

Predicting Timing Failures in Web Services

Nuno Laranjeiro, Marco Vieira, and Henrique Madeira

CISUC, Department of Informatics Engineering,
University of Coimbra, Portugal
{cni, mvieira, henrique}@dei.uc.pt

Abstract. Web services are increasingly being used in business critical environments, enabling uniform access to services provided by distinct parties. In these environments, an operation that does not execute on due time may be completely useless, which may result in service abandonment, and reputation or monetary losses. However, existing web services environments do not provide mechanisms to detect or predict timing violations. This paper proposes a web services programming model that transparently allows temporal failure detection and uses historical data for temporal failure prediction. This enables providers to easily deploy time-aware web services and consumers to express their timeliness requirements. Timing failures detection and prediction can be used by client applications to select alternative services in runtime and by application servers to optimize the resources allocated to each service.

Keywords: web services; timing failures; detection; prediction.

1 Introduction

Web services provide a simple interface between a provider and a consumer and are increasingly becoming a strategic vehicle for data exchange and content distribution [1]. Compositions, which are based on a collection of web services working together to achieve an objective, are particularly important. These compositions are normally defined at programming time as "business processes" that describe the sequencing and coordination of calls to the component web services. Calls between web service consumers and providers consist of messages that follow the SOAP protocol, which, along with WSDL and UDDI, form the core of the web services technology [1].

Developing web services able to deal with timeliness requirements is a difficult task as existing web services technology, programming models, and development tools do not provide easy support for assuring timeliness properties during web services execution. Although some transactional models provide basic support for detecting the cases when operations take longer than the expected/desired time [2], this usually requires a high development effort. In fact, developers have to select the most adequate middleware (including a transaction manager that must fit the deployment environment requirements), produce additional code to manage the transactions, specify their properties, and implement the basic support for timing requirements. Transactions are actually well suited for supporting typical transactional behavior, but they are inadequate for deploying simple time-aware services. Indeed, transactions provide

poor support for timing failures detection and no support for prediction.

Despite the lack of mechanisms and tools for building time-aware web services, the number of real applications that have to support this kind of requirements is quickly increasing. Typically, developers deal with these by implementing ad-hoc solutions to support timing failures (this is, obviously, expensive and prone to fail). The concept of time has been, in fact, completely absent from the standard web services programming environment. Important features such as timing failure detection and forecasting have been overlooked, although these are particularly important if we consider that services are typically deployed over wide-area or open environments that exhibit poor baseline synchrony and reliability properties. In these environments it is normal for services to exhibit high or highly variable execution times. High execution times are usually associated with the serialization process involved in each invocation, coupled with a high amount of protocol information that has to be transmitted per each payload byte (e.g., the SOAP protocol requires a large amount of data to encapsulate the useful data to be transmitted). This serialization process is particularly important since a given web service can also behave as a client of another service, thus duplicating the end-to-end serialization effort. Variable execution times are essentially related to the use of unreliable, sometimes slow, transport channels (i.e., the internet) for client-server and inter web services communication. These characteristics make it difficult for developers to deal with timeliness requirements.

Two outcomes are possible when considering timing requirements during a web service execution: either the server is able to produce an answer on due time, or not. The problem is that, in both cases the client application has to wait for the execution to complete or for the deadline to be violated (in this case a timing failure detection mechanism must be implemented). However, in many situations it is possible to predict in advance the occurrence of timing failures. In fact, execution history can typically be used to confidently forecast if a timely response will be possible to obtain. Note that, this is of utmost importance for client applications, that can retry or use alternative services, but also for servers, that can use this information to conveniently manage the resources allocated to each operation (e.g., an operation that is predictably useless can be canceled or proceed executing under a degraded mode).

This paper proposes a new programming model for web services deployment that allows online detection and prediction of timing failures (wsTFDP: Web Services Timing Failures Detection and Prediction). By using this model clients are able to express their timeliness requirements for each service invocation by defining a timeout value and an associated confidence value for prediction. When timing requirements are not possible to satisfy (e.g., because the deadline was exceeded or because it will predictably be exceeded) the server responds with a well-known and consistent exceptional behavior. A simple and ready to use programming interface implementing the proposed model is also provided.

The structure of the paper is as follows. The following section presents background and related work. Section 3 presents a high level view of the timing failures detection and prediction mechanism. Sections 4 and 5 detail the design of the detection and prediction components. Section 6 shows how the mechanism can be used in practice, and Section 7 presents some experimental results. Section 8 concludes the paper.

