Mestrado em Engenharia Informática
Dissertação/Estágio
Relatório Final

PreX – Preventive Exception Handling

João Ricardo Lourenço
joaoml@student.dei.uc.pt

Orientador:
Bruno Miguel Brás Cabral
bcabral@dei.uc.pt

Co-Orientador:
Jorge Bernardino
jorge@isec.pt

Data: 30 de Junho de 2016

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Mestrado em Engenharia Informática
Dissertação
Relatório Final

PreX – Preventive Exception Handling

Julho, 2016

João Ricardo Lourenço
joaoml@student.dei.uc.pt

Orientador:
Bruno Miguel Brás Cabral
bcabral@dei.uc.pt

Co-Orientador:
Jorge Bernardino
jorge@isec.pt

Júri:
Prof. Dr. Carlos Bento
bento@dei.uc.pt

Prof. Dr. Tiago Baptista
baptista@dei.uc.pt

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

The exception handling mechanism has been one of the most used reliability tools in programming languages for over four decades. Nearly all modern languages have some form of "try-catch" model for exception handling and encourage its use. Nevertheless, this model has not seen significant change, even in the face of new challenges, such as concurrent programming and the advent of reactive programming. As it stands, the current model is reactive, rather than proactive — exceptions are raised, caught, and handled. Online Failure Prediction techniques generally work at a very high level, showing potential for prediction of program crashes. However, these techniques have never been at the hands of the programmers as an effective tool to improve software quality. This work proposes an alternative exception handling model — PreX — where exceptions are no longer caught but, rather, predicted and possibly prevented. By applying recent advances in Online Failure Prediction to Exception Handling, PreX aims to fully prevent exceptions, bringing failure prediction techniques to a much more fine-grained level that the programmer can control. Predicting exceptions enables a range of preventive measures that enhance the reliability and robustness of a system, offering new revitalization strategies to developers. In addition to introducing the concept of PreX, this work defines its model and architecture and provides a full evaluation of its prototype implementation, showing that it offers significant advantages to developers and that it can be applied to real-world projects.

Acknowledgements

This work would not have been possible without the help of many people. First and foremost, I would like to thank my supervisors Bruno Cabral and Jorge Bernardino. Professor Bruno introduced me to the world of research three years ago, during my second year at DEI, and he has taught me too many valuable things to list. Professor Bernardino has gradually taught me that every little effort matters and that we should constantly strive to exceed ourselves. My supervisors have guided me through a fascinating – yet no-doubt difficult – path throughout this thesis, and I am truly grateful for their guidance, ideas, feedback support and, most of all, seemingly unshakable belief in my abilities. I would also like to thank professor Marco Vieira for believing in me when he integrated me into the FEED project. This project, and in particular his insight, have helped me grow as researcher.

I don't say it often, but I am extremely grateful to the group of friends I have formed in this department, many of whom probably don't really know how much I admire and look up to them. Among these, I would like to personally thank João Soares, João Simões, João Nuno Oliveira and Bruno Caceiro. Throughout these five years, they have offered me countless moments of fun, insight and reflection. I can always rely on them for feedback regarding my work, life or whatever was on the news thirty days ago. Although I don't think they realize it, they are on the front-line of the people I never want to disappoint. It wouldn't be this fun being crazy if I didn't have them to be crazy with.

I would also like to thank my girlfriend for always making me feel like I'm special. The days pass, but her support, love and availability remain the same: I always have someone to vent with, talk to and to help me forget that endless stack of unfinished work there is in the back of my head. She doesn't really realize how important she really is to my well-being, and I guess that's a good thing. I love you, Mariana.

Finally, the past year has made me realize just how much I owe to my parents. Not everyone can talk about their day uninterruptedly whenever they need to. Not everyone can have the privilege of being helped throughout their life, gradually, with school, with work and with all aspects of life itself. Not everyone can have a safe, sheltered, fun, friendly, motivating and highly rewarding environment. I am far from perfect, but whatever semblances to perfection I might have are surely the outcome of their hard-work in raising me and showing me the right things to do and the right time to do them. May this thesis be the least I can do to thank them, because they deserve the world.

Contents

Chapter 1

Introduction

This chapter provides an overall perspective of the thesis. Section 1.1 introduces the scope and motivation. Section 1.2 details the goals. Section 1.3 notes the contributions of the thesis. Finally, the last section describes the structure of the remainder of the document.

This thesis aims to bring forward a paradigm shift in exception handling models. Instead of acting on an exception, a new exception handling model can trigger an alarm of a possible (i.e. predicted) exception, allowing for attempts at its prevention. This adds robustness and reliability to the software, which can then act on a potential exception and proactively alter its behavior. The increased reliability can come at a fraction of the cost of other alternatives.

1.1 Motivation

The Exception Handling (EH) mechanism [1] has been one of the most used reliability tools in programming languages for more than four decades. Most programming languages, such as Java, Python and C++, provide some form of "try-catch" model for Exception Handling. This model has gone unchanged, even in the face of concurrent software and programming languages for multi-core platforms (e.g. Scala, Erlang and Elixir). Thus, the sequential Exception Handling model remains the leader in popularity [2].

However, the ubiquity of the Exception Handling mechanism for error recovery does not imply its most correct or desirable usage [3]. Developers often use Exception Handling language constructs as a way of hiding problems, performing log activities or informing the user of unexpected behaviour, rather than recovering from it autonomously [4]. This

kind of use of Exception Handling might be considered a symptom of a design flaw in the mechanism – the system only acts when it is too late, thus making the problem unavoidable.

Recent work in the field of Online Failure Prediction (OFP) [5–7] has shown that several techniques can be used with success in predicting failures. OFP mechanisms act in run-time, providing warnings and estimating failure probabilities according to the characteristics of the running system. While these systems have seen some success, they are a high-level approach to failure prediction, without fine-grained control at the source code level [5]. There has been some [6, 8] work on predicting failures of individual applications and components, but these are too generic, predicting only task completion status (success/error) [6] or generic component failures [8] (e.g. memory failures). There is lack of a lower-level OFP mechanism that can notify programs and applications of relevant, specific, and potential failures, such as a database timeout, instead of merely predicting the overall failure of the system or the exit status of applications.

It seems, then, that Online Failure Prediction mechanisms could be applied at a lower-level, together with the Exception Handling mechanism, to provide tools for programmers to act proactively in the face of potential exceptions, leading to an overall increase in software reliability and robustness. For example, by detecting that a database timeout is imminent due to excessive workload, an application can adjust its workload, preventing the failure instead of failing and only afterwards trigering the failure handling strategy, thus reducing overall downtime and increasing robustness, reliability and performance.

