Safe-Threads : a New Model for Object-Oriented Multi-Threaded Languages

Luis Mateu and José Miguel Piquer
DCC - Universidad de Chile
Casilla 2777, Santiago, Chile
{lmateu,jpiquer}@dcc.uchile.cl

Abstract

Threads have been present in programming languages for some time now. However, they have a bad image among software developers because they lead to unreliable applications. Most of the problems are produced by unexpected critical sections, which are very difficult to find. Little research has been done recently to increase the safety of thread programming.

In this paper we present a new model of concurrent threads for object oriented languages. We have called them safe-threads. We claim safe-threads are a major improvement over traditional threads because first, safe-threads are safe from critical section problems, and second, they are location transparent on a distributed system.

The main characteristics of our model are that static variables are not shared among safe-threads and that the objects must provide an explicit interface to be shared.

We also outline how the model could be implemented in Java.

1 Introduction

Threads in Java have a very negative image. On the Internet many important critics have been pointing out their weaknesses. For example, the SubArctic Java Toolkit authors wrote :

> *We urge you to think twice about using threads in cases where they are not absolutely necessary*[17].

Elliotte Rusty Harold, a well-known book writer on Java, said:

> *There is a cost associated with multi-threading. Multi-threading is to Java what pointer arithmetic is to C, that is, a source of devilishly hard to find bugs*[21].

Dave Dyer wrote for JavaWorld Magazine :

> *The Java language contains one feature that is so dangerous, so difficult to avoid using, so hard to use correctly, and so pervasively used incorrectly that it has to rank as a serious design flaw. That feature is multithreading*[10].

We think that the main problem with threads is that they are unsafe. A feature is said to be unsafe, when the cost of using it improperly is too high in debugging time or application reliability. For example the explicit object deletion of C++ is unsafe, and thus, it has been replaced by garbage collection in Java.

Programmers will face a major danger with threads : to leave critical sections undetected. A critical section occurs when a shared resource (typically a data structure) is manipulated from multiple threads. A critical section requires careful synchronization among threads to avoid the concurrent manipulation of the shared resource (through semaphores or monitors for example). Otherwise, the resource could be lead to an inconsistent state.

Unfortunately, it is rather difficult to detect all potential critical sections on a large application, and thus, programmers will frequently leave some critical sections without synchronization. And this is the main source of unsafety : these bugs are *devilishly* hard to find because they become apparent at random times. Final users can experience real bugs, but be unable to reproduce them in presence of the application's programmer. Therefore, threads have a bad image among software developers because they lead to unreliable applications.

Most of the recent research has been focused towards reproducing the shared memory model on distributed memory systems (i.e. transparent object distribution). However, little effort has been done to improve the safety of thread programming.

The main focus of this paper is to propose a new model of concurrent threads for object oriented languages. The primary goal of this model is to be safe from critical section problems. The model is valid for any concurrent and

object oriented language, but we will center the discussion on Java[1].

Our model is inspired by the mechanism for *Remote Method Invocation*[1] of Java (abbreviated RMI). RMI was developed for distributed applications running on different *Java Virtual Machines* (JVMs), not sharing memory. Our model extends this view to all threads, defining that a set of cooperating threads is like a set of communicating JVMs through RMI, each JVM running a single thread.

We argue that the adoption of this model makes multi-threaded applications more robust and makes distribution transparent for threads, executing under the same semantics when running on shared memory or distributed memory machines.

2 Concurrency and Static Variables

A static variable (also called class variable) is a variable that is related to the class itself and not to each object belonging to the class. It is instantiated only once, when the class is first created, and it is shared among all objects of the class and its extensions. It is used, for example, to count the number of instantiated objects of a given class, incrementing it in the object's constructor.

When adding multi-threading to the language, for simplicity of implementation, it is usually defined that the static variables are also shared among all threads. (This comes from the natural sharing of class definitions among threads.) However, this behavior introduces critical sections in the most unexpected places.