2 Background and Related Work

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 system. 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 [3]. The work in [4] introduces the notion of Global Stabilization Time (GST), which limits the uncertainty of the environment. The Timed model [5] allows the construction of fail-aware applications, which always behave timely or else provide awareness of their failure. The Timely Computing Base (TCB) model [6] provides a generic framework to deal with timeliness requirements in uncertain environments. In [7] we proposed a programming approach to help developers in programming time-bounded web service applications. The approach proposed implements timing detection (prediction is not addressed) in a transparent way.

Several works on prediction can be found in the literature. In [8] a Support Vector Machine-based model for software reliability prediction is proposed and issues that affect the prediction accuracy are studied. These issues include: whether all historical failure data should be used and what type of failure data is more appropriate to use in terms of prediction accuracy. The accuracy of software reliability prediction models based on SVM and artificial neural networks are also compared in this work.

Local prediction schemes only use the most recent information (and ignore information on far away data). As a result, the accuracy of local prediction schemes is limited. Considering this, in [9] it is proposed a novel approach named Markov–Fourier gray Model. The approach builds a gray model from a set of the most recent data and a Fourier series is used to fit the residuals produced. Then, the Markov matrices are employed to encode possible global information generated also by the residuals. The authors concluded that the global information encoded in the Markov matrices can provide quite useful information for predictions.

In [10] it is presented a framework that incorporates multiple prediction algorithms to enable navigation of autonomous vehicles in real-life, on-road traffic situations. At the lower levels, the authors use estimation theoretic short-term predictions via an extended Kalman filter-based algorithm that uses sensor data to predict the future location of moving objects with an associated confidence measure. At the higher levels, the authors use a long-term situation-based probabilistic prediction using spatiotemporal relations and situation recognition. The authors also identify the different time periods where the two algorithms provide better estimates and demonstrate the possibility of using results of the short-term prediction algorithm to strengthen/weaken the estimates of the long-term prediction algorithm.

A key aspect to retain from these works is that, both recent and older historic data can be valuable when applying a particular prediction algorithm. In this sense, a prediction mechanism must include support for considering different periods of historical data when trying to predict a timing failure. This particular support should be at least configurable on the server-side. On some applications it may also be important to provide a client-side selection of the historical to be considered on each prediction. This, of course, depends on the particular goals of the software being deployed.

3 Timing Failures Detection and Prediction Mechanism

The Web Services Temporal Failure Detection and Prediction (wsTFDP) mechanism builds on top of the failure detection mechanism we proposed in [7]. Besides failure detection, wsTFDP enables us to collect accurate runtime failure data that is later used for failures predicting. To be useful in real scenarios, a timing failures detection and prediction mechanism must achieve a key set of quality attributes. Thus, before developing the mechanism we have established the following objectives:

- **Low latency:** timing failures must be detected within small latency intervals. In other words, failures must be detected on time, or the detection process itself may become of little or no use. Our goal was to design the mechanism in such way that guarantees detection latency under 100ms, a reasonable value when considering the typically high execution times of web services.
- **High accuracy:** the number of false-positives must be very low. A low false-positive rate means that, in few cases, the mechanism detects or predicts a timing failure that, in reality, will not occur. On the other hand, the number of false-negatives (i.e., failures that are not predicted) must also be kept low. A system that fails less in identifying or predicting failures is obviously preferable. Based on our experimental knowledge, we aimed to provide an integrated mechanism able to maintain at least one of the rates under an average of 5%. We opted to define a single limit as lowering one of these two measures typically results in increasing the other.
- **Low performance overhead:** a time-aware service obviously performs more computational work than a basic service. For our initial prototype, we aimed to achieve a maximum overhead of 100ms in terms of response time (when compared to the equivalent service without timing characteristics).
- **Easy usage:** code, tools, or deployment complexity and non-portability are among some of the characteristics developers try to avoid nowadays. It is important that developers, either consumers or providers, do not have to make significant changes to existing code in order to use the mechanism. In the same way, when developing new services, the development model must be maintained as similar as possible to the common practice.
- **Generic:** the mechanism must be generic so that it can be reused, as much as possible, in different environments. Our goal is to provide extension features so that wsTFDP can be used outside the scope of web service applications, e.g., in Remote Method Invocation methods (RMI).

wsTFDP consists of a detection component and a prediction component. The detection component performs two basic functions: it detects timing failures transparently in runtime and collects timing data about web service execution (this data is basically a profile of the service's execution time). The prediction component uses this historical data to predict timing failures by monitoring the current execution point of a service and performing a pessimistic analysis of the estimated duration of the remaining path. When a failure is predicted the execution is aborted or continued under a degraded mode. In both cases, the client is immediately informed that an exceptional behavior has occurred.