To further illustrate the motivation for this new model, consider a system of at least a database and several client applications. Consider also that these client applications are write-heavy, meaning they processes several thousands of write operations per second, sending them to the database. Due to the heavy load, the database may become unresponsive and ultimately trigger a *ConnectionTimeout* exception on one of the applications. That application will then have to attempt to reconnect, and restart where it was previously, if such is really possible. This shows the aforementioned downfall of the conventional Exception Handling mechanism – the system only reacts to exceptions, it does not avoid them. The motivation for this work stems from this issue – client applications would benefit from a prediction (i.e. a warning) that the database may trigger a *ConnectionTimeout* exception. With such a warning, they can, for instance, proactively slow their execution rate and prevent the exception from happening. Ultimately, slowing execution down could prove to be more efficient than triggering the exception and restarting the whole process and, as seen in the results of this work, this is indeed a valuable alternative to current reliability tools.

Note also that the proposed model offers new strategies that are not traditionally available to application developers or system administrators. For example, consider a scenario where a web-service exposes several services and is faced with a Denial of Service (DoS) on one of these services. Ordinarily, solutions involve a blocking the source IPs or terminating the web-service altogether. This is a direct consequence of the layer at which intrusion detection systems work, which is outside of the application itself. With PreX, the attack can be detected at the application level (indeed, it can be detected within a specific method), allowing the developer to shut down only the relevant services and letting the remainder of the web-service continue working as normally. This fine granularity is one of the defining distinctions between PreX and other work in the area of failure prediction.

1.2 Goals

The purpose of this thesis is to introduce a novel approach to Exception Handling by providing the means for developers to act on an exception before it happens, thus broadening the range of their revitalization strategies. The approach reshapes the concept of "try-catch" blocks, so that programmers can be alerted of potential exceptions within a given time frame and take some action, much like in the conventional Exception Handling mechanism. Thus, a potential exception is detected and the program flow is transferred to an exception prevention block. In this block, the programmer can specify what to do when faced with a probable exception.

This new Exception Handling approach is called **Preventive Exception Handling (PreX)**, and applies the methods and techniques of the Online Failure Prediction field to exception handling, thus empowering programmers to act proactively. PreX introduces a new model for Exception Handling, with the goal of being easy to use, practical, and a successful integration of the fields of Exception Handling and Online Failure Prediction, as well as the first to act on potential exceptions before they have happened – errors are avoided, rather than handled.

The final goals of the thesis are: (i) the design of PreX; (ii) a prototype implementation of PreX; (iii) a validation of PreX; and (iv) publication of the full source-code for the implementation under an open-source license.

1.3 Contributions

The main contributions of this work are:

- **The proposal and design of a new preventive exception handling model** – **PreX** –, applying Online Failure Prediction techniques at the source-code/language level for fine-grained control over exception prediction.
- The **implementation of the aforementioned Exception Handling model**, in the form of **tools for efficient resource monitoring** (probes), a **Java 8 library** that can be used in any project to incorporate this model, a **system for real-time prediction of exceptions** and an **administrator application** to control this system with both CLI and GUI interfaces.
- The **evaluation of the proposed model**, by using the aforementioned implementation on **real-world scenarios**, namely the open-source Shopizer e-commerce application.
- The **publication of the full source-code for the implementation** and modified software files under an **open-source license**. The source code for the PreX system (coordinator, library and administration application) is available at `https://github.com/Jorl17/prex` and the source-code for a cross-platform probe that is compatible with PreX is available at `https://github.com/Jorl17/prex-probe`.

1.4 Structure

The remaining document is structured as follows. Chapter 2 introduces the state of the art in exception models in programming languages and failure prediction, as well as related work. Chapter 3 specifies the proposed model and its architecture, as well as a validation of its usefulness with simulation tools. Chapter 4 details the implementation of the proposed model in the Java programming language. Chapter 5 presents the experiments used to validate the model and its implementation. Chapter 6 presents the work plan used throughout the thesis, a risk analysis and a publication plan. Finally, Chapter 7 presents the conclusions and future work possibilities.

Chapter 2

State of the Art

This chapter presents the state of the art in two distinct areas of research – exception handling models and online failure prediction. In addition, it notes related work in the field of self-healing systems and business process exception handling. A brief summary is presented at the end of the chapter.

2.1 Definitions

The following sections provide definitions of the terms used throughout this work.

There have been several attempts to get a precise definition of faults, errors and failures (e.g. [9], [10] and [11]). Since this chapter is heavily based on the survey by Salfner et al. [5], the definitions of failure, error and fault here presented are the sames found in work by Avižienis et al. [11] and used in the work of Salfner et al. The definition of symptom is the one used in the work of Salfner et al.

2.1.1 Failure

A service failure, often abbreviated as **failure**, is the event that happens when a service deviates from correct operation. This may happen because it does not comply with its functional specification or because the original specification does not adequately describe system function. Salfner et al. note that a failure refers to "misbehavior that can be observed by the user, which can be a human or another computer system". In that sense, although "things might go wrong" in a system, it does not constitute a failure insofar as there is no deviation from correct service.

2.1.2 Error

The aforementioned situation where "things might go wrong" is formalized as the situation where "the system's state deviates from the correct state". This is called an **error**. An error corresponds to the part of the total system state that may lead to a subsequent service failure. Many errors never reach the system's external state, hence do not cause a failure.

Avižienis et al. [11] further distinguish between **undetected errors** and **detected errors**: An error can remain unidentified until a detector identifies the incorrect state.

2.1.3 Fault

A **fault** is the "hypothesized cause of an error". In other words, it is its root cause. Often, faults remain dormant until they are activated, leading to incorrect system state (an error) and, ultimately, might lead to a failure.

2.1.4 Symptom

Errors might cause failures, but they might also cause "out-of-norm" behavior as a side-effect. For example, the system might operate with expected results but do so with increased CPU usage or memory consumption. This behaviour is called a **symptom** [5].

2.1.5 Relationship between concepts

Figure 2.1 presents an overview of the concepts introduced in this section. As an example, consider a system with a memory leak. A fault, the underlying cause of the error, might be a missing *delete* or *free* statement. Once the offending piece of code (responsible for freeing) is executed, an incorrect state is entered, thus, an error becomes present. Through time, memory consumption will increase and this will affect overall system metrics and performance (these side-effects are the symptoms). Eventually, if no preventive action is taken, a failure happens (i.e. the system crashes).

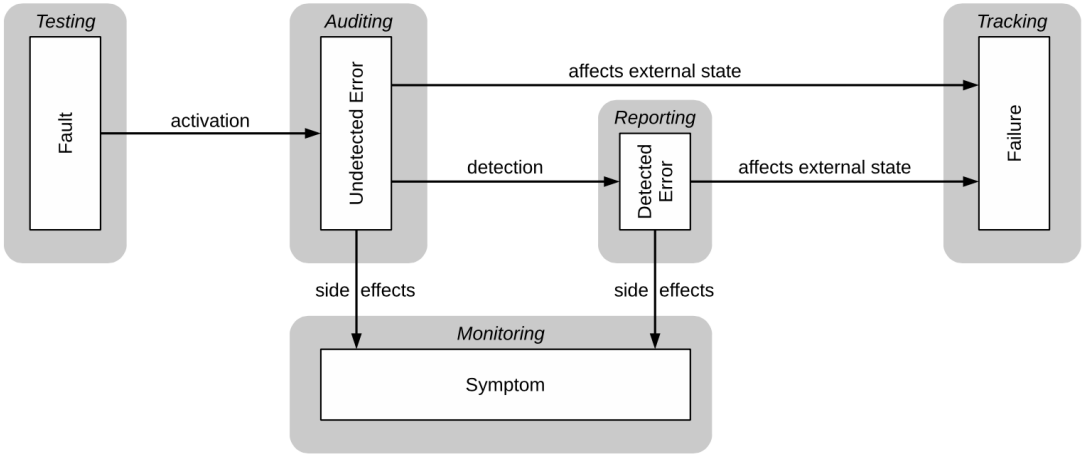

FIGURE 2.1: Relationship between faults, errors, symptons and failures. The encapsulating boxes show the technique by which the corresponding phemonenon can be made visible (Salfner et al. [5]).