To better understand this problem, let us look at a concrete example. A multi-threaded application in Java requires each thread to have its own dictionary. The dictionaries will never be shared, thus they will never be used concurrently.

It is natural then to use a sequential library class developed for sequential programs (single-threaded). However, a problem arises when the internal implementation of this library uses a static variable. For example, a static variable could be used for an internal method to return a second value to the caller (also internal). This simple design introduces an unexpected critical section, as two concurrent calls to the internal method on different dictionaries will have unpredictable results[2].

We will say a class is *reentrant* when each object is guaranteed to work properly if its methods are invoked sequentially. In other words, a reentrant class allows multiple threads to work concurrently on different objects, but it does not ensure proper working when there are concurrent invocations of methods of the same object.

The above dictionary class is not reentrant. But it can be modified to be reentrant by eliminating any use of static variables. As a rule of thumb, a class developed for sequential programs is reentrant if and only if (i) it does not use static variables and (ii) if it uses objects from other classes, those classes are also reentrant.

This incompatibility of static variables with concurrency motivated us to think of a thread model where static variables would not be shared. In such a model, all classes would be reentrant.

3 Transparent Distribution

We say that the threads are transparently distributed when the execution semantics are preserved if the threads are executed on a shared-memory machine or on a distributed system.

Java's Remote Method Invocation (RMI) enables a Java Virtual Machine (JVM) to reference objects that belong to another JVM. Typically, those JVMs are running on different machines, not sharing memory. The objects belonging to a different JVM are called remote objects.

RMI is very useful because it enables a multi-threaded application to execute distributed threads in multiple JVMs. A thread can invoke methods provided by remote objects in the same way as those provided by local objects. The runtime environment transparently sends the arguments to the remote JVM, executes the method there and then retrieves the returned value.

However, in Java the threads are not transparently distributed because the threads executing in the same JVM share the static variables while those executing on different JVMs do not.

One solution would be to extend RMI and the runtime to provide static variable sharing among different JVMs. However, this is really difficult to implement. This fact was a confirmation that it was necessary to take the inverse solution: that the threads executing in the same JVM do not share the static variables. This will also lead to transparent distribution of threads.

4 The Safe-Threads Model

To define our model of safe-threads we will introduce the concept of *logical JVM* or LJVM. An LJVM is conceptually a *Java Virtual Machine*. It has: a single logical processor for executing one safe-thread, logical memory for allocating objects and its own set of static variables, not shared with other LJVMs.

[1] RMI specification available at: `http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html`

[2] Note that adding the *synchronized* attribute to the dictionary methods will not solve the problem in this case because synchronized methods ensure mutual exclusion for accessing instance variables, not static variables.

A safe-thread running in an LJVM will create objects. Such objects always stay in the same LJVM where they were created and its methods are invoked there.

An LJVM is conceptually a JVM, but differs in implementation. A JVM is typically implemented as a heavy process with its own address space. In contrast, several LJVM may be run on the same heavy process. In fact, a single JVM may be multiplexed to simulate several LJVMs, provided that each LJVM keep its own set of static variables.

Our model for safe-threads is inspired by the following metaphor : a program running multiple safe-threads is like a distributed system composed of several communicating JVMs through RMI, where each JVM runs a single thread. Therefore in our model, a safe-thread corresponds to one thread in the distributed system and an LJVM is equivalent to a single-threaded JVM which does not share static variables with other JVMs.

The adoption of this metaphor has the following consequences :

- *An LJVM can use foreign objects.* Objects on an LJVM can reference objects resident on another LJVM (see figure 1). We will call these objects foreign objects. Such objects can live in the same JVM, i.e. they are not necessarily remote objects.
- *Methods of foreign objects are executed in the LJVM where they were created.* An LJVM only executes methods of local objects. When a safe-thread invokes a method of a foreign object, the safe-thread moves to the LJVM owning such object and the method is executed there. The safe-thread moves back upon method return (see figure 2).
- *An object can be referenced from another LJVM if an only if it implements the shared interface.* A shared interface is any interface extending the interface `Shared` and is equivalent to the remote interface of RMI.
- *A safe-thread only knows a foreign object through its shared interface*. Therefore, a safe-thread can invoke methods foreignly if and only if they are defined in the shared interface. A safe-thread can not access the instance variables of foreign objects nor invoke the methods that are not in the shared interface.
- *Upon foreign method invocation, object arguments can be passed by reference only if they have a shared interface.* As in RMI, object arguments not having a shared interface are passed by copy.