A difficult aspect is the integration of this kind of mechanisms with current applications and development models. In order to make the wsTFDP as transparent as possible to programmers, we decided to deploy it as an isolated package that holds all logic aspects related to timing failures detection and prediction and that can be merged into any application by using regular Aspect Oriented Programming (AOP) tools [11]. This process consists in compiling the candidate application and the wsTFDP component (using an AOP framework compiler) into a single application that is time-aware ready. All wsTFDP logic is automatically injected at particular points in the target code (described further ahead). Using AOP involves understanding a few key concepts, namely (see [12] for details):

- **Aspect:** a concern that cuts across multiple objects.
- **Joinpoint:** 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 joinpoint. Types of advice include: ‘before’, ‘after’, and ‘around’ (i.e., before and after).
- **Pointcut:** a predicate that matches joinpoints. An advice is associated with a pointcut expression and runs at any joinpoint matched by the pointcut (e.g., the execution of a method with a certain name).

By using AOP we can inject crosscutting concerns into any application in a way that avoids writing extra code to execute these same crosscutting concerns. wsTFDP uses an around advice (so that we have control before and after our join point) and a pointcut that matches all methods that correspond to a web service invocation and take a *TimeRestriction* object as parameter (see Section 5.1 for further details on the interception). This object is defined in the context of the wsTFDP mechanism and is the one used by clients to specify deadlines and confidence values (see Section 6 for details on the object including its usage mode). With this configuration it is possible to intercept the web service calls for which users want to perform timing failure prediction. At the moment of interception several actions are taken by our framework to determine if the execution is on time, if a timing violation has occurred, or if it is expectable that a timing violation will occur.

Despite the key concepts behind wsTFDP are completely generic, our initial prototype is Java-based, for practical reasons only (implementing the mechanism using another language does not impose any relevant technical difficulty). Note that this is relevant only at the server-side. In fact, any client using any programming language with support for web services can use a time-aware web service (even if this service is implemented in Java) with no extra special measures.

4 Detection Mechanism Design

As previously referred, we have presented a preliminary version of the detection component of wsTFDP in [7]. For the sake of completeness we present here a brief description of its operating mode.

Figure 1 illustrates the internal design of the detection component 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.

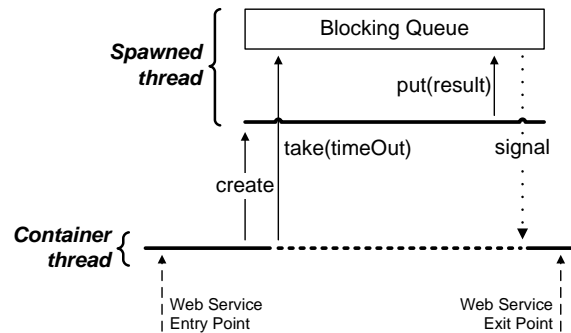


Fig. 1. Temporal failure detection mechanism.

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 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 is delivered to the client. On the other hand, if the retrieved object is an instance of `Exception`, the parent thread re-throws it, which enables us to maintain program correctness (that would be lost if the child thread did throw the exception itself).

The detection component offers two types of exceptions (it is the responsibility of the provider to choose the type that better fits his business model):

- **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.

5 Prediction Mechanism Design

In order to predict timing failures, wsTFDP executes the following set of steps:

- 1) Analyzes the service code and builds a graph to represent its logical structure;
- 2) Gathers time-related performance metrics in a transparent way during runtime;
- 3) Uses historical data to predict, with a degree of confidence chosen by the client, if a given execution will or will not conclude in due time.

5.1 Graph Organization

wsTFDP analyzes the logical structure of web services and automatically **organizes a graph structure** for each operation provided to clients. A graph consists of vertexes (or nodes) and edges that connect nodes. Each edge represents a connection between two nodes (i.e., an edge provides a path between two nodes, which in our case is unidirectional) that has an associated cost (i.e., travelling between two nodes involves a predefined cost) [13]. This data structure perfectly fits the requirements of our mechanism. To build the graph we have to define what information will constitute the nodes, edges, and costs:

- **Nodes:** specific instants in a service’s execution where timing failures should be predicted. Natural candidates are the invocation of the target service itself and all nested service invocations (if any) performed by the service. Additionally, it is important to add support for user-identified critical points that allow the developer to instruct wsTFDP to add specific code parts to the graph.
- **Edges:** these naturally represent the available connections between nodes, and are automatically defined by wsFTP based on a runtime analysis of each service.
- **Cost:** for our goals, the cost involved in travelling between two nodes is the execution time in milliseconds.

