Timing Failures Detection in Web Services

Nuno Laranjeiro, Marco Vieira, Henrique Madeira
CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
{cnl, mvieira, henrique}@dei.uc.pt

Abstract

Current business critical environments increasingly rely on SOA standards to execute business operations. These operations are frequently based on web service compositions that use several web services over the internet and have to fulfill specific timing constraints. In these environments, an operation that does not conclude in due time may have a high cost as it can easily turn into service abandonment with financial and prestige losses to the service provider. In fact, at certain points, carrying on with the execution of an operation may be useless as a timely response will be impossible to obtain. This paper proposes a time-aware programming model for web services that provides transparent timing failure detection. The paper illustrates the proposed model using a set of services specified by the TPC-App performance benchmark.

1. Introduction

Developing web services with timeliness requirements is a very difficult problem as existing programming models do not provide a practical way to assure any temporal properties, not even the detection of the cases when operations take longer than the expected/desired time. However, real applications very often have to cope with the possible occurrence of timing failures, when the operations executed do not complete within the expected deadlines. Without adequate support to help designers and programmers to solve timing requirements, the development of these applications is a very complex task.

The notion of time is completely absent from the typical web services programming environment. In fact, important features such as timing failure detection or, more generically, timing fault-tolerance, have been completely neglected. The problem is even worse if we consider that web services are typically deployed over wide-area or open environments. Such environments exhibit poor baseline synchrony and reliability properties, thus making it more difficult to deal with timeliness requirements. Obviously, this uncertainty and lack of timeliness will directly affect the execution of web service operations, which, as an immediate effect, will be delayed. However, more severe effects may also be observed on the account of timing failures.

Web services environments can be characterized, essentially, as environments of partial synchrony. In fact, their basic synchrony properties are only cluttered by specific parts of the structure. Several partial synchrony models have been proposed, with different solutions to address application timeliness requirements. The idea of using failure detectors that encapsulate the synchrony properties of the system was first proposed in [7]. The work in [8] introduces the notion of Global Stabilization Time (GST), which limits the uncertainty of the environment. The Timed model, proposed in [9], allows the construction of fail-aware applications, which always behave timely or else provide awareness of their failure. The Timely Computing Base (TCB) model [10] provides a generic framework to deal with timeliness requirements in uncertain environments.

This paper proposes a new model for web services deployment, which allows concurrent detection of timing failures during execution. Clients are able to set a timeout for a given service invocation and will receive a well known exception in case the service execution takes longer than the expected/desired time. A key objective is to reduce as much as possible the required changes both in the web service and client code. Although this may seem to be a simple idea, the design of a generic and non-intrusive mechanism is a complex task, which explains the inexistence of this kind of solutions in real use in the industry. In fact, most temporal failure detection solutions are coupled to particular applications or to specific sets of services (solutions are part of the application itself).

A programming interface implementing this new programming approach is provided. It can be used by web services programmers to easily implement applications with timeliness requirements. All source code, including an example of utilization, is available at [4] for public use.

The structure of the paper is as follows. The following section presents the basis of the timing failure mechanism. Section 3 shows how the mechanism can be used. Section 4 presents the experimental evaluation
Section 5 concludes the paper.

2. Timing failure detection mechanism

The goal of our temporal failure detection mechanism is to provide client applications the possibility of invoking services in a timely manner. Before starting the design of the temporal failure detection mechanism we have identified a set of key properties that need to be fulfilled by this type of mechanisms. These properties represent the quality attributes of the mechanism and are indeed essential to make the mechanism useful in real scenarios. This way, the mechanism must be:

- **Effective**: it must provide low detection latency (i.e., it must be able to detect failures on due time) and must achieve a very low number of false-positives (i.e., the mechanism should be able to detect all failures that are indeed temporal failures).

Our implementation goal, defined based in our experimental knowledge of web service-based environments, was to keep the detection latency below 100ms and the false-positives rate under 5%.

- **Non-intrusive**: the performance overhead must be very low when compared to a similar environment that does not use temporal failure detection. We aimed to keep the performance overhead below 100ms for our initial prototype.

- **Easy to use**: the mechanism must be easy to apply by both providers and consumers, so that developers do not feel overwhelmed by its complexity and refrain from using it. Current development models, or legacy code, must not be significantly changed to use our failure detector.

- **Generic**: the mechanism must be generic in such way that it is eligible to be used in as many environments as possible. However, from the beginning we have targeted our server-side mechanism to be used in Java environments. Despite this, any client built in any programming language is able to use it without any extra special measures.