2.2 Online Failure Prediction

Recent trends in industry and academia have seen a shift from traditional fault handling approaches to new efforts on autonomic computing, trustworthy computing, recovery-oriented computing and other techniques for proactively handling failures. This shift encompasses techniques that fit onto the term *Proactive Fault management* [5]. In Proactive Fault Management, there are essentially four steps [5]:

1. Online Failure Prediction is performed in order to identify problems. This step can estimate failure probabilities, outputting a probability, a binary (fail/no fail) scenario or some other measure of failures happening *in the future*.

2. Once failures have been predicted within a threshold, they have to be further pinpointed (e.g. which component is likely to cause failure?), in an act of *diagnosis*.

3. Based on the previous steps, some actions (i.e. countermeasures) must be taken using several decision making systems. These systems take into account the confidence in the diagnosis and other relevant system information in a process called *action scheduling*.

4. Lastly, the selected actions are carried out (i.e. *executed*) in hopes of preventing the predicted fault. Examples of these actions are reconfiguration of distributed systems or data synchronization among data centers.

Thus, Online Failure Prediction is only one of four distinct and critical steps in Proactive Fault Management systems. It is, nevertheless, a large research field on its own.

2.2.1 Online Prediction

Online prediction is visualized in Figure 2.2. A failure is to be predicted with some given **lead-time** Δt_l, at current time t, based on the current system state measured within a data window of length Δt_d (which we call **data validity time**). This prediction (e.g. fail/no fail, failure probability, etc) is valid during a window Δt_p, called the **prediction period**.

A failure might not always be predicted in time. For instance, if we predict an "out-of-memory" error within 2 seconds, but need 5 seconds to fully slow down our process and prevent the error, then the prediction *lead-time* will have been useless. Thus, there is a **minimum warning time**, Δt_w, needed for a system to react proactively to the failure prediction.

The prediction period, Δt_p, is critical in online prediction. If a value is too low, the prediction is prone to fail many times, so higher values always increase the amount of correct predictions. However, very large windows make the prediction useless – it is not useful to know that a system is going to fail somewhere between today and five hundred years.

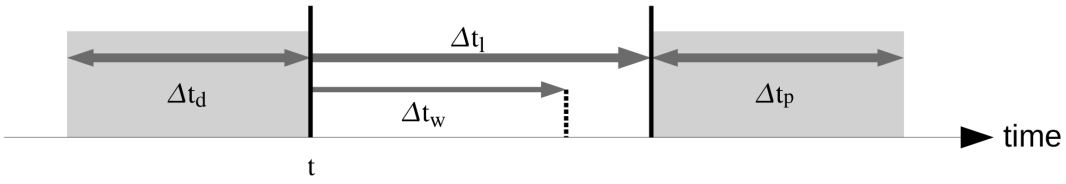

FIGURE 2.2: Time relations in online failure prediction (Salfner et al. [5]).

2.2.2 Evaluation Metrics

In order evaluate the quality of fault prediction algorithms, a set of evaluation metrics must be defined. There are several common metrics used to assess these algorithms. Most of the metrics are defined based on the Contingency Table as displayed in Table 2.1.

TABLE 2.1: Contingency table

	True Failure	**True Non-Failure**	**Sum**
Predicted Failure (failure warning)	true positive (TP) (correct warning)	false positive (FP) (false warning)	positives (POS)
Predicted Non-failure (no failure warning)	false negative (FN) (missing warning)	true negative (TN) (correct lack of warning)	negatives (NEG)
Sum	failures (F)	non-failures (NF)	total (N)

Based on the contingency table (also known as the confusion matrix), there are common ways to measure the performance of a fault prediction algorithm [5]. These are shown in Table 2.2

TABLE 2.2: Common formulas to measure the performance of a fault prediction algorithm

Name	Formula
Precision	$\frac{TP}{TP+FP} = \frac{TP}{POS}$
True Positive Rate Recall	$\frac{TP}{TP+FN} = \frac{TP}{F}$
False positive rate	$\frac{FP}{FP+TN} = \frac{FP}{NF}$
True negative rate Specificity	$\frac{TN}{TN+FP} = \frac{TN}{NF}$
False negative rate	$\frac{FN}{TP+FN} = \frac{FN}{F}$
Negative predictive value	$\frac{TN}{TN+FN} = \frac{TN}{NEG}$
False positive error rate	$\frac{FP}{FP+TP} = \frac{FP}{POS}$
Accuracy	$\frac{TP+TN}{N}$
Odds ratio	$\frac{TP \times TN}{FP \times FN}$

The True Positive Rate (TPR), False Positive Rate (FPR), False Negative Rate (FNR) and True Negative Rate (TNR) can all be expressed as a percentage or a value in the range $[0; 1]$. If $TPR = 1$, then all failures are accurately predicted. However, this metric alone is not enough to evaluate the performance of an algorithm. Indeed, one can trivially achieve $TPR = 1$ by *always* predicting a failure. The downside of this is that the True Negative Rate will be 0. There is a compromise between TPR and TNR because there are two distinct kinds of errors: (i) failing to correctly predict a failure (a false negative) and (ii) predicting a failure when there is none (a false positive). This leads to another popular metric that tries to balance these two different kinds of errors, called the F-measure (F_1):

$$F_1 = 2\frac{precision \times recall}{precision + recall}$$

An F_1 of 1 (or 100%) is achieved by a perfect prediction system which never fails.

2.2.3 Prediction Methods

Salfner et al. [5] identified four distinct major branches of methods in Online Failure Prediction:

- Failure tracking
- Symptom Monitoring
- Detected Error Reporting
- Undetected Error Auditing

In the following sections, we introduce each of these branches and note relevant research work in them. For a more in depth state of the art, interested readers should look at the aforementioned survey [5].

2.2.3.1 Failure Tracking

Methods belonging to the class of Failure Tracking draw conclusions regarding future failures from past failures. These methods don't explore run-time data beyond simple tracking of time.

There are two main classes of methods within Failure Tracking. These are ***Probability Distribution Estimation*** **methods**, which have their roots in *offline* reliability prediction, and ***Co-Occurrence*** **methods**. The former methods try to estimate the probability distribution of *time-to-fail*, whereas the latter methods explore spatial proximity (i.e. machines tend to fail in proximity clusters; thus, the probability of failure should increase with co-occurent failures).