We did not reuse the remote interface of RMI for foreign object to stress that foreign objects are not necessarily remote objects : they can live in a foreign LJVM running in the local JVM.

As we can see in figure 2, a safe-thread can execute on many LJVMs, as it invokes methods of foreign objects. Several LJVMs can be distributed on different machines, as shown in figure 3.

Finally, to make this model safe from critical section problems, we state that *an LJVM is a monitor*. In other words, to ensure the sequential invocation of all methods on a same LJVM, we define that only one safe-thread executes at one time on a given LJVM. The LJVM acts as a monitor[12], and a safe-thread can give up the monitor to another safe-thread using condition variables.

Therefore, the main difference between a JVM and an LJVM is : a JVM can execute many threads concurrently while an LJVM never has more than one safe-thread active because it is a monitor.

5 An Example

A consequence of our model is that an LJVM have its own standard input and standard output. In Java those streams are referenced by static variables (`System.in` and `System.out`) and so every LJVM have its own version of them. In this example we will use that feature.

We want to implement, in Java, a classic Unix pipeline between two commands. For example, in Unix the following piped commands count the lines containing the word `hello` in the file `notes.txt` :

```
grep hello notes.txt | wc
```

We can use the same design principle with safe-threads. Let us suppose that the commands `grep` and `wc` are implemented in Java by the classes `Grep` and `Wc` respectively. Each class has its own static method `main`. The Unix pipeline can be implemented with the following code :

```
import java.io.*;
public class CountHello {
  public static void main(String[] args) {
    // Create the output side of the pipe
    PipedOutputStream out=
                  new PipedOutputStream();
    // Create an LJVM for executing wc
    LJVM wc= new WcLJVM(out); // (A)
    wc.start();
    // Change the standard output
    // for Grep.main (B)
    System.setOut(new PrintStream(out));
    // Execute Grep in the current LJVM
    String[] args= { "hello", "notes.txt" };
    Grep.main(args);
    System.out.close();
    // Now wait wc to finish
    wc.join();
} }
```

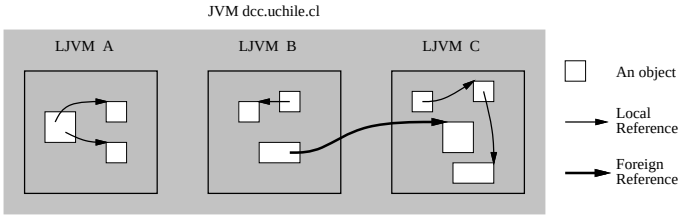

Figure 1. Many LJVMs on the same JVM

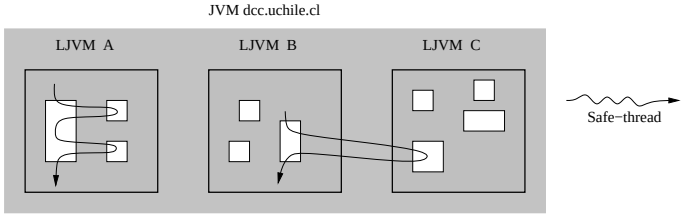

Figure 2. Execution of foreign methods

```
class WcLJVM extends LJVM {
  WcLJVM(PipedOutputStream out) {
    // Create the input side of the pipe,
    // connect it to the output side and
    // change the standard input for
    // Wc.main (C)
    System.setIn(
      new PipedInputStream(out));
  }
  public void run() {
    String[] args= { };
    Wc.main(args);
} }
```

This example is using standard Java syntax and is only including the new class `LJVM`. However, the execution semantics are different in two fundamental points:

- In (B) the variable `System.out` is changed, and we are only modifying it in the LJVM of grep, without changing the corresponding variable in the LJVM of wc. In (C) we change the `System.in` variable of the LJVM of wc because the constructor of `WcLJVM` will execute in the newly created LJVM for wc. This means that the LJVM constructor must be invoked as a foreign method and, for this example to work, we need `PipedOutputStream` to define a shared interface. Its implementation is not the same as the Java standard one.

