Implementing Retry - Featuring AOP

Bruno Cabral, Paulo Marques
CISUC, University of Coimbra, Portugal
{bcabral, pmarques}@dei.uc.pt

Abstract

Everyday experience tells us that some errors are transient, but also that some can be handled simply by "retrying" the failed operation. For instance, a glitch on the network might turn a resource unreachable for a short period of time; or a sudden peak of work on a server can cause a momentary denial of service. In many occasions, without other kind of specialized recovery code, it is possible to keep a program running only by retrying a failed operation. Unfortunately, retry is not explicitly available on many platforms or programming languages and, even if it were, it could not be blindly used for dealing with every abnormal situation. On languages like C# or Java, or even on languages that offer the retry construct such as Smalltalk and Eiffel, where errors are represented and communicated through exceptions, there is no simple way to clear the effects of a failed operation and, thus, re-attempt its execution. Programmers have to explicitly write sometimes complex and error-prone code to repair the state of a program and the execution context. In this paper, we propose an AOP technique for implementing "retry" on systems lacking such a feature without using any language extensions for AOP or imposing modifications to the development language. Our approach eliminates the need for programmers to write "state-cleaning" code for normal objects by means of a transparent transactional mechanism and provides the means to identify non-idempotent operations on the code. In our evaluation we show that a relevant number of application failures can be masked using this approach.

1. Introduction

Retrying a failed operation is many times the most simple error recovery strategy that we use. Though, the limits for when it can be applied are dependent of the kind of anomaly affecting the system; the complexity of the program, if the failed operation is idempotent or not, and of the programmer's will and skill to write code that can be re-executed. Many programming language designers advocate that "retrying" is an essential system feature for implementing resilient code [1].

There are many systems where the retry operation can be useful. Perhaps the most flagrant examples are associated with distributed computing applications. Considering that a significant fraction of the software used nowadays has a strong relation with the Web or networked environments, it is common for users and applications to have to cope with transient errors. Errors than can cause an operation to fail in one moment but to succeed on a second attempt. For example, when trying to access a web address on a browser it is possible to obtain an error on a first attempt and a successful access a second later. Likewise, assessing an SMTP server to send an e-mail can be unsuccessful on a first attempt but immediately succeed on a second attempt. The reasons why this happens are not always the same and can be fairly diversified - the server can be too busy to respond on that instant, there can be a momentarily network failure, an unexpected and temporary resource unavailability, and many other plausible causes. But, the fact is that without any other kind of specialized recovery code, it is possible to keep a program running on a valid state only by retrying a failed operation.

Retry operations may not always be as simple as first expected: it may be necessary to wait for a predetermined period of time before re-executing the failing code; it may be necessary to retry-more-than-once; it may be necessary to make adjustments to the program state or context before retrying; or, all of the above simultaneously. But, in any case, a skilled programmer can always deal with such concerns easily. A bigger issue "is how to write idempotent operations in a simple manner?" Although, such goal may be unattainable for some operations, for a large number of situations it can be achieved with careful programming. Furthermore, such programming tasks can be substantially simplified if the runtime environment provides means to automatically repair the application state after an error.

Several programming languages, such as Smalltalk and Eiffel, implement a "retry" construct. But, many developers consider it hard to use: programmers have to undo by themselves the effects of failed operations. That implies being forced to know the internals of the invoked methods in order to assess if they are idempotent or not. In general this is quite difficult to do since there can be several methods in the call stack, being executed. Also, forcing programmers to consider the internals of different methods breaks encapsulation – one of the key design principles of well structured applications.

Nevertheless, no one argues the usefulness of this mechanism. Even in languages that do not implement the retry construct, we can see developers using a combination of other language constructs to obtain the same effect (e.g. combining `for`/`while` loops with `try` blocks).

In this paper we propose a new mechanism for implementing retry on object-oriented systems lacking such a feature. Our approach is based on Aspect Oriented Programming (AOP) [2] and uses a Software Transactional Memory (STM) [3] mechanism to enable *rolling back* failed operations.

```
try {
    // the transaction is initiated here

    // try to send a datagram
    socket.send(dp);

    // the transaction commits when
    // leaving the try{} block
} catch (IOException ce) {
    // the transaction is aborted and
    // its effects are eliminated when
    // the handler begins executing. A
    // new transaction is initiated.

    // try to reconnect the socket
    socket = new DatagramSocket();

    // the programmer requests the re-
    // execution of the failed block
    // if some condition is verified
    if (...)
        retry();
    // Note: if retry is not executed the
    // handler transaction commits here
}
```

Figure 1. Code sample illustrating the use of a retry operation.