The Web Services Timing Failure Detection (wsTFD) mechanism is Java based software. In order to make it as transparent as possible to programmers, we opted to code all logic aspects related to the timing failure detection in an isolated package. It can then be merged into any application by using a widely known Aspect Oriented Programming (AOP) [1] framework – AspectJ [2]. This process consists in compiling the candidate application and the wsTFD component (using the AspectJ compiler) into a single Java application that is wsTFD-ready. All wsTFD logic is injected at special points (described further ahead) in the target application. Using AOP involves understanding a few key concepts [3]:

- **Aspect**: a concern that cuts across multiple objects.
- **Join point**: a point during the execution of a program (e.g., the execution of a method or the handling of an exception).
- **Advice**: action taken by an aspect at a particular join point. Types of advice include: ‘before’, ‘after’, and ‘around’ (i.e., before and after).
- **Pointcut**: a predicate that matches join points. An advice is associated with a pointcut expression and runs at any join point matched by the pointcut (e.g., the execution of a method with a certain name).

By using AOP we can inject cross-cutting concerns into any application in a non-intrusive way. For example, it is possible to define a single detection mechanism to be used in many different web service operations in the context of a given application.

wsTFD uses an around advice (so that we have control before and after our join point) and a pointcut that matches all methods that are annotated with @WebService and take a TimeRestriction object as parameter. This object is defined in the context of the wsTFD mechanism and is the one used by clients to specify timeout values (see Section 3 for details on how to use it). With this configuration at the provider side we are able to intercept the web service calls for which we want to perform timing failure detection. At the moment of interception several actions are taken by our framework in order to determine if the execution is on time or if a timing violation has occurred.

Figure 1 illustrates the internal design of wsTFD and the sequence of events that occur at the time of interception of a web service call. The horizontal solid lines represent a thread that is in a runnable state and is performing actual work. Dashed lines represent a thread that is waiting for some event.

At interception time each container thread (parent), that is responsible for serving a particular client request, spawns a new thread (child). This spawned thread is now responsible for doing the actual web service work and placing the final result in its blocking queue. Immediately after the child thread is started, the parent tries to retrieve the result from the child’s blocking queue. As at that starting point no result is available, the parent thread waits during a given time frame (timeout specified by the client application) for an element to become available on the queue. During this period there is no periodic polling of the blocking queue, which reduces the impact of the mechanism. Instead, the parent waits for the occurrence of one of the following events:
A signal to proceed with object removal, which occurs when a put operation is executed over the blocking queue. This operation signals any waiting thread (on that queue) to immediately stop waiting and proceed with object removal (this object corresponds to the result of the web service execution).

The waiting time is exhausted. After the time has expired, the parent thread continues its execution (i.e., leaves the waiting state), ignoring any possible later results placed in the queue. In this case, an exception signaling the occurrence of the timing failure will be thrown to the client application.

There are two types of results that can be expected from the child thread’s execution. The first is a regular result (i.e., the one that is defined as the return parameter in the method signature) and the second an exception being thrown (it can be a checked or unchecked exception). In both cases, an object is placed in the waiting queue (exceptions are caught by the child thread and also put in the queue as regular objects). When the parent retrieves an object from the queue, type verification is performed. For a regular object, execution proceeds and the result will be later delivered to the client. On the other hand, if the retrieved object is an instance of Exception, the parent thread rethrows it, which enables us to maintain program correctness (that would be lost if the child thread did throw the exception itself).

An important aspect is that when a temporal failure occurs an exception is thrown by the container thread to the client. wsTFD offers two types of exceptions:

- **TimeExceededException**: this is a checked exception, i.e., if the provider decides to mark a given public method with a ‘throws’ clause, then the client will be forced to handle a possible exception at compile time (it will have to enclose the service invocation with a try/catch block).
- **TimeExceededRuntimeException**: this is an unchecked exception, i.e., the client does not need to explicitly handle a possible exception that may result from invoking a particular web service operation. This is true even if the provider adds a ‘throws’ clause to its public operation signature.

It is the responsibility of the provider to choose the exception type that better fits his business model.

### 3. Using the timing failure detection mechanism

Using wsTFD implies minimal changes at server and client applications. The usage mode is explained in the following subsections.

#### 3.1. Server side usage

Developing a web service with timing failure detection characteristics is a very easy task. In fact, a developer wanting to provide time-aware web services just needs to execute the following steps:

1. Add the wsTFD library (or source code) to the project. This library and source code is available at [4].
2. Add a TimeRestriction parameter to the web service operations for which timing failure detection is wanted. This object holds a numeric value that is set by clients to specify the desired service duration (in milliseconds).
3. Compile the project using AspectJ’s compiler. For example, when using Maven [5] as a building tool, compilation and packaging can be attained by issuing ‘mvn package’ from the command line.