- In (A) the compiled type of the expression `new WcLJVM(...)` is `LJVM` and not `WcLJVM`. The compiler should make this semantics change when the object being created by `new` is of a derived class from `LJVM`. The type of the expression could not be `WcLJVM` because it would give full access to the instance variables.

An important feature here is that even if it is known that `Grep` and `Wc` were programmed as strictly sequential programs, the multi-threaded example will work correctly. We can be sure that the only shared object is the argument of the `WcLJVM` constructor. No static variables are shared among the LJVMs. Both LJVMs in the example have an equivalent behavior to the Unix heavy-weight processes.

6 Implementation

We will outline here a reference implementation of safe-threads for a single JVM, using Java as the target language. We will not seek an efficient implementation, but a clear one.

The implementation requires the definition of the classes `LJVM`, `SafeThreadImp` and the interface `Shared`. The class `LJVM` contains a hash table to store the static variables for that LJVM, and a monitor for access control. The class `SafeThreadImp` inherits from `Thread` (now hidden) and defines an instance variable that references the LJVM where the safe-thread is currently running. The interface `Shared` is used by programmers to signal which objects can be foreign objects.

To compile static variable access, we proceed as follows: the current executing safe-thread is obtained, and from it, the LJVM where it is executing. The static variable is searched in the hash table belonging to that LJVM. The variable is implemented through an object with an instance variable of the appropriate type.

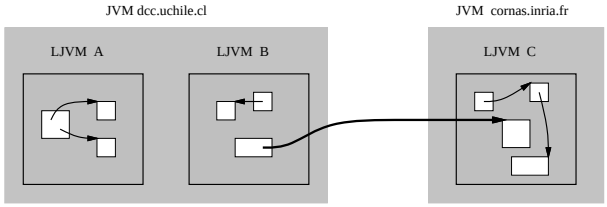

Figure 3. Many LJVMs on different JVM

When a shared interface is defined extending a class without shared interface, the compiler must add an instance variable of type `LJVM`. There we store, for each object belonging to that class, the LJVM where the object was created.

The definition of a method of the foreign interface requires the following modifications. The object through which the method is invoked can be in a different LJVM from the calling safe-thread. In that case, upon method initialization, the LJVM field in the running safe-thread must be made to reference the new LJVM, and its monitor must be acquired. When the method returns, the previous LJVM must be restored in the safe-thread and the monitor must be released.

The compiler must also clone all arguments which do not have a shared interface. The compiler should send an error message when the object is not cloneable nor shared.

Finally, the class constructors must be called only once per LJVM created. In practice, it is more efficient to implement a lazy invocation of the constructors : when the first object of a given class is created or the first time a static variable from the class is accessed.

This description does not attempt to provide a detailed implementation, but to give an idea of the implementation overhead. The proposed implementation does not introduce any overhead for sequential programs not using static variables. If the hash table search is well-coded, the overhead of accessing static variables should be very low.

The overhead for multi-threaded programs should be non-existent because the main cost of invoking a method of a shared object is acquiring the monitor, which is present in both models. However, this will depend on how the applications are programmed.

We are starting a project to implement safe-threads. The goal is to have a platform for experimentation with safe-threads. We will base the implementation on an in-house developed preprocessor for Java. We used this preprocessor to enrich Java with Ada's rendez-vous[16][3]. Basically, we will change the preprocessor so it will now read Java enriched with safe-threads and will produce standard Java, with the modifications described in this section.