Csenki [12] uses a Bayesian predictive approach to improve the prediction of the next time to failure in an offline software reliability prediction context. Pfefferman and Cernuschi-Frias [13] also propose a Probability Distribution Estimation method by using a non-parametric method to model the failure process (time between failures) as a Bernoulli-experiment where a failure of type k occurs at time n with probability $p_k(n)$. With

this information, the authors apply different "window sizes" based on the probability for each kind of failure, applying their new model to software reliability data. Regarding Co-Occurence methods, these are mostly used for root cause analysis rather than failure prediction. Liang et al. [8] explored temporal and spatial correlation to successfully predict hardware component failures in IBM's BlueGene/L. Fu and Xu [14] use a similar approach, coupled with a neural network, to predict the number of hardware failures with 81.6% accuracy and software failures with 72.9% accuracy for the LANL HPC system. These results are above average for online failure prediction, where, as shown in this section, most results tend towards 50% accuracy.

2.2.3.2 Symptom Monitoring

Symptom Monitoring methods periodically analyze samples of system state variables, such as the amount of free memory or page swaps, to estimate imminent failures. A key concept of these methods is the idea of *service degradation* – the system may start to exhibit some form of degradation before the actual failure happens (e.g. before a database time-out, successive requests might take a longer time to process). This degradation can be observed in system *side-effects*, such as longer response times, high CPU load or high memory consumption (e.g. in the case of memory leaks). These side-effects are called the *symptoms* of the failure.

Subclasses of Symptom Monitoring methods are:

- ***Function Approximation* methods** use the system samples to estimate and approximate a target function. This function can be one of (i) the probability of failure occurrence (only available in the training dataset as a boolean function) or (ii) some computing resource such as the amount of page swaps (which is available in run-time within the *data validity time*).
- ***Classifier* methods** directly use the values of system variables to ascertain failure-prone or failure-free state. A *decision boundary* in the system variable space is computed from previous data in a training phase, separating data points which indicate a "failure" and data points which indicate "normal operation". For example, if there is only one variable, such as free memory, a decision boundary can be expressed in the form $freemem \leq 100$ Mb. If free memory falls below 100 Mb, the predictor outputs a failure-prone state. Classifiers generalize this idea to higher dimensionality cases, and then compare the current data values to the precomputed decision boundary.

- ***System Models* methods** are similar to Classifier methods, but differ in the sense that they only rely on failure-free data. Instead of computing a decision boundary, they compute a normal operation system model, comparing its estimation outcome with the current system functioning and using this as the basis of the failure prediction.
- ***Time series analysis* methods** treat monitored system variables as a time series and use analysis of samples to predict future values which are then used for the prediction.

Regarding function approximation, Vaidyanathan and Trivedi [15] present a stochastic approach in the form of a semi-Markov reward model based on workload and resource usage data to predict resource exhaustion due to software aging. Their goal is to control software rejuvenation processes and validate the phenomenon of software aging. The proposed model gives better trend estimates of swap-space and free memory than simply using a time-based approach. However, since this model does not model interactions between resources, it does not necessarily accurately predict a system failure. In Li et al. [16], another stochastic approach is given, where an auto-regressive model with auxiliary input is used in a similar way, predicting resource exhaustion times. Their method was shown to be computationally less expensive and have better results than Castelli et al.'s method [17] on their dataset. Andrzejak and Silva [18] employ a regression based approach to model the performance of an Apache Axis SOAP server. Their goal was to optimize software rejuvenation times, something which they were able to achieve. Regarding Machine Learning approaches, Neville [19] described how standard neural networks can be used for failure prediction in large scale engineering plants. Hoffmann [20] used an approach based on Universal Basis functions (UBF) to predict failures of a telecommunication system. In a followup work [21], the authors predict resource consumption of the Apache webserver using different modelling techniques. Their UBF approach yielded the best results for free physical memory prediction, although Support Vector Machines were a better choice for server response times.

Classifier methods for this category of online failure prediction include the work of Hamerly and Elkan [22]. The authors perform hard disk drive failure prediction with two different bayesian methods. The first uses a mixture of naive Bayes submodules and the second is a naive Bayes classifier trained with Expectation-Maximization. This second method computes conditional probabilities for SMART (Self-Monitoring, Analysis, and Reporting Technology) values belonging to the failure/non-failure classification. Using a decision threshold of 0.005, their approach achieves a true positive rate of 0.33 with false warnings having a probability of 0.0036. Of all failures, they predict 56% of them

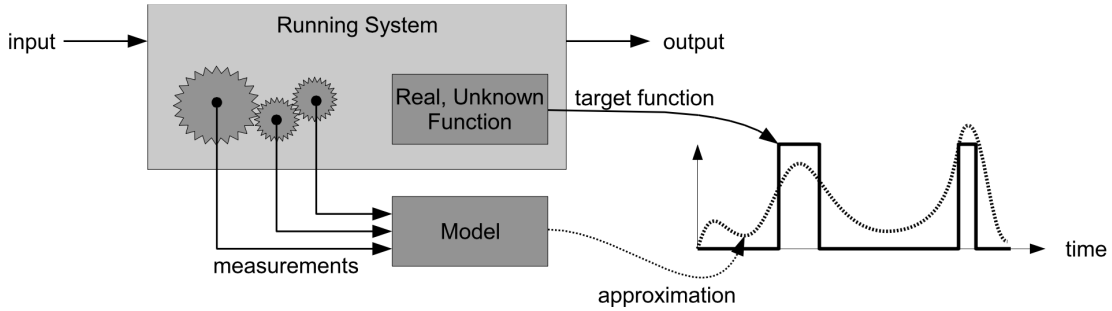

FIGURE 2.3: Function approximation methods approximate target functions (Salfner et al. [5]).

with a false positive rate of 0.0082 and a class threshold t of 0.001, outperforming industry standard methods. Pizza et al. [23] present a method for classification of faults in transient or permanent faults whenever erroneous behavior is observed. Salfner et al. note that this method can be used for failure prediction. Turnbull and Alldrin [24] use Radial Basis Function networks to classify data windows as failure/non-failure. These data windows contain monitoring values of hardware sensors (e.g. temperature and voltage). They achieve a 0.87 true positive rate and 0.10 false positive rate on a balanced dataset (i.e. with the same number of positive and negative classifications). Murray et al. [25] apply SVMs, among other algorithms that do not fit into this category, to predict hard-drive failures. They predict roughly 45% of failures (true positive rate of 0.45) with about 12% false alarms (false positive rate 0.12), which the authors consider might be higher than desirable. Their approach is interesting in the sense that they develop a window-based method to group data for classification. Bodík et al. [26] analyse hit frequencies of web-pages using a naive Bayes classifier. They provide a visual tool that can predict failures with an average warning time of 37 hours, although their dataset is very small and labels for failure/non-failure were not available (the authors extracted these using unsupervised learning techniques). Although not included in the survey by Salfner et al., Irrera et al. [27] presented a sliding window approach to incorporate the time dimension in a classification failure prediction problem, showing that this dimension improves results, but that it makes the problem harder as larger windows are used.