Figure 2 presents an excerpt of code from the New Customer web service specified by the TPC-App performance benchmark [6] (used in the experimental evaluation in Section 4) and augmented to be a time-aware web service. The wsTFD-related changes are presented in bold. As shown, the changes needed to achieve timing failure detection are quite simple and very easy to understand.

```java
@WebMethod
public NewCustomerOutput newCustomer(
    TimeRestriction restriction,
    NewCustomerInput input) throws SutException {
    // do regular work
}
```

**Figure 2. Server code for a wsTFD-enabled service.**

An interesting aspect is that, although wsTFD targets the web services technology, it enables not only the interception of web services but also other methods the provider may choose, like for instance Remote Method Invocation methods (RMI). The requirement for this is to mark these methods with a `@Interceptable`
annotation (an annotation provided by the wsTFD library). Obviously, these methods will also have to accept a TimeRestriction object as parameter.

3.2. Client side usage

A client wanting to invoke a time-aware service needs to take no special measures, since the only difference from a regular service and a time-aware service, from the client point-of-view, is the use of an extra object that represents the TimeRestriction parameter. This is a regular object that has to be created and set with the desired time value. Figure 3 shows the client code needed to invoke the time-aware service presented in Figure 2. In this example the client is setting a maximum service execution time of 1000ms. Again, the wsTFD-related changes are presented in bold.

NewCustomerInput myServiceInput = new NewCustomerInput();
TimeRestriction restriction = new timeRestriction();
timeRestriction.setTimeInMillis(1000L);
NewCustomer proxy = new NewCustomerService().getNewCustomerPort();
proxy.newCustomer(restriction, myServiceInput);

Figure 3. Client code invoking a time-aware service.

4. Experimental Results

In this section we present and discuss the experimental evaluation performed to assess the properties of wsTFD (i.e., assess if it is effective, non-intrusive, easy to use, and generic). In other words, the experiments presented try to give answer to the following questions:
- Can wsTFD be used without causing additional load in the target system/application?
- Does the mechanism provide low detection latency? What is the latency under heavy workloads?
- Is it able to maintain a low false-positive rate during the detection process?
- Can developers easily use wsTFD in common web service based applications?

In order to give answer to these questions, a large set of tests was executed with our mechanism applied to a set of services defined by the standard TPC-App performance benchmark [6]. This is a performance benchmark for web services and application servers, that is widely accepted as representative of real environments. A subset of the web services specified by TPC-App (Change Payment Method, New Customer, New Product, and Product Detail) was used to illustrate the capabilities of our mechanism. It is important to refer that the Change Payment Method and the New Customer services include an external service call that simulates a payment gateway. We added a variable delay of 1000 to 2000 ms to this service’s invocation in order to capture the variable delay that frequently occurs when using services over the internet.

4.1. Experimental setup

The setup for the experiments consisted in deploying the three main test components into three separate machines connected by a Fast Ethernet network. These components are:
1) A web service provider application that provides the set of web services used in the experimental evaluation.
2) A database server on top of the Oracle 10g Database Management System (DBMS). The TPC-App benchmark simulates an on-line store whose services make use of a database.
3) A workload emulator that simulates the execution of business transactions by multiple clients (i.e., that performs web services invocations).

Table 1 describes the software and hardware used in each of these three nodes.

Table 1. Systems used for the experiments.

<table>
<thead>
<tr>
<th>Node</th>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Windows server 2003 R2 Enterprise x64 Edition service pack 2 &amp; JBoss 4.2.GA</td>
<td>Dual Core Pentium 4, 3Ghz, 1.46GB RAM</td>
</tr>
<tr>
<td>DBMS</td>
<td>Windows server 2003 R2 Enterprise x64 Edition service pack 2 &amp; Oracle 10g</td>
<td>Quad Core Intel Xeon 5300 Series, 7.84 GB RAM</td>
</tr>
<tr>
<td>Client</td>
<td>Windows XP pro SP2</td>
<td>Dual Core Pentium 4, 3GHz, 1.96GB RAM</td>
</tr>
</tbody>
</table>

4.2. Results and Discussion

To answer the questions presented above, we executed 36 tests with different configurations. Each test had a duration of 20 minutes and was repeated three times to increase the results representativeness. Between each run, the system state was restored so that tests could be performed in the same starting conditions.

Our first experimental campaign targeted performance overhead, and so we ran 2 experiments, one with a plain server application not using wsTFD and another using a wsTFD-enabled application (experiments A and B). The goal was to collect baseline performance measures from experiment A and then compare them with the performance measures extracted from experiment B. Each of these experiments in-
cluded 4 sets of tests (each set consists of 3 executions of a given test). In these tests the client application was configured to emulate 2, 4, 8, and 16 business clients (one configuration per set of tests). We used these configurations to expose the system to different service loads and extract more realistic measures. For experiment B we have defined a high constant value for the execution timeout. The goal was to have no particular service being aborted by our mechanism (i.e., no timing exceptions thrown, although the detection mechanism was in place checking time usage). Figure 4 presents a summary of the results obtained in this first experimental campaign.