Figure 2 presents an example of a web service augmented to be time-aware and the respective graph organization (source code in bold is specific to our framework). The arrows connecting the nodes indicate unidirectional relations between those nodes. That is, it is possible to go from the ‘Service C’ node to the ‘DB query’ (database query) node but not the other way around, which of course accurately represents what the programmer intended when writing this particular piece of source code. This service uses two wsTFDP methods: *check* and *ignore*. The *check* method indicates that that specific point in the code is critical and must be included as a node on the graph

and a point for timing failures prediction. The *ignore* method indicates that the following web service call should not be considered as a node when building the graph and also when predicting timing failures. It is the responsibility of the programmer to identify critical execution points that should take part of the runtime graph. This task can be easily achieved with a regular profiling tool. For example, the invocation of ‘Service D’ (represented by the *invokeServiceD()*; statement) is not included in the graph as it was explicitly ignored by the programmer.

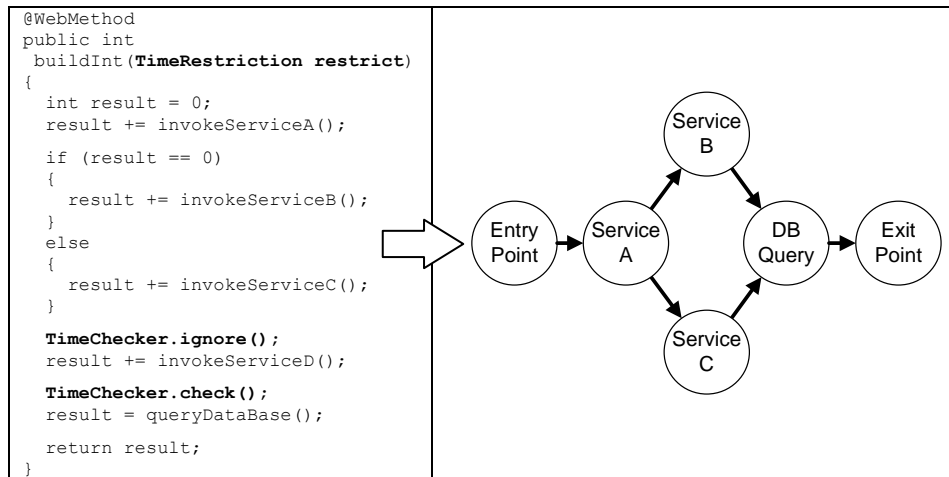


Fig. 2. Transformation from source code into a graph structure by wsTFDP.

The graph structure is organized during regular service execution. Instead of implementing a fairly complicated approach for compile-time path analysis, we use AOP (and AspectJ [14] in particular for our prototype) during service execution. In Java, a web service can be invoked by executing a generated method that is marked with the *@WebMethod* JAX-WS annotation [15]. In order to build the graph, wsTFDP assumes that the provider is using any implementation form of JAX-WS, which is ever more frequent (e.g., JBoss provides a JAX-WS compliant implementation, such as many major middleware vendors). In our case, AspectJ is instructed to intercept calls to *@WebMethod* annotated methods and uses this interception information to build the service graph. As mentioned before, wsTFDP intercepts all external service calls (i.e., nested calls) by default. However, the mechanism can also be configured to intercept only specific (i.e., marked by the developer) external service invocations, ignoring the remaining.

5.2 Metrics Management

Time-related metrics are collected by the detection component and stored in the graph’s edges. In each run, wsTFDP stores information that expresses how long it takes to travel from a preceding node to the current node (in terms of execution time). In practice, each edge maintains a list of historical values sorted from the shortest to the longest time. This list is updated in each web service call with the observed value.

Obviously, as the maximum size of this list must be limited, the provider has to configure it according to the specificities of the web services being monitored. In environments that experience small variations in execution time a small list may be sufficient, while other type of environments may require larger lists that can capture the natural execution time variation of the web service. When considering the mechanism's performance smaller lists are preferable, since the sorting algorithm, required by our pessimistic approach for failure prediction, will obviously perform faster in smaller than in larger lists (see Section 5.3 for details on the prediction process).

An important aspect is that the graph will be quite incomplete during the first executions of the web service. Typically, web services can be rather complex and several calls may be needed to explore all possible paths, and hence build a complete graph. The more paths are explored, the more accurate prediction becomes.