[3]The original preprocessor is currently available at : `http://www.dcc.uchile.cl/~lmateu/rendezvous.html`

7 Discussion

We argue that the static variables must be local to each LJVM and not shared. For example, the Singleton design pattern[11] requires the use of static variables. This shows that static variables are frequent when using sequential methodologies because they are useful to implement implicit parameters. Any class using these variables is useless in a multi-threaded application sharing static variables because then they are not reentrant (see section 2).

As static variables are not shared among LJVMs, all classes developed for single-threaded programs are reentrant in applications running multiple safe-threads. Sometimes a shared static variable among safe-threads is needed, but this can always be implemented encapsulating it in an object and passing a reference to the object to all the safe-threads needing it.

We also argue that, when a foreign method is being executed, the static variables accessed by the method should be those of the LJVM where the object belongs, and not those of the LJVM created for the running safe-thread. This is because choosing the latter would make the implementation of distributed transparency of LJVMs very inefficient. The former is compatible with the current implementation of Java RMI.

We also defined that foreign objects are only known through their shared interface, instead of allowing all objects to be shared. We think that object sharing is dangerous due to the unexpected critical sections that could be created. The shared interface forces the programmer to explicitly declare an object as shared and then to encapsulate the concurrent operations in a well-defined interface. We also require this encapsulation for proper compilation of shared methods.

On the other hand, it is useful that the local, shared, objects remain to be known by their classes, giving the compiler more options to optimize them than forcing to use the shared interface.

Finally, we defined each LJVM as a monitor, ensuring mutual exclusion among safe-threads trying to execute in the same LJVM. In this way, all methods of objects in the same LJVM will always be invoked sequentially. As we stated in section 2, reentrancy together with sequentiality

guarantee proper working of sequential classes.

Moreover, we ensure that concurrent accesses to shared objects are synchronized, so they are *safe* from critical section problems, which was our first goal for safe-threads. Here we mean safe from critical sections inside a foreign method executing from beginning to end in just one LJVM. This explains why we chose the prefix safe to name the safe-threads.

On the other hand, someone could believe that by placing coarse-grain monitors on LJVMs, we are increasing the frequency of deadlock problems. We think that this will not happen. On a traditional multi-threaded application, programmers are encouraged to handle concurrency in every class, just to be guarded from unexpected critical section problems. With safe-threads, we expect applications to be decomposed in a tiny set of classes dealing with concurrency, and a large number of classes developed strictly under sequential methodologies (which are better known than concurrent methodologies). Therefore, there are less sources of deadlock to analyze than with traditional threads, and so there will be less deadlock problems.

We chose critical section safeness as our first priority, disregarding deadlocks because we estimate that critical section problems are more harmful than deadlocks. The former damages the application integrity without shooting it down immediately. The problem will be detected when it is too late to find the source (as like in the dangling reference problem in C++). In contrast, when some threads are in deadlock, their states will not change, so they can be examined at any time to diagnose the problem.

8 Related Work

The first research for achieving safeness from critical sections on concurrent programming languages was lead by Brinch Hansen[5] and Hoare[12]. Both works lead to the development of the monitor abstraction. The main idea was that processes could not share data directly. The only way to share data was to put it inside a monitor. The monitor provided operations to manipulate concurrently the shared data and ensured that those operations would be executed in mutual-exclusion.

The work of Brinch Hansen lead to the development of the Concurrent Pascal Programming Language[6] and the work of Hoare lead to CSP[13] and the Occam Programming Language[14]. All these languages were safe from critical sections, like our safe-threads. Unfortunately, to achieve safeness, they prohibited the use of pointers, because it was difficult to ensure that two threads would not share data through pointers. Since object oriented languages are founded on pointers, language designers have been choosing among (i) safety and object orientation without concurrency, (ii) concurrency and object orientation without safety, or (iii) concurrency and safety without object orientation, i.e. no pointers.

Sequential languages like Smalltalk, Eiffel and C++ have chosen (i). Concurrent Pascal, CSP and Occam chose (iii). Java is the most famous example of a language choosing (ii). Although the reference manual claim that Java has monitors, Brinch Hansen pointed out in [7] that they are not true monitors, at least not in the sense that he conceived them. He argues, and we agree with him, that the most important safety measure for a parallel language is to check that processes access disjoint sets of variables only and do not intefere with each other in time-dependent ways. As Java threads can share data without any synchronization, Java monitors are not true monitors.