For each web service, we calculated the average service duration (of 3 runs) in each set of tests (i.e., 2, 4, 8 and 16 clients) in experiment A (baseline) and B (wsTFD-enabled). We then subtracted the average duration values and extracted the minimum, maximum and an average duration.

As we can see by analyzing Figure 4, the average overhead values remain fairly the same and around 20ms. This is a low value, particularly considering that web services environments are characterized by high execution times. Nowadays it is frequent for a web service to behave as a client of another web service (i.e., nested calls are frequent in web service compositions), and it is also frequent to find this target service in another geographic location: This often requires the request to travel over the internet, increasing this way the total execution time of the web service. The minimum values were quite low, from 1 to 5 milliseconds (we used nanosecond precision, provided by Java 6, for all time measurements) and the maximum values were in the range of 38 to 56 milliseconds. These are values that we also find quite acceptable as explained. Furthermore, these values fully comply with the objectives we traced before the beginning of the experiments by being clearly below the 100 ms mark (see Section 2).

Our second experimental campaign aimed to quantify the detection latency of our mechanism (i.e., the difference between the expected time set by the client and the real detection time). Since we were expecting minimal latency values, we decided to add load to the server by modifying the TPC-App implementation. We added 2 extra threads in an infinite loop, which was enough to push the CPU usage of our dual core processor to 100%. In order to measure the detection latency we needed to configure the services invocation with an acceptable service duration. This duration needed to be lower than the maximum service execution time, so that our detection mechanism was triggered and we could measure the detection latency. We opted for a variable duration, in order to emulate real environments more accurately (in real environments clients will be requiring different service durations). Using the baseline experiments as reference, each client was given the possibility of randomly choosing the deadline for each service execution from an interval ranging from the service minimal historical execution time to twice the historical average execution time of the same service. With these values we are able to mix services that finish in a timely fashion with services that exceed the clients’ expectations and are terminated by our mechanism by raising a timing exception.

Considering this environment, we then ran 4 sets of tests (2, 4, 8, and 16 clients) in a single experiment (experiment C). The results in Figure 5 show that the latency steadily increases as we increase the client load. Keep in mind that, with the aim of extracting meaningful values, these experiments were performed in an extremely stressed environment with a constant CPU load of 100%, which is a harsh condition for any application. The important aspect is that, the detection latency did not exceed 50 ms which we find to be very good results for these conditions. These results fulfill the traced objective of achieving a detection latency of less than 100ms (see Section 2).

To get a clearer view of the detection latency, we selected 50 random values from one of the tests executed with 16 clients (the New Products web service in experience C). These are displayed in Figure 6. It is clear that, in the majority of these random cases, detection follows the expected time very closely. There are 3 cases (marked by the arrows) where this does not happen, which is probably related with transient
conditions, thread scheduling, etc., that only reflect the non-deterministic behavior of software.

An interesting aspect is that, in a preliminary version we used millisecond precision for the failure detection mechanism implementation. This led us to a low rate of false-positives, i.e., failures being detected before the actual service deadline. With these results in hand, we switched to nanoseconds precision (by using a Java method that however does not guarantee nanosecond accuracy), and in all tests no failure detection was triggered before the service deadline. All detected timing failures were indeed failures which represents a false-positive rate of 0%, an excellent result for wsTFD that clearly exceeds our expectations of achieving a false-positive rate of less than 5%.

We had a TPC-App implementation prior to these experiments, and we wanted to check if a moderately experienced programmer was able to easily integrate wsTFD into this TPC-App implementation. Since the application was already using Maven as build tool, the programmer’s tasks were reduced to merging the application’s project descriptor (a Maven configuration file) with the one provided by wsTFD, and adding a TimeRestrictionParameter to each web service. The main task executed in the merging process was the replacement of the standard Java compiler with AspectJ’s compiler. The whole process was concluded in about 5 minutes, indicating that it is a fairly easy procedure.

5. Conclusion

This paper discussed the problem of timing failure detection in web service applications and proposes a programming approach to help developers in programming web service applications with time constraints. The approach proposed implements the timing detection in a transparent way.

The results extracted from the several experiments executed have shown that wsTFD is an effective mechanism, that provides low detection latency and low false-positive rates (none in our experiments). It also has a low performance impact on web service-based applications. Additionally, it is a generic mechanism that can be easily fit into other technologies and requires a low integration effort from the developer.

As future work we intend to augment this work to include failure prevision information, a particularly useful feature in web service based environments.

5. References