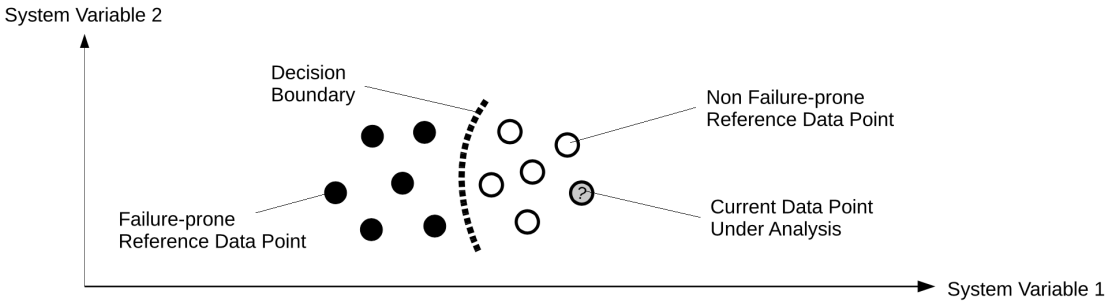

FIGURE 2.4: Classifier methods use a decision-boundary to distinguish between failure and failure-free states (Salfner et al. [5]).

Regarding system models, Elbaum et al. [28] propose methods where they inspect function calls, changes in configuration and module loading of the *pine* email client. Their most successful method used sequence-based checking: a failure was predicted whenever two successive events that did not belong to the stored training event data happened. Hughes et al. [29] use a statistical test for hard disk failure prediction. They compare values from fault-free hard-disks at run-time and predict a failure whenever a certain threshold is exceeded. They are able to predict failures with 40%-60% accuracy and 0.002 false positive rate (thus, having lower performance than Hamerly and Elkan [22]'s method). A similar statistical approach is used by Bodík et al. [26]. Their work, already mentioned in another categorization, provides an additional method where website hit frequencies of the 40 most frequently used pages are compared to "historically normal" hit frequencies using a statistical test. Significant differences are considered anomalous and failure-prone. Their results can predict failures with the aforementioned average warning time of 37 hours, but have the same aforementioned limitations. Ward et al. [30] use a statistical test to compare the mean and variance of the number of TCP connections of two web proxy servers to identify performance failures. They achieve 80%-90% accuracy with "only two false failures".

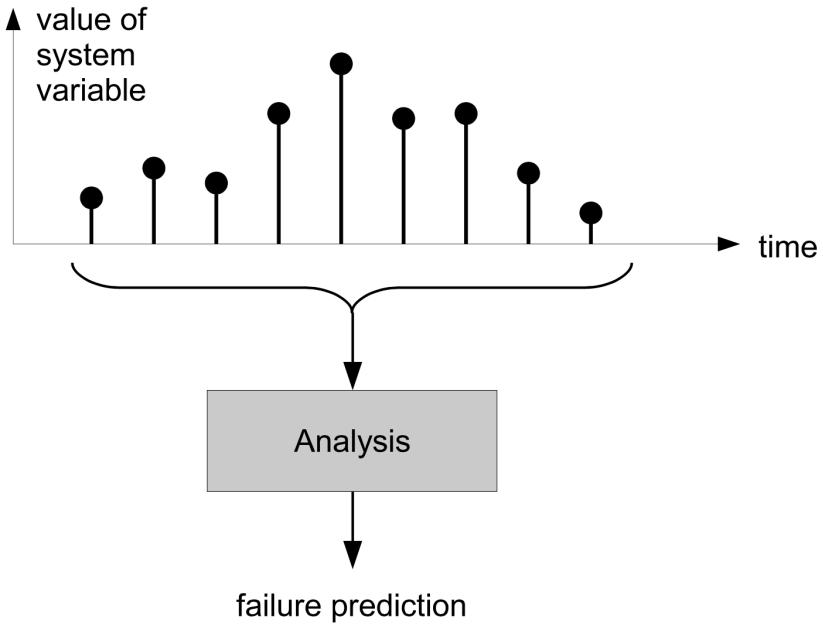

FIGURE 2.5: Time series analysis methods operate on a sequence of variables through time (Salfner et al. [5]).

Lastly, concerning time series analysis models, Garg et al. [31] present a three-step approach to predict resource exhaustion. Their approach smooths the time-series and detects trends using a seasonal Kendall test (the method cannot be applied otherwise), finally applying a non-parametric procedure for prediction. Their experiments were performed on variables such as free memory, size of file table, process table size and used

swap space of UNIX machines. Castelli et al. [17] note that IBM implemented a curve-fitting algorithm for the xSeries Software Rejuvenation Agent, using extrapolation for prediction. Cheng et al. [32] presented a two-step approach for failure prediction within a high availability cluster system. Their approach first computes a health index using fuzzy logic. If a node is considered "sick", then they use a linear function to estimate mean time to resource exhaustion. They showed that they could improve availability due to accurate prediction and recovery mechanisms (backup nodes and system administrator notifications). Shereshevsky et al. [33] observed that memory-related system parameters show multifractal characteristics in the case of software aging, using this knowledge to predict system crashes.

2.2.3.3 Detected Error Reporting

In *Detected Error Reporting* methods, past error logs are used to predict future failures. This approach differs significantly from Symptom based approaches (which use system variables) in two ways: (i) events, unlike system variables, are not continuously spaced/-monitored, and thus require an *event-oriented* approach; (ii) system variables are usually real-valued, whereas event logs usually contain discrete information such as a timestamp or an ID. This branch of methods, then, carries a data window (coinciding with the *data validity time*) whose past events are taken into account when predicting the future.

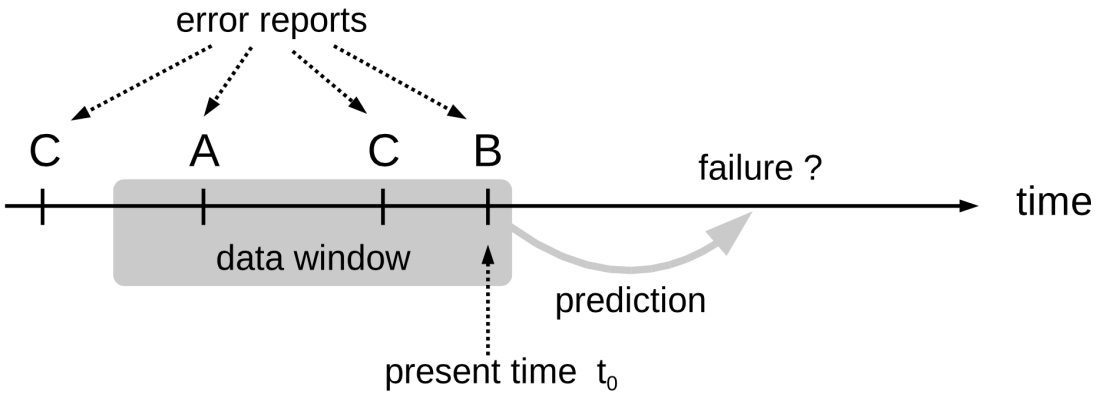

FIGURE 2.6: Detected error reporting methods use previous error reports for predicting failures, as opposed to other variables such as system statistics (Salfner et al. [5]).