Besides our work, to our knowledge the only other work combining safeness, concurrency and object orientation is [9]. In this work, the authors implemented a concurrent dialect of Eiffel, providing multiple heavy processes without any object sharing, and therefore, processes communicating through messages. The disadvantage of this aproach is that there were no provisions for inter-process object references.

Recently, a great effort has been done to achieve transparent object distribution [3, 4, 8, 18, 19]. Other works can be found in [2] and [22]. The main goal has been to simplify the programming of distributed memory systems by emulating the objects of a shared memory computer. Therefore *the ideal system has been transparent distributed objects*, i.e. a system where every object can be shared transparently by any thread running on any machine.

The Java RMI (*Remote Method Invocation*) arised as a restricted way to achieve transparent invocation of remote methods without modifying the Java language nor the Java Virtual Machine, at the expense of true transparent distributed objects. Therefore RMI is far from the ideal of transparent distributed objects. It is amazing that the restrictions of RMI inspired us to conceive a model for safe concurrency. Basically, we claim that transparent object distribution is the wrong ideal, because unmonitored object sharing is unsafe. The ideal should be safe distributed LJVMs and, if we are right, the limitations of RMI made it better than its ancestors.

Another problem with massive concurrency is the space allocated to all the stacks, as we need a contiguous block for each thread. In [15] we showed that with generational garbage collection, heap allocation of frames can be as fast as stack allocation, but much more efficient in memory usage. In that paper the study was done for a Lisp dialect, but the results should be quite similar in an Object-Oriented language as Java.

Finally, in a real implementation of this model in a distributed system many new problems arise, as replication and migration of complete LJVMs. For the garbage collection of distributed objects Indirect Garbage Collection[19, 20]

could be used.

9 Conclusions

In this paper, we have proposed a new model for threads, called *safe-threads*. This model is inspired by the following metaphor : a program running multiple safe-threads is like a set of communicating JVMs through RMI, each JVM running a single thread.

Therefore, in our model, we associated one logical JVM (LJVM) to each safe-thread. Objects always stay at the same LJVM where they were created, but an LJVM can reference objects on another LJVM. An LJVM can invoke methods of objects living in another LJVM, but those methods will be executed in the LJVM where the object lives. Moreover, an LJVM is a monitor which synchronizes all the methods invocations of the objects living in the LJVM.

We have used the name *logical JVM* to stress that one physical JVM can be multiplexed to implement many LJVMs. This is more efficient than using one JVM for each LJVM because : (i) LJVMs implemented in the same JVM can share the same code, (ii) references to foreign objects in the same JVM can be implemented as direct pointers, and (iii) invocation of foreign methods of objects living in the same JVM is as eficient as the invocation of a synchronized method in standard Java. Of course, on a distributed system, one JVM must be used for each physical processor.

Today, there have been two main approaches for concurrency. The first considers the application as a set of heavy processes, not sharing memory at all. The second approach views the applications as a set of threads sharing memory. In distributed systems a combination of both approches is used. The former approach is more robust than the latter because it has less critical section problems. However, the former is more expensive than the latter in memory usage and communication time.

The main contribution of this work has been to propose a model for concurrency which combines the best of both approches : robustness because it is safe from the typical problems of critical sections in the shared memory model, and lightness in memory usage and communication time. Furthermore, safe-threads are location transparent on a distributed system. This means that the execution semantics are the same for safe-threads running on a shared-memory computer or safe-threads running on a distributed system or a combination of both.

The main disadvantage of our model is that some concurrency is lost by placing a monitor at the LJVM level because it forbids the concurrent manipulation of objects living in the same LJVM. On the other hand, in Java, programmers are encouraged to maximize concurrency by avoiding the synchronization of methods when they believe there is no critical section. We claim these freedom is harmful because sooner or later the programmer will miss a critical section and the cost of this error will be huge in debugging time. The freedom to choose where to synchronize is unsafe (as the freedom of pointer arithmetic is unsafe in C).