5.3 Prediction Process

At each node wsTFDP subtracts the elapsed time (counted from the moment execution started) from the maximum amount of time specified by the client application (which represents the total available time for execution). It then uses the Dijkstra's shortest path computation algorithm [16], which makes use of the best (fastest) time values stored on the graph's edges, to determine if execution can terminate on time. If history indicates that the amount of time left in a given service invocation is not enough to complete execution in a timely fashion, then it is fairly safe to return a well known prediction exception to the client.

When a timing failure is predicted, two outcomes are possible: the server aborts the operation or the server continues the operation under a degraded mode. The former should be used in the cases where it is safe to artificially abort a particular operation. The latter mode is useful for those cases where consistency is needed on the server side (i.e., an initiated operation must not be artificially aborted, as the application may not be prepared to deal with this situation). These two modes are particularly important to providers since they both ease the load experienced by servers when timing failures occur. Obviously, it is up to the provider to decide which mode better fits the application being deployed.

An important aspect is that there is always some degree of confidence involved in a prediction process. In wsTFDP, this degree of confidence can be manipulated by the provider or by the consumer in the following ways:

- When the list that stores execution times achieves its maximum in a given edge, it is necessary perform elements removal to accommodate new historical data. The **provider** can configure two different removal strategies. If accurate prediction is critical, then the provider should opt by dropping the longest duration values. This will make the prediction much more accurate since it uses a pessimistic approach (the algorithm always uses the fastest values on each edge). The other option is to do a random removal. In this case the mechanism may lose some accuracy (which in many cases is not highly relevant to clients), but it is able to capture the web service typical behavior in a better way.
- In runtime the **consumer** can discard a given percentage of the lowest execution times that exist in the collected history (i.e., discard the fastest values). This is quite useful to express an amount of confidence over the collected values. For

example, if the consumer considers that 5% of the fastest executions happened in very particular conditions that do not capture the normal behavior of the web service and its related resources, then those 5% of values should be discarded. This represents a 95% confidence degree for timing failures prediction (see Section 6 for more details on how to express this confidence value at the client-side).

As in the detection mechanism, both checked and unchecked prediction exceptions are available. These are, respectively, *FailurePredictionException* and *FailurePredictionRuntimeException*. An important aspect is that these exceptions are hierarchically organized and extend a superclass (*TimeFailureException*) also available in unchecked and checked versions. This enables clients to handle both detected and predicted timing failures in a single catch block.

The complete procedure for detecting and predicting timing failures is relatively straightforward. However, it is important to notice that, despite being a simple approach, it is able to produce very good results that can be used for service selection and resource management, with clear advantages for clients and servers respectively (see Section 7 for details on the experimental results).

6 Using the timing failure detection mechanism

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

- 1) Add the wsTFDP library (or source code) to the project. The library and source code are available at [17].
- 2) Add a *TimeRestriction* parameter to the web service operations for which timing failure detection and prediction 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 an AOP compiler (AspectJ, in the case of our prototype). For example, when using Maven [18] as a building tool, compilation and packaging can be done by executing the following command from the command line: ‘mvn package’.

As we can see in Figure 2 (Section 5.1), the wsTFDP-related changes (presented in bold) are quite simple. In fact, the code needed to include timing failure prediction in a web service is quite straightforward and very easy to understand. In its basic form, the provider only needs to add an extra *TimeRestriction* parameter to the web service and our framework will perform all remaining tasks automatically. An interesting aspect is that, although wsTFDP targets web services technology, it is also able to intercept other methods (fulfilling this way the ‘to be generic’ requirement presented in Section 3), like for instance Remote Method Invocation methods (RMI). To achieve this, the programmer should mark these methods with a *@Interceptable* annotation (an annotation provided by the wsTFDP library). Obviously, these methods will also have to accept a *TimeRestriction* object as parameter.

No special measures are needed to **invoke a time-aware service**. In fact, the only

difference between a regular service and a time-aware service, from the client point-of-view, is the use of a regular extra object – the *TimeRestriction* parameter. This has to be created and set with a desired time value and a prediction confidence factor.