In our system, `try{}` blocks are transactional. Thus, each method invoked inside these protected regions is part of a transaction. A transaction is *started* when entering the `try{}` block and is *commited* if no exception is raised during the complete execution of that block of code. If an exception is raised, the transaction is *aborted* at the start of the `catch` statement. At this point, it is possible to perform the necessary operations to try to correct the problem and allow the re-execution of the failed code block. An important point is that the heap and stack variables of the application are rolled-back, representing a clean state in the program. After performing a recovery action, re-execution of the protected block can be requested by the programmer. This is illustrated in Figure 1. Although very simple, this example illustrates the basic principles of our approach. More complex issues, such as nested transactions, multi-threaded code and the execution of non-transactional code along with transactional, will be addressed on the following sections of the paper.

The remaining of this paper is organized as follows: Section 2 discusses the related work. Section 3 describes the programming model and the system architecture. The implementation, testing and evaluation data are discussed on Section 4. Section 5 draws the conclusions.

2. RELATED WORK

The `retry` instruction is available on several programming languages, such as Smalltalk [4], Ruby [5] and Eiffel [6]. On these languages, `retry` works sort of like a `goto` statement by restarting a failed block of code after some corrective measure has been taken. If an exception is raised inside a protected block, execution jumps to the handler for the block where the problem occurred. In there, a recovery operation can take place before re-executing (retrying) the original block of code. This can obviously lead to an infinite loop if the fault causing the problem is not corrected. Nonetheless, `retry` is a good and simple way to implement re-invocation upon recovery.

```
SqlConnection conn = new Connection();
conn.ConnectionString = ...;
for (int i=0; i<max; i++) {
    try {
        conn.Open();
        break;
    } catch (SqlException ex) {
        Thread.Sleep(10000);
    }
}
```

Figure 2. Retrying to open a database connection up to max times.

Smalltalk also offers a special kind of `retry`, the `retryUsing`. It works like `retry` but replaces the original block with the one given as an argument. With this operation, failing a second time passes the exception up the chain.

The Eiffel documentation reports that the `retry` clause is only used 16 times on all the 2000 classes that constitute the Eiffel libraries and mainly on networking and database oriented operations. Nonetheless, its usefulness is obvious if we consider that developers using languages, such as C# [7], that do not have a `retry` instruction use other constructs to mimic its functionality. Figure 2 is an example of such practice. If we consider that the `SQLException` might be raised because the server has not yet been started, it makes sense to wait for 10 seconds and retry the operation. Using AOP, the same result could be achieved in a much organized fashion and transversely to the code of the program.

Exception handling has been considered an interesting application area for AOP since its origins [2]. Some authors have already assessed the feasibility of using AOP for dealing with exceptions in a clean and efficient way [8][9][10][11][12][13]. As a result, the AOP community also recognized the need to support the retry functionality as an aspect of particular handling actions [14][15][16].

If we consider exception handling as a *crosscutting aspect* of a program, the locations where an exception is raised are the ideal points (*join points*) for inserting (*weaving*) code (*advice*) for dealing with the occurring exceptions. This happens in a number of frameworks that use this functionality [15][17][18][19][20]. For instance, in *Spring* [18], it is possible to declare rules (*point cuts*) to weave exception handling advices: Spring 1.2 provides the `AfterThrows` advice interface whilst Spring 2.0 supports `@ThrowsAdvice` annotation. Either one of the mentioned constructs will ensure that, when a certain exception is thrown, the advice specified is invoked. Typically, each advice defines a method which accepts an exception class as an argument (e.g. `void afterThrowing(Exception exp)`).

In *Spring.Net* [19][20] it is also possible to specify a retry advice. This can be done using a Domain Specific Language (DSL). A simple example, from Pollack et al. [21], is: `on exception name ArithmeticException retry 3x`

```
RetryTemplate template =
   new RetryTemplate();

template.setRetryPolicy(
    new TimeoutRetryPolicy(30000L));

Foo foo =
    template.execute(
        new RetryCallback<Foo>() {
          public Foo doWithRetry(
            RetryContext context) {
            // business logic here
              return result;
          },
        new RecoveryCallback<Foo>() {
          Foo recover(RetryContext context)
            throws Exception {
            // recover logic here
          }
});
```

Figure 3. Code example for Spring Batch retry.