Like Symptom based approaches, Detected Error Reporting shares many topics with the area of machine learning, and Salfner et al. distinguish the following subclasses of methods:

- ***Rule-based* methods** use rule-based systems for the basis of their predictions. A series of conditions (called *rules*) are iterated until the first that matches is found

(indicating failure). If none are found, then failure-prone states are discarded. Thus, systems of this form usually have rules in the form of

$$IF\, condition_1\, THEN\, failure\_prone\_status$$

$$IF\, condition_2\, THEN\, failure\_prone\_status$$

...

- ***Co-Occurence* methods** are similar to the Co-Occurence methods presented in Section 2.2.3.2 with the only difference that they are based on detected errors (events) rather than previous failure data.
- ***Pattern Recognition* methods** search for patterns in the error reports. These methods usually use a ranking value to express similarity between patterns that are known to lead to failures and patterns that are known to be of normal operation.
- ***Statistical Test* methods** rely on statistical testing approaches to failure prediction. Examples include checking the histogram of current number of error reports and comparing it to one during normal operation using statistical tests.
- ***Classifiers* methods**, like Co-Occurence methods, are similar to the Classifiers methods presented in Section 2.2.3.2 with the only difference that they are based on detected errors (events) rather than previous failure data.

The first rule-based approach seems to be that of Hätönen et al. The authors built a system with simple rules of the form "if errors A and B occur within X seconds, then error C occurs within Y seconds with probability p". Their algorithm returned too many rules and needed to be post-processed/filtered by a human operator. Weiss [34, 35] presents a failure prediction technique called "timeweaver" based on a genetic training algorithm. The algorithm builds rules using a genetic programming approach, starting from an initial set of rules and building new ones with crossing and mutations. These are then evaluated with a fitness function that considers prediction quality as well as diversity of the rule set. They apply this algorithm to alarms of the 4ESS Switches that route the majority of the AT&T network traffic. The author notes that the performance of the algorithm is heavily influenced by the lead-time and that good results can be achieved if the lead-time is low (1-10 seconds). Vilalta et al. [36] propose the eventset method, using a data mining approach (a rule-based model) based on the type of reported error (the time dimension is discarded). Under specific conditions, they have a false negative rate of only 0.16, although this value can be as extreme as 0.83 under other system conditions (the false positive rate is always lower than 0.1).

Regarding Co-Occurence, Lin and Siewiorek [37] present a set of heuristic rules on the time of occurrence of consecutive error events to identify permanent failures. Their technique is mostly suited to hardware failures, and achieves a true positive rate of 0.9375 and precision of 0.75. However, the method has little value if failures do not occur frequently. Several authors [38–40] use grouping methods to make the observation that temporal co-occurence should be used to predict upcoming failures in short timespans.

Salfner et al., who have proposed the classification used in this thesis, also present the only identified methods of Detected Error Reporting with Pattern Recognition. Their central idea is that of merging the time-based characteristics of Lin and Siewiorek's [37] method with the type-of-error characteristic of Vilalta et al.'s [36] method. In this sense, in [41], the authors present Similar Events Prediction (SEP), a semi-Markov chain model. Their method achieves a precision of 0.800, recall of 0.923 and F-Measure of 0.8571 on data from an industrial telecommunication system. In Salfner and Malek [42], the authors propose a hidden semi-Markov model to "add an additional level of flexibility". This method achieves precision of 0.852, recall of 0.657 and F-Measure of 0.7419. Both of these methods seem to perform better than other failure prediction methods, whose precision and recall tend towards 50-60%.

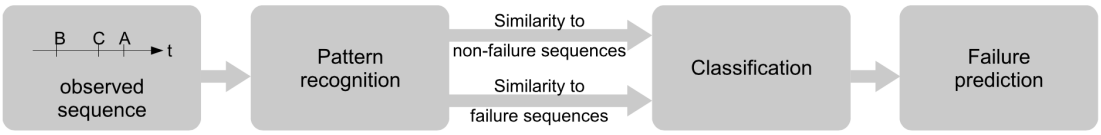

FIGURE 2.7: Pattern Recognition methods extract features from data and compare these to failure and non-failure sequences (Salfner et al. [5]).

There is only one documented instance of classification approaches to Detected Error Reporting. Domeniconi et al. [43] present an extension of Support Vector Machines called SVD-SVM (Singular-Value-Decomposition and Support-Vector-Machine) and apply this extension to predict critical or fatal system errors. The approach was used with data of a production computer network with an "error rate" of 7.2% to 8.6% for online prediction.

2.2.3.4 Undetected Error Auditing

Detected Error Reporting, which we have already discussed, searches for errors in data that is actively being used. *Undetected Error Auditing*, on the other hand, looks for errors and error states in data not currently used. For example, a consistency check on the filesystem might help predict failures in the future – if some files are found to be inconsistent, we can predict that accesses to those files will trigger a failure.

Salfner et al. did not find any examples of such systems in their literature review, and this still seems to be the case.

2.2.4 Current Challenges

Online Failure Prediction faces several different challenges in order to be practical. One such challenge regards variable selection and data gathering. Indeed, in [44], the authors note that it is difficult to obtain the necessary data for failure prediction, since failures are rare events. Hoffman et al. [45] also show that the set of variables that help prediction might be quite small. In [46], the authors propose an approach for selecting the most adequate variables for failure prediction. The authors use this approach in followup works [47, 48].

Failure Prediction has been showed to be highly sensitive to the training data. As such, datasets should be chosen carefully, taking special precaution with data resulting from virtualization or fault injection methods. In this sense, Irrera and Vieira [48] present a framework to assess the representativeness of fault injection-generated failure data. In [47], the authors show that data generated within virtualized environments can be used for failure prediction, but that virtualization technology and different workloads can influence the correlation of these variables with failures.

Another challenge in failure prediction is related to naturally evolving software systems. In dynamic environments and real-world scenarios, the system setup is prone to changes (e.g. configuration, hardware and workload changes). In [49], the authors study the performance of a failure predictor when the supervised system (a webserver) is subjected to successive updates. Their results show that the performance of failure predictors is indeed greatly affected by the updates and that re-training improves performance. Berardini et al. [50] propose CASSANDRA, an online failure prediction technique combining compile-time and run-time information to determine the current application state and the possibility of failure. Their method is designed specifically for evolving systems where state changes dynamically, but requires a high degree of compile-time information (e.g. “Which sequence of states leads to a failure?”).

2.3 Exception Handling Models

Exception Handling separates the operation domain (the execution domain of a particular segment of code) into two distinct domains: the operation’s *standard domain*, and the operation’s *exceptional domain*. Normal program flow, absent of errors, is contained

in the *standard domain*. If an operation is invoked within its exceptional domain, it leads to an exception being *raised*, followed by the invocation of an *exception handler*, that, in theory would deal with the abnormal condition (e.g. by logging it, correcting it, or using a different approach altogether). This invocation is done by a *signaller* (i.e. *callee*). An *Exception Handling Model* defines the interaction between the signaller and its handler. An *Exception Handling Mechanism* defines the language constructs within a programming language to express a given Exception Handling Model. [51]. In this section, we give an overview of the classical and current Exception Handling models and their limitations.