Today people accept the price of having a garbage collector to eliminate the problem of dangling references. In the same way, we think people will accept loosing some concurrency to eliminate the problem of critical sections.

References

[1] Ken Arnold and James Gosling, "The Java Programming language," Addison-Wesley, 1996.

[2] Henri Bal, Jennifer Steiner and Andrew Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, V. 21, N. 3, September 1989.

[3] Andrew Black, Norman Hutchinson, Eric Jul and Henry Levy, "Object Structure in the Emerald System," *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, SIGPLAN Notices, V. 21, N. 11, November 1986.

[4] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy and Larry Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, V. 13, N. 1, pp. 65–76, January 1987.

[5] Per Brinch Hansen, "Structured Multiprogramming," *Communications of the ACM*, V. 15, N. 7, pp. 574–578, July 1972.

[6] Per Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions of Software Engineering*, pp. 199-207, June 1975.

[7] Per Brinch Hansen, "Java's Insecure Parallelism," *ACM Sigplan Notices*, V. 34, N. 4, pp. 38–45, April 1999.

[8] Luca Cardelli, "A language with Distributed Scope," *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pp. 286–297, January 1995.

[9] Denis Caromel, "Toward a Method of Object-Oriented Concurrent Programming," *Communications of the ACM*, V. 36, N. 9, pp. 90–102, September 1993.

[10] Dave Dyer, "Can Assure save Java from the perils of multithreading? Not without your help, but it's a big step in a good direction," *JavaWorld Web Magazine* `http://www.javaworld.com/javaworld/jw-10-1998/jw-10-assure.html`

[11] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, “Design Patterns, Elements of Reusable Object-Oriented Software,” Addison-Wesley, 1995.

[12] C. A. R. Hoare, “Monitors: An Operating System Structuring Concept,” *Communications of the ACM*, V. 17, N. 10, pp. 549–5577, October 1974.

[13] C. A. R. Hoare, “Communicating Sequential Process,” *Communications of the ACM*, V. 21, N. 8, pp. 666-677, August 1978.

[14] INMOS Limited: “OCCAM Programming Manual,” Prentice-Hall International, 1984.

[15] Luis Mateu, “Efficient Implementation of Coroutines,” LNCS 637, *International Workshop on Memory Management*, Saint-Malo (France), pp. 230–247, Springer-Verlag, September 1992.

[16] Luis Mateu, J. M. Piquer and Juan León, “Resurrecting Ada’s Rendez-Vous in Java,” *Proceedings of the XVIII International Conference of the Chilean Society of Computer Science*, IEEE, pp. 106–112, November 1998.

[17] Hans Muller and Kathy Walrath, “Threads and Swing,” *SunWorld Web Magazine*, `http://java.sun.com/products/jfc/tsc/archive/tech_topics_arch/threads/threads.html`

[18] J. M. Piquer, “A Re-Implementation of TransPive: Lessons from the Experience,” *Proc. Parallel Symbolic Languages and Systems (PSLS’95)*, LNCS 1068, Vol I, pp. 310–329, Springer-Verlag, Beaune, France, October 1995.

[19] J. M. Piquer, “Indirect Distributed Garbage Collection: Handling Object Migration,” *ACM Trans. on Programming Languages and Systems*, V. 18, N. 5, September 1996, pp. 615–647.

[20] J. M. Piquer and I. Visconti, “Indirect Reference Listing: A Robust Distributed GC,” *Proceedings of EuroPar’98*, Southampton, UK, September 1998, Lecture Notes in Computer Science, N. 1470, pp. 610–619.

[21] Elliotte Rusty Harold, “Java Lecture Notes,” `http://metalab.unc.edu/javafaq/course/week1/14.html`

[22] Tommy Thorn, “Programming Languages for Mobile Code,” *ACM Computing Surveys*, V. 21, N. 3, September 1989.