`delay 1s`. The semantics of the example are: "*when an exception that has* `ArithmeticException` *in its type name is thrown, retry the invocation up to 3 times and delay for 1 second between each retry event.*" Furthermore, *Spring Batch* [22] has the `RetryOperations` strategy - with this framework it is possible to automate retry operations on the code. Figure 3 shows a call to a web service which returns a result to the user. If the call fails the operation is retried until a timeout is reached. If the business logic does not succeed before the template decides to abort, then the client is given the chance to do some alternate processing through the recovery callback. As it can be seen on this example, current approaches for weaving exceptions are somewhat convoluted, leading to complex and error prone code.

Our approach does not require the use of DSLs, or of any specific language construct. It is also more refined than the previously described methodologies. Since it assures automatic rollback of the effects of failed operations and allows the validation of retry usage at compile-time, code becomes much cleaner, maintainable, and the programmer is freed from trying to recover the state of the application.

3. ARCHITECTURE AND PROGRAMMING MODEL

Exception handling models, existent in modern programming languages (e.g. C++ [23], C# [7], Java [24]), commonly use `try` blocks to "guard" method calls that might raise exceptions during their execution. In our programming model, each `try` block in a program is also a transaction. In fact, this is the major difference between our approach and "traditional" exception handling models. It is also a key point of the whole approach. Suffice to say that this transactional type of system actually corresponds to having a Software Transactional Memory (STM) framework in place where the `atomic` keyword corresponds to a `try` block.

In terms of the exception model, our approach uses the *termination* model [25] with *checked exceptions*. The termination model offers greater simplicity than the *resumption* model [25] and conceptually allows the usage of the *retry* construct. Checked exceptions support the creation of compile-time code checking rules and, at the same time, augment the programmers' knowledge about invoked methods. With checked exceptions, developers are forced to declare a list of exceptions that each method might raise, thus, such information will always be available to developers and development tools if necessary.

In terms of the programming model, the system works as follows:

1. While writing code where exceptions can be raised, the programmer demarks them with a normal `try` block. The code inside the protected block becomes part of a transaction that starts and ends within the block limits.
2. The programmer specifies `catch` handlers for the exceptions raised inside the protected region of code. The transaction initiated on the associated protected block is aborted if an exception occurs and the flow of execution is passed to a handler block. The application state, in the heap and stack, is rolled back at this point.
3. If the programmer wishes to do so, he or she can call `retry()` (only) inside the `catch` handler. This will prevent the remaining of the code on the handler from executing and will re-attempt the execution do the failed `try` block immediately.
4. The programmer can write code to make the execution of the `retry()` method conditional and to control de number of times a `try` block should be retried.
5. If the failed `try` block is not to be re-executed, the execution continues inside the handler and, at the end of the block, it jumps to the first instruction outside.
6. If an exception is not handled by the `catch` handlers associated with the faulting block, it must be declared on the method's exception list or handled by a surrounding `try-catch`. Thus, `try` blocks can be nested and, consequently, also transactions.

When retrying the execution of a `try` block it is essential to guarantee:

1. The application state (variables, data structures, etc.) is as it was on the first execution of the block.
2. The execution of the `try` block is isolated. This means that if external I/O operations are executed, it must be possible to undo them, or they must be idempotent.

These two aspects are discussed next.

3.1. Restoring Application State

Restoring application state is vital in order to guarantee that the intent of the programmer is preserved. For instance, if a programmer is incrementing a variable "`total`" inside a `try` block, if the block is re-executed due to an exception, after exiting the block the variable must only have been incremented once, i.e. the program semantic is correct independently of the recovery system in place.

For restoring the application state several approaches are possible. Typically methods include:

1. When entering a protected block, create a copy of all touched objects. These copies are used whenever updates are made. If the block exits normally, the original objects are replaced by the updated ones. If a re-execution takes place, the copies are discarded, being the original objects preserved.
2. No copies are made. Instead updates are done in place. All changes to objects are logged. If a block exists normally, the logs can be discarded. If a re-execution takes place the logs are used to restore the objects to their initial state.

The details regarding each method are actually somewhat tricky to implement and incur in different overheads. In object-oriented platforms, the first approach feels more natural, though it may require more memory than the second one and rely heavily on the execution of the garbage collector. Although a transactional model is essential to our system, its definition is not the core of this proposal. As a general guideline, an STM system for usage with our model must support *closed nested transactions* [26][27]. Other researchers are currently intensely investigating how to optimize such systems, especially in terms of concurrent programming, and make them available on the mainstream. In our case we not only benefit from those systems becoming generally available (e.g. AtomJava [28], Atomos [29]), but also the associated overheads for exception handling can be much lower. In general, we expect `try` blocks to succeed, thus the system can be heavily optimized in that direction.