2.3.1 Classical Exception Handling Models

Yemini et al. [52] identified four distinct Exception Handling Models, summarized in [53]:

- Resumption model – When an exception is raised, the control flow is transferred from the raise point to the handler and, after the exception has been handled, it is transferred back to the raise point. This model effectively binds the caller and the callee together and is prone to recursive resumption, thus being difficult to implement [53].
- Termination model – An exception is raised within a protected block, with the control flow transferred to the handler, terminating any intervening blocks. The control flow then resumes as if the protected block terminated without any errors. This is the most widespread Exception Handling Model in use [53].
- Retrying model – The signaller is invoked after some operation has been made. This model is more appropriate to transient faults, where retrying the invocation might lead to no exceptions. The main disadvantages of this model are its inherent implications for non-idempotent operations, counters, etc – the programmer must be wary of how the code executes. Since it has been shown [53] that this model can be mimicked by using a loop and the termination model, it is often not explicitly supported.
- Nonlocal transfer – The program flow can be transferred to any other location in the program. This model has the obvious drawback of being hard to maintain and much more error-prone. [53]

All of these models share a common characteristic – an exception is raised, and only then can some action be taken. There is no proactive approach in this form of exception

handling. This, unfortunately, means that many incorrect uses of Exception Handling Mechanisms arise, with programmers focusing on hiding errors, rather than attempting to fix them [4].

2.3.2 Advances in Exception Handling Topics and Models

Exception Handling has been around for more than forty years. Several authors and programming languages have proposed alternative models which extend the classical models and adapt them to new scenarios. In this section, we focus on some of these new models and ideas.

2.3.2.1 Side-Effects

In most programming languages, as mentioned, the termination model is implemented with an Exception Handling mechanism. Typical and familiar syntax provided by modern programming languages, such as Java, Python or C# follows a *try-catch-finally* syntax:

```
1 try {
2     ... Some code that can throw exceptions ...
3 } catch ( ... ) {
4     ... Exception Handling code ....
5 } finally {
6     ... Code to be executed regardless of exception invocation ...
7 }
```

Listing 2.1: Typical syntax for current exception handling models

This syntax provides ways for programmers to employ the termination model and append to it some additional cleanup procedures that should always be executed in a *finally* block. For example, if a file is opened before the *try* block, it does not matter if an *EndOfFileException* is raised and caught within that *try* block, since the programmer will always want to close the file – thus, this should be done in the *finally* block.

The *try-catch-finally* demonstrates how the termination model does not deal with *leaked state* (i. e. *side-effects* of the called code). Code that has been executed may have undesired side-effects that need to be undone so that the application maintains consistency. Indeed, in accordance to Lanvin et al. [54], there are three main goals to exception handling mechanisms:

1. Detection of an error

2. Localization of the error source

3. Error recovering

Lanvin et al. [54] argue that most programming languages achieve the two initial goals but fail to ensure that error recovering ensures a return to a consistent state.

There have been different proposals for extending the termination model and augmenting it with richer semantics to reduce or eliminate the problem of side-effects. In the same paper, Lanvin et al. [54] propose the idea of *reconstructors*: counter-actions for each operation that can "undo" that operation and, thus, bring the program back to a previous state. This approach, however, further adds complexity, as it forces programmers to explicitly program the reconstructors. Note how the *finally* block, seen in other programming languages, tries to accomplish part of this task, but may lack all the necessary information to perform the required cleanup operations (e.g. it may not know exactly where the exception happened).

Apple's Swift programming language [55] introduced error handling techniques that highly resemble exception handling mechanisms. Although implemented differently, they offer very similar syntactic and semantic facilities, the only main difference being the introduction of a *defer* block. Like *finally* in other languages, the *defer* block marks instructions that should be run independently of an exception[1] being caught or not. However, *defer* adds significantly more flexibility than *finally*, since: (i) there can be multiple defer blocks anywhere through the function (they are not tied to a *try-catch* block); (ii) defer blocks are always processed when the function is exited, even if the function itself throws an exception[2] and (iii) the multiple defer blocks are invoked in reverse order of their declaration. This addition, thus, is conceptually similar to Lanvin et al.'s work, and allows errors to be handled more gracefully and state consistency to be achieved more easily due to the more fine-grained *defer* blocks.

2.3.2.2 Concurrent Exception Handling

Concurrent programming is now a core part of most software projects. Yet, the classical Exception Handling models were not developed to take concurrent programming into

[1] Swift 2 does not contain the concept of exceptions explicitly. Instead, it is functionally backwards-compatible with the previous iteration of the language, "piggybacking" error codes in additional function parameters. Nevertheless, the end result is code that shares high resemblance with exception handling, and, in this document, we treat these error messages as exceptions for simplicity.

[2] Note that this makes more sense if we realize that Swift internally implements exceptions mostly as in/out parameters and using conventional return statements.

account [2]. This issue becomes increasingly complex when dealing with implicitly parallel languages (e.g. [2]) or *futures* [56]. In these particular situations, different problems clearly emerge:

- Catching an exception does not necessarily inform us about which potentially asynchronous operations in the sequential *try-catch* block have been executed, thus making it impossible to reliably know the current program state or how to revert to a previous, consistent state.
- If an exception is raised in an asynchronous method call, in which thread should it be handled?
- If an exception is raised in an asynchronous method call, when should it be caught? For example, if Futures are used, should the exception be caught at the point of method invocation, or at some other point in time, such as when requesting the Futures result itself?

There have been attempts at simplifying the process of Concurrent Exception Handling. Zhang et al. [56] propose an *as-if-serial* exception handling mechanism for their particular flavor of Futures for Java (called DBLFutures). Their mechanism provides *as-if-serial* semantics for futures, meaning that "the semantics of the parallel future version is the same as that of the sequential version" (exceptions are delivered to the same point as they are delivered when the program is executed sequentially). The mechanism does not provide true *as-if-serial* semantics, but only does it in terms of the control flow of exception delivery. Thus, the global side effects of parallel execution are not "undone" in the face of an exception. For this, the authors also implement a stricter version of their DBLFutures, named Safe DBLFutures (SDBLFutures), which uses a technique similar to software transactions within the JVM. It should be noted that this mechanism is proposed by the authors as a "cheap solution" to concurrency, where a sequential version has already been written and a concurrent re-write would be too costly.

Gesbert et al. [57] use a barrier system to ensure that parallel executions often hit checkpoints. As noted in [2], this would allow programmers to know the previous safe state to which they should recover, but introduces the problem of a delicate trade-off between the frequency of checkpoints (more frequent checkpoints make Exception Handling easier) and the amount of parallelism. Other solutions [58, 59] rely on Software Transactional Memory to guarantee that operations within a try block either all happen or not, though these can be costly and cannot reliably deal with all side-effects (e.g. undoing writes to files).