Figure 3 shows the client code needed to invoke a time-aware service. The service in question is an implementation of the ‘New Customer’ web service specified by the TPC-App performance benchmark [19] and augmented to be time-aware. In this example the client is setting a maximum service execution time of 1000ms, and choosing a 95% confidence factor for historical data. The wsTFDP-related changes are presented in bold.

```
NewCustomerInput myServiceInput = new NewCustomerInput();
TimeRestriction restriction = new timeRestriction();
timeRestriction.setTimeInMillis(1000L);
timeRestriction.setConfidenceValue(0.95D);
NewCustomer proxy = new NewCustomerService().getNewCustomerPort();
proxy.newCustomer(restriction, myServiceInput);
```

Fig. 3. Client code for invoking a time-aware service.

7 Experimental Evaluation

In this section we present the experimental evaluation performed to assess the effectiveness of wsTFDP. The experiments performed try to give answer to the following questions:

- Does the mechanism introduce a noticeable delay in services?
- How fast can the mechanism detect failures and how does this detection latency evolve under higher loads?
- Is it able to provide low false-positive and false-negative rates during the detection and prediction process?
- Can programmers easily use the mechanism?

7.1 Experimental Setup

The TPC-App benchmark was used as test case [19]. This is a performance benchmark for web services and application servers widely accepted as representative of real environments. We applied our mechanism to a subset of the services defined by this benchmark (Change Payment Method, New Customer, New Product, and Product Detail). The *Payment Method* and the *New Customer* services include an external service call that simulates a payment gateway. In order to have a more realistic representation of what usually happens when invoking services over the Internet, and based in our empirical knowledge of common service execution times, we introduced a variable delay of 1000 to 2000 ms whenever the payment gateway was contacted.

The setup for the experiments consisted in deploying the three main test components into three separate machines connected by a Fast Ethernet network (see Table 1). 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) used by the TPC-App benchmark; and 3) a workload emulator that simulates the concurrent execution of business transactions by multiple clients (i.e., that performs web services invocations).

Table 1. Systems used for the experiments.

Node	Software	Hardware
Server	Windows server 2003 R2 Enterprise x64 Edition service pack 2 & JBoss 4.2.2.GA	Dual Core Pentium 4 3Ghz 1.46GB RAM
DBMS	Windows server 2003 R2 Enterprise x64 Edition service pack 2 & Oracle 10g	Quad Core Intel Xeon 5300 Series 7.84 GB RAM
Client	Windows XP pro SP2	Dual Core Pentium 4, 3GHz, 1.96GB RAM

7.2 Results and discussion

To analyze the effectiveness of our mechanism, we executed 36 tests with different configurations. In all tests, wsTFDP was configured to predict at all service invocation points (i.e., the target service entry point and any existing nested service invocations). Each test had a duration of 20 minutes and was executed three times to increase the results representativeness. We ran tests considering different service loads (i.e., 2, 4, 8, and 16 simultaneous clients) and the system state was restored between each run, so that tests could be performed based on the same starting conditions. Time measurement was done using nanosecond precision, provided by Java 6.

A first set of tests was conducted to assess the **performance overhead**. We ran 2 experiments, one without wsTFDP and another using wsTFDP (baseline experiment A and experiment B, respectively). Each of these experiments included 4 sets (correspond to a client-side configuration of 2, 4, 8, and 16 emulated business clients) of 3 tests (3 executions of a given configuration). Our goal was to test the system using different service loads to get meaningful measurements. For experiment B we defined a high deadline for the execution so that we would not have any service being aborted due to timing failures (we just wanted to assess the performance overhead of the mechanism, when compared to the normal conditions of experiment A). The results obtained are presented in Figure 4.a).

For each of the four web services, we calculated the average execution time (of 3 runs) in each set of 4 tests (2, 4, 8 and 16 clients) for experiments A and B. We then extracted the minimum, maximum and average differences between experiments in each set. We finally averaged the extracted differences of each of the four web services in a single value per set. As expected, as the number of clients increases, the overhead of the detection and prediction process also increases. However, the overhead remains quite low, with an observed maximum of about 56 ms, which perfectly fits our initial goal of keeping it under 100ms for moderately loaded environments. Note that the introduced overhead is tightly associated with the complexity of running Dijkstra’s algorithm over the graph. It is well known that when the graph is not fully connected, which is frequently the case in this type of applications (i.e., each node is not connected to all remaining nodes), the time complexity is $O((m + n) \log n)$ where ‘m’ is the number of edges and ‘n’ is the number of vertices in the graph (for algorithms that make use of a Priority Queue as in our case) [16].

The second experimental goal was to assess the **detection latency** (i.e., the time

between the failure occurrence and its detection). As we were expecting very low values for these experiments, we decided to add extraneous load to the server by creating two threads in the server application, running in a continuous loop. This pushed the CPU usage of our dual core machine to 100%.