3.2. Isolation in try blocks

As it was mentioned, preserving only application state is not enough. For instance, some operations offer serious challenges – "we do not want to credit a value into a bank account more than once; we do not want to write the same data to disk more than once; etc". If such problems arise, what it means is that the transaction is not isolated from the exterior. It leaks information.

For dealing with this problem, as a general rule, if an unprepared class (i.e. non-transactional aware) is detected inside of a `try` block, the compiler forces the programmer to remove retry orders from the associated handlers. Furthermore, it marks the method as not being transactional-aware thus alerting the developer for a situation where it might be used in a transactional environment.

This means that for some legacy code, the programmer is in the same situation than today – the runtime system cannot help him. Also, some I/O actions cannot ever be undone (e.g. consider the case of "firing a missile"). Thus, it is necessary to allow transactional and non-transactional code to coexist in the same program. But, transactional code cannot reference non-transactional classes and methods.

Part of our technique consists in weaving transaction management code into programs. Thus, it is mandatory to be able to distinguish which classes can be made transactional and which can not. Classes transaction-aware must somehow be recognized by the runtime system. A simple approach for this problem is making them implement a `Transactional` interface. Transactional-aware classes must explicitly provide at least three callback methods on this interface: one for the runtime system to signal that a transactional context is being entered; one for the runtime system to signal that the changes are to be committed; and one for the runtime system to signal that the operations have to be undone. Since `try` blocks can be nested, an identifier for each transactional context must also be available.

Although using this approach may apparently seem complicated or cumbersome it is not so in practice. Only classes that perform external I/O operations must be carefully marked in this way. Ordinary objects are already automatically made transactional by our STM system. The complexity of implementing the callback methods varies, but their semantics is clear.

In terms of the classes that can be made transactional, several alternatives exist:

1. If I/O is involved, it may be possible to temporarily buffer it, until a *commit* is possible.

2. In some cases, it is possible to create idempotent operations for the class. This means that the operations can be repeated.

On a final note, it should be mentioned that the same problems that occur in database systems and STM systems can also take place. For instance, if inside a transaction the data previously written is read, it can lead to problems (e.g. consider the case where inside a `try` block a `FileReader` tries to read data that has just been written. Since the writer has not yet committed, the reader may not be able to read it on a naïve implementation.) There is currently no general solution for those problems. The implementer of these classes must take care in preventing them. In most cases it is simple to provide hooks for undoing the operations or detect conflicts with other operations. Also, in a general development platform, the number of classes that are directly involved with external I/O is limited. In any case, if supporting a certain class proves to be too hard, the platform provider can always opt for making it non-transactional.

3.3. AOP methodology

The main purpose of our technique is to allow programmers to use the *retry* construct on systems lacking such feature in a simple way. Furthermore, from the beginning of the design process, we decided that such feature should be introduced without imposing substantial modifications to the targeted programming languages or runtime environments. Basically we *weave* the new functionality into the code of the applications at load-time – thus the usage of AOP.

With AOP, retrying and transactions become *aspects* of the target programs. Classes, fields, methods, and instructions on the code are handled as points (*join points*) where it is possible to insert additional behavior. The inserted behavior is named *advice* and the rules that guide the identification of *join points* and the *weaving* of *advices* are called *pointcuts*.

In the next section, we will detail the most significant

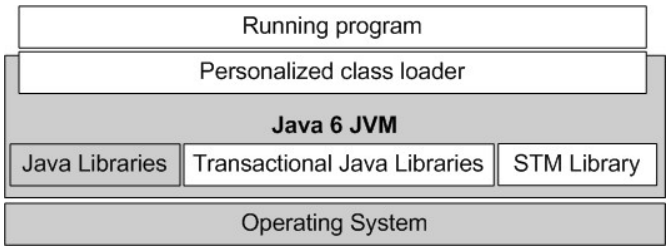

Figure 4. System architecture.

aspects of an implementation made for the Java platform. But, before we can speak of such details, a short summary is in place. It is important to understand that our approach comprehends two simple steps performed at load-time: a) make loaded classes transactional; b) replace calls to retry by jump instructions. On each class that can be transactional we append interfaces, methods and fields necessary for accessing the functionality of a STM library. We also replace the objects access operations (such as read and write fields values) by calls to special methods that guarantee the transactional behavior on data accesses. Furthermore, we introduce calls to the methods starting, committing and aborting transactions at the required locations. Calls to retry must be replaced by jump instructions to the starting location of the code block to be re-executed.

4. IMPLEMENTATION AND EVALUATION

The mechanism described in the previous section was implemented for the Java 6 platform. It consists in two major components:

1. A custom-made *Software Transactional Memory System* implemented as a library. Although it would be possible to use other available STM libraries for Java, at the time we started the project there was a definite lack of such implementations. That led us to rollout our own system. It is a simple STM implementation based on object shadowing and supporting closed nested transactions with strong atomicity. Admittedly, in the future, we may replace it by a more mainstream STM library.
2. A Java system *class loader* which performs bytecode instrumentation at load time. It does the insertion of the appropriate code so that the transactional system is correctly invoked. It is also responsible for making sure that the correct application state is available for the developer, if necessary, be able to internally define the code to reconfigure the application.

The architecture of our system is illustrated on Figure 4. In the diagram, it is possible to observe the building blocks of the system organized accordingly to their run-time relationships in regard to the Java 6 JVM.

In order to implement the transactional model, it was necessary to modify some of the system's classes and libraries, providing new transactional implementations (*Transactional Java Libraries*.) These libraries will still work with the remaining default system libraries (left unmodified.)

Additionally, a new software packages was incorporated into the JVM: the *STM library*. This library implements the types, interfaces and functionality necessary for creating the transaction environment for program execution.

The running applications "do not know" anything about transactions, since neither the programmer nor the compiler will introduce such information on the program's code (except for `try` blocks and `retry` invocations). Thus, it is necessary to modify or introduce new code into the programs being executed in order to allow them to benefit from the new functionality. The *Class Loader* mechanism of the JVM allows us to perform such modifications on programs by intercepting the loading of new classes into the JVM, and allowing the modification of their code before they get executed for the first time. The *Customized Class Loader* has the tasks of: parsing the bytecode of the classes being loaded and locating protected blocks of code; inserting code that allows the transactional execution of each class's methods; and the substitution of retry statements. Basically, the class loader is responsible for interconnecting the functionality provided by the STM library and the Transactional Java Library, with the code on the target programs.

4.1. The STM Library

At the core of the STM library there is the `Transaction` class. This class implements the methods for creating, committing, and aborting a transaction. The transactional system contains a list of the transactions. This list contains the transactions that are in execution, and that have been committed or aborted.

`Transaction` has a number of key operations and data structures. In particular it holds a read-set (`rObjects`) and write-set (`wObjects`) of objects that have either been read or touched during the transaction. When a new transaction is initiated, the system registers its parent thread identification, provides a new transaction identifier (sequential number), records the local time as the transaction start time, sets the transaction status to "running", and adds the new `Transaction` object to the list of transactions.

A `commit()` method performs several verifications before allowing the changes made on ending transaction to be persisted. This method starts by setting the current `end_time` for the transaction. Afterwards, the execution enters a mutual exclusive section of the code where it checks if there are any conflicting exceptions that can prevent the transaction from committing: basically if the read-set of a transaction intercepts with the write-set of another.

Since changes made inside the transactions are not visible to other concurrent transactions it is necessary to maintain a list of object versions for each time a nested transaction is started. Objects on the system are organized on the form of a double linked list. At the head of the list we have the original object (the first instance to be created) and the following objects are copies made to be modified inside transactions. The first time a transaction touches an object with the intention of modifying it, the system makes a copy (the clone) and adds it to the instances list for that object. The copy can

be based on the original object or on the active version on the parent transaction. When commit is performed, the system locates the original instance of the clone object being committed on the ending transaction and moves the changes of the clone onto the original.

4.2. The Customized Class Loader

A custom class loader is used to intercept the loading of classes into the runtime system (JVM). By doing so, we are able to instrument the original code of the classes and provide the JVM with new transaction-enabled implementations of those classes. This is the mechanism responsible for *weaving* our *advices* into the code of the program.

To perform the instrumentation of metadata and the bytecode on the `.class` files we used the ASM library [30]. Although, there are other options in terms of Java bytecode instrumentation libraries (e.g. [31]), ASM is currently one that provides better performance and smaller memory footprint. Such qualities are essential in our system. The ASM library implements the *Visitor design pattern*. This allows modifications to propagate through an application's code in a systematic way. The visitor pattern provides the means to instrument (visit) each component of a program (classes, fields, methods, and constructors) in a specialized way.

When a class is first loaded by the class loader, ASM adds a new interface to it (`ITransObject`). This interface allows the runtime system to duplicate objects as needed by the transactional system. It also allows the transactional system to navigate through the list of shadow copies created when a new transaction is started, committed or aborted, adding, removing or updating the corresponding copies as needed. Finally, this interface allows the runtime system to gather context information about parent transactions when nested transactions are taking place.