2.3.2.3 Actor Model and Functional Reactive Programming

The actor model [60] has also emerged as an alternative model of computation that promises higher scalability [61]. In the actor model, actors communicate through asynchronous messages and without any form of memory/state sharing. Exception Handling in actor-based models shares a high degree of similarity with traditional Exception Handling. For example, in the Akka toolkit [62], when an exception is thrown, the corresponding actor is suspended, as well as all of its children actors, and the failure is signalled to a supervisor. This supervisor can employ four different strategies: (i) resuming the actor, ignoring the exception; (ii) restarting the actor; (iii) terminating the actor; and (iv) escalating the failure to its own supervisor.

Similarly to the actor model, in current Functional Reactive Programming (FRP) languages (where the core concept is that of a stream or flow of data that changes with time and is transformed by language specific operators and functions [63]), Exception Handling also resorts to similar methods, although there are some alternative models. For example, the Flapjax language has different data flows: one for errors and one for data itself, with the additional possibility of using a global error flow [64]. In some languages and frameworks, such as Rx, if an exception handler is not provided for the thrown exception, then it is simply ignored. Thus, FRP can opt by mimicking the typical *try-catch-finally* blocks, although with a different syntax.

2.4 Self-Healing systems and predictive exceptions model

The previous sections have shown how Online Failure Prediction can be successfully used to predict failures, in spite of challenges regarding, for instance, data sampling and feature selection. However, little work has been done for predictions at a more fine-grained level. Predictions are usually made at the system level, at most predicting a generic "crash" of some component. Thus, while promising, these techniques have no practical use for developers who wish to provide specific counter-measures when faced with the possibility of an exception.

There has been some related work done in the field of self-healing systems. Magalhães and Silva [65] propose a general self-healing proactive framework for web-based applications. Their work introduces a general framework to create self-healing transactional web-based systems. The framework, although operating at a lower level than traditional Online Failure Prediction methods, does not support run-time notifications at code-level nor operate at the fine-grained level that might be desired for applying more efficient preventive measures. Psaier et al. [66] propose a similar framework for mixed

interactions between humans and Software-Based Services (SBS). Schneider et al. [67] present a recent and thorough survey of self-healing systems and frameworks. Their survey shows that self-healing systems are becoming more autonomous, although this is partly attributed to more specialization. For example, different approaches are used for mobile and centralised computing environments. The most successful systems are also those that use heavily supervised methods, thus relying strongly on human interaction. This interaction can be in the form of failure detection, as well as in manual insertion of recovery and healing strategies. Some methods employ evolutionary programming techniques to dynamically build new recovery solutions, such as the Plato framework proposed by Ramirez et al. [68]. Self-healing systems, thus, are still heavily dependent on predefined assumptions. For example, in the case of the Plato framework, crossover and mutation operators must be defined (the authors give an example of dynamic remote data mirror reconfiguration). Furthermore, these systems do not allow application developers to leverage information regarding system failure. It is usually the job of a system administrator to define recovery actions, and no preventive actions are taken.

To the best of our knowledge, there has not been any proposal for an exception model that notifies application code of potential exceptions (i.e. a predictive exception model). However, Kim et al. [69] propose a proactive approach to exception handling within a business process. In their work, the authors note that business processes often involve "human exception handlers" that react to "exceptions". They conclude that there is a need for a proactive exception handling which allows for action as soon as a business process exception is "predicted", and that this action can be specified by an external agent (e.g. a system administrator) as a reaction to a prediction. Kim et al.'s work is, thus, similar to the work presented in this thesis, but differs significantly in the following key points:

- It concerns exceptions in a business process management context.
- It does not learn how to predict business process exceptions. Instead, a system administrator can build a set of rules similar to "$Necessary\_Roll\_Replacement > Available\_Roll\_In\_Supplier$". Thus, the prediction code is a set of rules determined from experience, only dependent on business conditions (i.e. this work cannot "predict" a database timeout).

Nevertheless, the core concept of PreX is the same as the concept seen in the work by Kim et al.: a shift from reactive exception handling is needed, in favour of more proactive behaviour. PreX focuses on the exception handling mechanism in the context of programming languages, and on systems which can use online failure prediction mechanisms to "learn" exceptions independently of human interaction.

2.5 Summary

There is a broad class of Online Failure Prediction methods which have been shown to produce good results in practical scenarios. Many of these methods are based on machine learning algorithms. Thus, some of the major challenges in failure prediction are associated with machine learning challenges – for each problem, there is a particular set of features deemed ideal, but it is not a trivial task to find this ideal set. Furthermore, failure prediction in the context of dynamic environments requires alternative methods which support re-training or can act on-demand with new data.

The Exception Handling mechanism has not seen much change throughout the years. Although there have been several alternative models proposed, in particular due to an increase in popularity of alternative programming paradigms, such as functional reactive programming, the base principle of modern exception handling is still the same – the mechanism reacts to an exception, it cannot prevent it. Online Failure Prediction methods show potential as the base for a paradigm shift in Exception Handling mechanisms where exceptions are no longer caught, but can be predicted on a fine-grained level.

There has been some related work in the context of self-healing systems and business process exception handling, but there is no automated fine-grained model for the prevention of exceptions. This thesis presents PreX, a preventive exception model that proposes that the system, as a whole, actively work towards predicting and preventing exceptions. Applications can then be more resilient, robust, reliable and have increased performance.

Chapter 3

PreX – A predictive model for exception handling

PreX is an exception model that focuses on preventing exceptions rather than catching them. The central idea was depicted with the example given in Section 1.1: it could be more efficient to temporarily reduce the throughput of a write-heavy application than to catch a *ConnectionTimeout* exception and have to restart the process. It makes sense that there are other scenarios, similar to this, where systems and developers would benefit from an easy-to-use proactive model for Exception Handling. We now present PreX.

Preventing exceptions implies predicting them. To this end, the area of Online Failure Prediction provides valuable insight. There have been successful failure prediction systems, but these operate on a broader level. In order to predict exceptions, the proposed model needs to adapt failure prediction techniques to a per-exception basis. Furthermore, since no two systems are alike, the prediction models will have to be trained for specific deployments. Thus, the PreX model comprises different phases from development to successful prevention:

1. **Coding phase**. The programmer develops the application using a new set of programming language constructs introduced by the PreX model. These are similar to traditional *try-catch* blocks seen in several languages.

2. **Training phase**. In this phase, different machine learning algorithms are applied (after feature selection, data pre-processing, etc), determining which is the most applicable to the specific exception. Data is gathered for different runs of the application, using resource monitoring facilities.

3. **Detection phase**. The application is deployed with the trained model and executed. The model is used to detect potential exceptions. If the trained model

becomes ineffective due to changes in environment conditions, the training phase might be required again. Alternatively, self-adapting pattern recognition algorithms can be used.

4. **Prevention phase**. If an exception is predicted, the application can be alerted to apply preventive measures and try to avoid the potential exception from effectively being raised.

In the following sections, each of these phases is detailed from the perspective of the syntax and semantics of the model, followed by an overall perspective of the architecture and the necessary components of PreX. