to be a proof-of-concept for the use of continuations in agent mobility. At the present moment, it provides just a minimum framework to build java applications with the ability to migrate. Of course, these applications are executed within a restricted environment (the platform), which provides communication channels to other platforms and migration facilities to its hosted agents.

The basic idea is using continuations to stop agent's execution whenever it calls a mobility primitive. A continuation of the agent's execution state will be recorded. The agent context, and its code in the form of a serialized continuation will be sent through the network to the destination platform, which will use the received continuation and associated context to restore agent's execution at the point where it left off.

The architecture of the platform is pretty straightforward. Figure 4 shows a simplified class diagram of our implementation, which can be breakdown into the following main components:

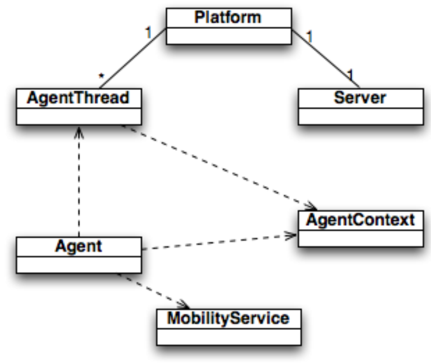


Fig. 4: Platform Class Diagram

Thread pool It is initialized in the `Platform` class when the platform is started and it is used to execute all running agents and other platform processes simultaneously. Its design and implementation relies on the new Java 5.0 thread facilities.

Communications The platform commiuncations infrastructure is used to connect different platforms and has been implemented by means of the *Server* RMI interface. In our current implementation, it is only used to transfer mobile agents between platforms, but it can be extended to provide additional services, such as an inter-platform messaging system.

Mobility Service This entity makes a set of migration functions available to any platform agent.

Mobile Agent A mobile agent in this platform is any Java class implementing the `Agent` interface. This interface extends the standard `Serializable` and Javaflow's `Continuable` interfaces, which are necessary to stop agent's execution and record its data. It also declares a `run` method that will be used as the start point for the execution of the mobile agent.

Agent Control Thread In our system, the agent's class is not a `Runnable` class to be directly executed within a thread. For each agent in the platform we have a Java thread in which we execute the code of at least two classes. The first one is the user's agent main class which, as already mentioned, implements `Agent`. The second, named `AgentThread`, contains the code in charge of controlling the execution flow of the user-supplied agent code.

When the execution of a new agent is requested, the platform creates a control thread using the `AgentThread` class. The latter loads the agent class, creates a new continuation, and the execution flow is transferred to the user's agent code, which starts its execution. When the agent calls the `migrate` function (provided by the *Mobility Service*), Javaflow's `suspend` method (cf. section 2.5 is invoked. This invocation causes the transfer of the execution flow back to the *AgentThread* class. At this point, this class stores the continuation and uses the platform RMI-based communication services to send the agent bytecode and the serialized continuation to the destination platform.

When a mobile agent is received by a platform, the process is basically repeated in the new host. The platform creates an AgentThread instance to control the agent's execution flow. The difference is basically that the control thread is created using an alternative constructor designed to handle incomming agents. This constructor deserializes the agent's continuation, which is later on used to resume the execution flow at the point where it was left off in the previous platform. The agent will be executed by the platform until it invokes the migration method again, provoking a new transition to a remote host.

3.3 Programming an Agent

One of the main goals of implementing a mobile agent platform with strong migration is providing an easy-to-use migration system. The usual way of writting an agent in most mobile agent platforms is extending a provided Agent class. The main drawback of this approach is that mobile agent developers are constrained to a very restrictive interface, and forced to use implementation (as opposed to interface) inheritance. We are able to avoid this design problem by providing a simple interface defining just one method, run, which is used as the agent's execution trigger. Figure 5 shows how easy it is to implement an agent in our platform. Since we are just implementing a (minimal) Java interface, pre-existing code can be easily reused, fostering rapid deployment of mobile agent applications.

```
public class MyAgent implements Agent{

  public void run(){
    System.out.println("Hello");
    MobilityService.migrate("destination ip");
    System.out.println("This will be esecuted at destination");
  }

}
```

Fig. 5: Sample code of a mobile agent

4 Conclusions

This work proposes a new way of conceiving and implementing Java-based agent mobility based upon continuations. The Java Virtual Machine does not provide any native thread flow control mechanism, so there is not an off-the-self way to use continuations in the platform. After studying different solutions proposed in the literature, a bytecode modification approach has seemed to be the best way to provide the requisite thread control flow primitives. The Apache Commons Javaflow component, described in section 2, implements those ideas through a nice and simple API.

The second contribution of this paper consists of the application of the Javaflow library to the development of a lightweight mobile agent platform with strong migration. The main components of the platform are presented in section 3.

The main goal of this platform is providing a simple interface to create mobile agents, which, when combined with the strong migration provided by continuations, results in an elegant method to rapidly deploy mobile agent applications. Since the proposed platform imposes no hard design constraints to application developers, re-use of existing code to create mobile agent-based distributed applications is facilitated.

As future work, we are working in enhancing the platform's capabilities to provide inter-agent messaging and security mechanisms, in order to implement a full-freatured platform. The final step is using the platform to host realistic applications, in order to test and enhance its performance in different environments.

References

1. S. Bouchenak. Pickling threads state in the java system, 1999.
2. E. Elliott. Design and implementation of mojo, a mobile agent precompiler. Master's thesis, University of California, 2000.
3. Matthias Felleisen, Mitch Wand, Daniel Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling full jumps. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 52–62, New York, NY, USA, 1988. ACM Press.
4. Apache Software Foundation. Jakarta commons. http://jakarta.apache.org/commons.
5. Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. *Essentials of programming languages (2nd ed.).* Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.
6. Paul Graham. *On Lisp: Advanced techniques for Common Lisp.* Prentice Hall, 1993.
7. Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst.*, 9(4):582–598, 1987.
8. Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer Verlag.
9. Christian Queinnec. Continuations and web servers. *Higher Order Symbol. Comput.*, 17(4):277–295, 2004.
10. Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, USA, 1997.
11. Gerald J. Sussman and Jr. Guy L. Steele. An interpreter for extended lambda calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.
12. Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in java. In *ASA/MA*, pages 29–43, 2000.
13. James E. White. Telescript technology: Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.
14. Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.