Just adding mechanisms for creating and updating objects by using code instrumentation is not enough. It is essential to ensure that the code of the modified classes uses the fields and variables of the correct objects within a transaction. This means that the bytecode of the classes has to be modified or it would access the original object references. Thus, the following modifications are made:

- Replace the `PUTFIELD`, `GETFIELD`, `PUTSTATIC`, `GETSTATIC` instructions by calls to custom field access methods;
- Replace the `ILOAD`, `FLOAD`, `ALOAD`, `LLOAD`, `DLOAD`, `ISTORE`, `FSTORE`, `ASTORE`, `LSTORE`, `DSTORE`, and `RET` instructions by calls to the local variables access methods;
- Locate `retry` instructions (actually, for simplicity we opted for using a method call instead of special instruction) on the code and replace northem by jumps to beginning of the `try` code block to be retried;
- Insert calls to `startTransaction()`, `commit()`, and `abort()` methods at the start of `try` blocks, `finally` blocks and exception handlers.

4.3. Validation and testing

To evaluate the effectiveness of the approach we used 11 different applications. These consisted in the examples that come with the Sun's Community Version of its Java System Message Queue (MQ) framework – *GlassFish* [32], implementing the JMS standard. We have chosen this system because it is a widely used platform and, at the same time, the test applications are not so complex, allowing focused experiments to be made.

Our purpose, while performing such experiments, was to assess the feasibility of the mechanism on allowing the usage of retry semantics on Java programs and to verify the performance penalty imposed by the STM system.

We started by analyzing the source code of the test applications and by locating the exception handling code. Overall, we observed that the existent exception handlers were very small and simple. In most cases they were limited to notifying the user when exceptions occurred, logging exceptions and terminating the program. No serious attempts were made to provide recovery actions. This was not surprising. In previous work we have already shown that exception handling mechanisms are not being correctly used as an error recovery tool [33].

After selecting the potential locations on the code where retry semantics could be used we modified the applications to invoke retry from inside exception handlers. Afterwards, we built a fault injector that systematically raises exceptions inside `try` blocks, monitoring the subsequent behavior of the applications.

In our system, after raising an exception, its effect on the program is observed from the injector software. The outcome is stored in a database. The effects are classified in one of three different categories: *Abort* – the application aborts its execution; *Successful* – the application does not abort and its output is the same as an execution with no errors; *Incorrect* – the application does not abort but its output is different from an execution with no errors.

On the unmodified applications, 20% of the raised exceptions did not produce any different observable result compared with a normal execution ("golden run"). Note that this does not mean that the applications did what they should. It only means that their output was identical. 79% of the injected exceptions led to the applications crashing. In 1% of the cases, the output of the applications was different from the one that would be obtained on a non-erroneous execution. The same experiments were then performed using retry. The results are quite compelling: by using simple retry strategy there are less 79% application crashes. The results are shown on Table 1-a.

As we mentioned before, the fact that applications do not produce different results does not mean that they were doing what they should. This is especially true in our set of applications since they mostly send and receive JMS messages. For investigating this issue, three applications were selected where it was easy to monitor if sent messages actually

	Successful	Abort	Incorrect
Original Applications	**44** (20%)	**171** (79%)	**1** (1%)
Modified with retry	**216** (100%)	**0** (0%)	**0** (0%)

(a) Results of retrying the execution of a block vs. the original code.

	Successful	Abort – Correct Delivery	Abort – Incorrect Delivery
Original Applications	**3** (12%)	**6** (24%)	**16** (64%)
Modified with retry	**25** (100%)	**0** (0%)	**0** (0%)

(b) Results of retrying the execution of a block vs. the original code with content checking (3 applications).

Table 1. Comparison between original applications vs. retrying by raising checked exceptions.

reached the destination and their contents were uncorrupted. Again, exceptions were raised, and both the execution of the source application and the contents of messages at the receiving application were examined. The effects of the exceptions that were raised are classified in the following categories: *Abort – Correct Delivery*, the application aborts but the message it tried to send reached its destination correctly; *Abort – Incorrect Delivery*, the application aborts and the message it tried to send either did not reach its destination or its contents were corrupted; *Successful*, the application does not abort and the message reached the destination correctly.