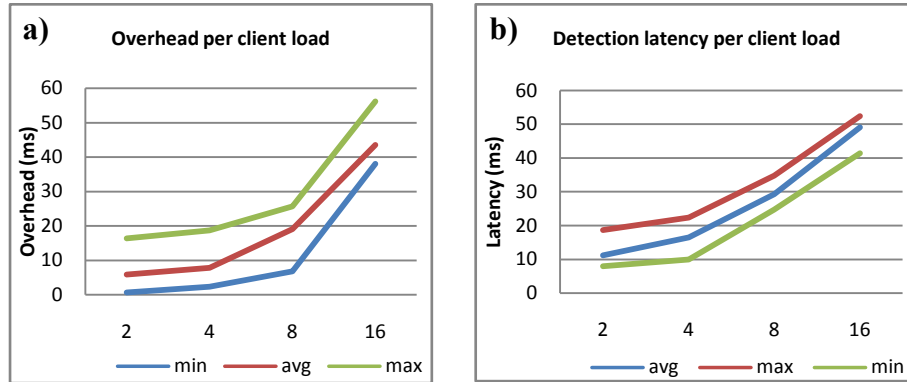


Fig. 4. wsTFDP mechanism overhead a) and detection latency b) per client load.

Based on the baseline reference values, we configured clients to request random service deadlines ranging from the observed average execution times and twice this value. Our goal was to mix services that finish on time with services that violate the client-set deadline and are terminated by wsTFDP, hence enabling us to measure the detection latency. As shown in Figure 4.b) four sets of tests (2, 4, 8, and 16 clients) were executed (experiment C) under the referred stress conditions. As expected, we observed increasing latency values as we added load to the server. The maximum observed latency was of 52 ms, which fulfills our initial objective. The extracted results are, in general, very good, particularly when considering the demanding environment used in this experiment (i.e., the extraneous load in the server).

In the third experiment we tried to assess the **false-positive** (i.e., the number of times wsTFDP predicted a failure that did not occur) and **false-negative** (i.e., the number of times a failure was detected but not predicted) rates. An important aspect is that, these two measures can be considered as conflicting, in the sense that, when we tune the mechanism to minimize the number of wrong predictions we increase the number of failures that may occur without being predicted. Balancing both measures can be a hard task and it is up to the user to choose if a balanced approach is preferable, or if one particular measure is the most important. In our particular case, we executed a single experiment (experiment D) and tried to minimize the false-positives, assuming that a wrong prediction has a higher cost than a failure that is not predicted. Note that, even if a timing failure is not predicted, it will be for sure detected (the detection coverage observed in all experiments performed was of 100%).

We performed some preliminary tests to check which configuration parameters would provide the lowest false-positive rates. Note that, we did not intend to execute an exhaustive search over all the available configurations, but merely aimed to choose a good enough configuration that enabled a solid demonstration of our mechanism. In real situations, users may perform a more thorough experiment according to their specific needs. We maintained the same client-side selection of deadline values (rang-

ing from the average observed values to twice the average) and empirically we observed that, for our goals, an 80% of client-set confidence value provided the best results (we tested all lower and higher values in 5% increments, with worse results). On the provider side, the main configuration was related to the graphs edges management. We opted for a maximum list size of 100 elements and also for a random element removal strategy (see Section 5.3 for details on the available removal strategies). Figure 5 presents the obtained false-positive rates for the executed tests.

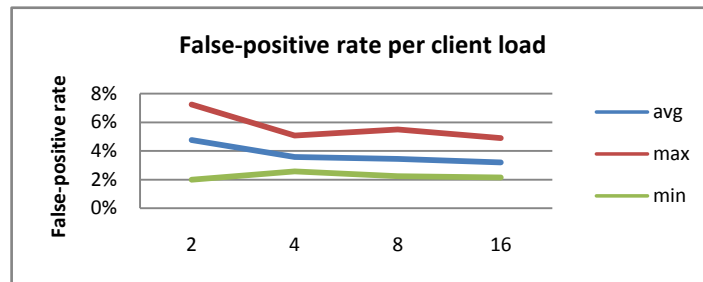


Fig. 5. The observed false-positive rate under different client loads.

As we can see, with this moderate configuration tuning we managed to maintain the average false-positive rate under 5%, which fulfills our initial goals. This is an excellent result if we consider that we did not perform exhaustive configuration tweaking. An interesting aspect is that, the false-positive rate generally decreases as we increase the number of clients. This is related with the fact that, under heavy loads more operations are executed and, in consequence, history comprehends a time-frame that is smaller and closer to the current environment, representing it more accurately.