Notice that the "Incorrect" category does not exist in this case since the sending applications do not produce output. Again, on the modified programs we attained a 100% success rate. But, the most interesting result is that in 24% of the cases observed on unmodified applications, although the applications aborted, they were able to send their messages correctly, which was their primary goal. This suggests that with little effort these applications could be much more resilient, which is actually verified by looking at the results of the retry approach. In 12% of the cases the original programs were "Successful" and in 64% "Abort – Incorrect Delivery" was the result. These results are shown on Table 1-b.

Our approach to exception handling has some performance impact. Since each `try` block is transactional, in our implementation, it implies to create a shadow copy of each object that is updated inside a `try` block. Even so, the impact is not as high as one might expect. The reasons are: 1) `try` blocks are normally relatively short and are not deeply nested. This means that the number of objects that are touched inside of a block is typically small. 2) `try` blocks are not normally re-executed. This means that when a block commits, most of the work consists in a small number of checks and on substituting the original object references by the updated copies. Comparing with "normal" STM systems, our case is similar to the situation where the code commits almost every time. This means it can be heavily optimized.

For assessing performance we used the test applications

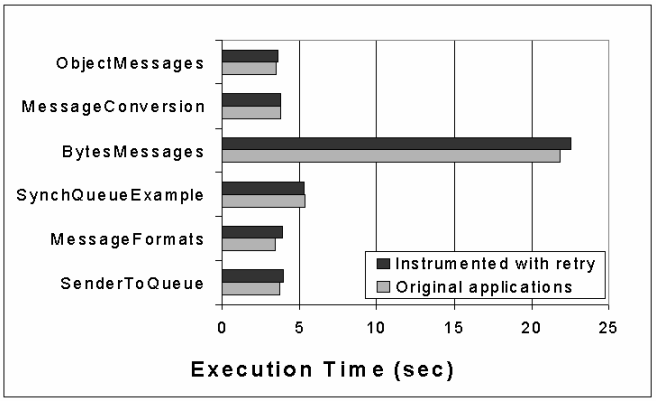

Figure 5. Performance tests.

which could be run in a loop, sending/receiving or processing messages. This corresponded to 6 different applications, where each was run for 100 loops. As it can be seen on the graph of Figure 5, the overall performance impact is negligible. In this graph, "Original" refers to the original applications (without code instrumentations).

5. CONCLUSION

This paper described a process by which AOP can be used to implement retry semantics on systems lacking such a feature. Furthermore, we proposed that retry blocks are made transactional thus eliminating the need for the developer to write specialized code to recover the program's state when something goes wrong during the execution of `try` blocks. Our system is able to transparently rollback the effects of a transaction when entering `catch` handlers.

We also discussed how a minimal retry approach can be quite successful when recovering from some kinds of transient errors (e.g. a momentary glitch on the network; a work peak on a server).

We presented an implementation of the mechanism for the Java 6 JVM that required no changes to the Java language and a minimal tweaking of the Java System Classes. This implementation was tested using several re-written Java programs and validated using a fault injection technique. The results were extremely encouraging. In all cases the system was able to recover from the occurring problems with no side effects. Also, the performance penalty was negligible.

ACKNOWLEDGMENTS

This work was supported in part by Portuguese Science Foundation under Grant SFRH/BD/12549/2003 and by CISUC (R&D Unit 326/97).

REFERENCES

[1] S. Jiang and B. Xu, "An efficient and reliable object-oriented exception handling mechanism", in SIGPLAN Notices, Vol. 40(2), ACM Press, New York, New York, USA, Feb. 2005, pp 27-32.

[2] T. Elrad, R. E. Filman, and A. Bader, "Aspect-Oriented Programming", in Communications of the ACM, Vol. 44(10), ACM Press, New York, New York, USA, October 2001.

[3] N. Shavit, and D. Touitou, "Software Transactional Memory", in Proceedings of the Fourteenth Annual ACM Symposium on

Principles of Distributed Computing, ACM Press, New York, New York, USA, 1995.
[4] A. Goldberg, and D. Robson, "Smalltalk-80: The Language", Addison-Wesley, 1989.
[5] D. Flanagan, Y. Matsumoto, "The Ruby Programming Language", First Edition, O'Reilly, 2008.
[6] B. Meyer, "Object-Oriented Software Construction", Prentice-Hall, 1988.
[7] ISO/IEC, "Information Technology – Programming Languages - C#", 2nd Edition, ISO/IEC, Switzerland, 2006.
[8] M. Lippert, and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming", in Proceedings of the 22nd international Conference on Software Engineering (Limerick, Ireland, June 04 - 11, 2000), ICSE '00, ACM Press, New York, New York, 2000, pp 418-427.
[9] C. Lopes, J. Hugunin, M. Kersten, E. Hilsdale and Gregor Kiczales, "Using AspectJ for Programming the Detection and Handling of Exceptions", in Proceedings of the ECOOP Exception Handling in Object Oriented Systems Workshop, 2000.
[10] J. Viega, J. Vuas, "Can aspect-oriented programming lead to more reliable software?", in IEEE Software, Vol. 17(6), IEEE, Nov.-Dec. 2000, pp19 – 21.
[11] F. C. Filho, A. Garcia, and C. M. Rubira, "Error handling as an aspect", in Proceedings of the 2nd Workshop on Best Practices in Applying Aspect-Oriented Software Development (Vancouver, British Columbia, Canada, March 12 - 16, 2007), BPAOSD'07, Vol. 211, ACM Press, New York, New York, 2007.
[12] F.C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, A., and C.M. Rubira, "Exceptions and aspects: the devil is in the details", in Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Portland, Oregon, USA, November 05 - 11, 2006), SIGSOFT'06/FSE-14, ACM Press, New York, New York, 2006.
[13] F. Filho, C. Rubira, and A. Garcia, "A Quantitative Study on the Aspectization of Exception Handling", in Workshop on Exception Handling in Object-Oriented Systems (held in ECOOP 2005), Glasgow, Scotland, July 2005.
[14] P. Fradet, M. Suholt, "AOP: towards a generic framework using program transformation and analysis", in Proceedings of the Workshop on AOP, ECOOP'98, 1998.
[15] ELCA, Switzerland, 1.3.6 edition, February 2007. URL http://el4j.sourceforge.net/docs/pdf/ReferenceDoc.pdf.
[16] C. Tellefsen, "An Examination of Issues with Exception Handling Mechanisms", MSc thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, June 2007.
[17] R. Laddad, "AspectJ in Action: Practical Aspect-Oriented Programming", Manning Publications Co., Greenwich, CT, USA, 2003.
[18] C. Walls, and R. Breidenbach, "Spring in Action", Manning Publications Co., Greenwich, CT, USA, 2007.
[19] Jimmy Nilsson, "Applying Domain-Driven Design and Patterns: With Examples in C# and .NET", Addison-Wesley Professional, May 8, 2006.
[20] SpringSource, "Spring.NET Framework", 2009, http://www.springframework.net/
[21] M. Pollack, R. Evans, A. Seovic, B. Baia, F. Spinazzi, R. Harrop, G. Caprio, R. Bartelink, C. Rim, The Spring Java Team, "The Spring.NET Framework Reference Documentation", Springs Source, 2009, http://www.springframework.net/doc-latest/reference/html/aop-aspect-library.html
[22] L. Ward, D. Syer, T. Risberg, R. Kasanicky, W. Lund, "Spring Batch - Reference Documentation", SpringSource, 2009, URL http://static.springframework.org/spring-batch/reference/html/index.html
[23] B. Stroustrup, "The Design and Evolution of C++", Addison-Wesley, 1994.
[24] K. Arnold, J. Gosling, and D. Holmes, "The Java Programming Language", 3rd Edition, Addison-Wesley Longman Publishing Co., Inc., 2000.
[25] S. Yemini, D. Berry, "A Modular Verifiable Exception Handling Mechanism", in ACM Transactions on Programming Languages and Systems, Vol. 7(2), 1985.
[26] J. Moss and A. Hosking, "Nested Transactional Memory: Model and Architecture Sketches", in Science of Computer Programming, Vol. 63(2), Elsevier Science, December 2006.
[27] M. Martin, C. Blundell and E. Lewis, "Subtleties of Transactional Memory Atomicity Semantics", in IEEE Computer Architecture Letters, Vol. 5(2), IEEE Press, 2006.
[28] B. Hindman and D. Grossman, "Atomicity via source-to-source translation", in The 2006 workshop on Memory system performance and correctness, pages 82–91. ACM Press, 2006.
[29] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun, "The Atomos transactional programming language", SIGPLAN Notices Vol. 41(6). Jun. 2006, pp 1-13.
[30] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems", in ACM SIGOPS Adaptable and Extensible Component Systems, ACM Press, Grenoble, France, November 2002.
[31] S. Chiba, "Load-Time Structural Reflection in Java", in Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00), Springer-Verlag, LNCS 1850, Sophia Antipolis and Cannes, France, June 2000.
[32] GlashFish Open Message Queue, 2009, http://java.sun.com/products/jms
[33] B. Cabral and P. Marques, "Exception Handling: A Field Study in Java and .NET", in Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP '07), LNCS 4609, Springer-Verlag, 2007.