Concerning the false-negatives, we observed a total of 26% considering all tests performed in experiment D. This is quite acceptable, as our main goal was to have a low false-positive rate (which for sure increases the false-negatives rate). Users that prefer more balanced results can fine-tune the wsTFDP mechanism, for instance by increasing the history size, decreasing the confidence value at the client-side or testing multiple combinations of the different available parameters.

To check the easiness of use of the wsTFDP mechanism we asked an external programmer to implement time-awareness into our reference TPC-App implementation. Since the application was already using Maven as building tool, the programmer's tasks were reduced to merging the application's project descriptor (a Maven configuration file) with the one provided by wsTFDP, 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 less than 10 minutes, indicating that it is a fairly easy procedure.

6 Conclusion

This paper discussed the problem of timing failure detection and prediction in web service environments and proposes a programming approach to help developers in programming web services with time constraints. The development of a generic and

non-intrusive detection and prediction mechanism is a complex task. Most temporal failure detection/prediction solutions available are coupled to particular applications or to specific sets of services (i.e., these solutions are part of the application itself), which obviously has a huge impact on specific quality attributes.

The approach proposed implements timing failures detection and prediction in a transparent way. The results from several experiments showed that our mechanism can be easily used and introduces a low overhead on the target system. The mechanism is also able to perform fast failure detection and can accurately predict timing failures. This paper sets the basis for a prediction mechanism that targets web services and is perfectly tailored for compound services. Future work will involve studying learning algorithms, and using them to further improve prediction capabilities.

7 References

1. Chappell, D.A., Jewell, T.: Java Web Services: Using Java in Service Oriented Architectures, O'Reilly (2002).
2. Elmagarmid, A.K.: Database Transaction Models for Advanced Applications, Morgan Kaufmann (1992).
3. Chandra, T., Toueg, S. Unreliable Failure Detectors for Reliable Distributed Systems, *Journal of the ACM*, 43(2), 225–267 (1996).
4. Dwork, L., Stockmeyer, L.: Consensus in the Presence of Partial Synchrony, *Journal of the ACM* (1988).
5. Cristian, F., Fetzer, C.: The Timed Asynchronous Distributed System Model, *IEEE Transactions on Parallel and Distributed Systems* (1999).
6. Verissimo, P., Casimiro, A.: The Timely Computing Base Model and Architecture, *Transactions on Computers - Special Issue on Asynch. Real-Time Systems* (2002).
7. Laranjeiro, N., Vieira, M., Madeira, H.: Timing Failures Detection in Web Services, *IEEE Asia-Pacific Services Computing Conference* (2008)
8. Bo, Y., Xiang, L.: A study on software reliability prediction based on support vector machines, *IEEE Intl Conf. on Industrial Eng. and Eng. Management*, pp. 1176-1180 (2007).
9. Kootbally, Z., Madhavan, R., Schlenoff, C.: Prediction in Dynamic Environments via Identification of Critical Time Points, *Military Comm. Conf. (MILCOM 2006)*, pp. 1-7 (2006).
10. Su, S.-F., Lin, C.-B., Hsu, Y.-T.: A high precision global prediction approach based on local prediction approaches, *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 32, pp. 416-425 (2002).
11. Kiczales, G. et al.: Aspect-Oriented Programming. In: *11th European Conference on Object-oriented Programming* (1997).
12. SpringSource, Aspect Oriented Programming with Spring, <http://static.springframework.org>
13. Biggs, N., Lloyd, E.K., Wilson, R.J.: *Graph Theory*, 1736-1936, Clarendon Press, 1986
14. The Eclipse Foundation: The AspectJ Project, <http://www.eclipse.org/aspectj/>.
15. Sun Microsystems, Inc.: JAX-WS: JAX-WS Reference Implementation, <https://jax-ws.dev.java.net/>.
16. Dijkstra, E.W.: A note on two problems in connexion with graphs, *Numerische Mathematik*, vol. 1, pp. 269-271 (1959).
17. Laranjeiro, N., Vieira, M.: wsTFDP: Web Services Timing Failures Detection and Prediction, <http://cisuc.dei.uc.pt/sse/downloads.php> (2008)
18. Apache Software Foundation, Apache Maven Project, <http://maven.apache.org/>.
19. Transaction Processing Performance Council, TPC Benchmark™ App (Application Server) Standard Specification, Version 1.3, http://www.tpc.org/tpc_app/.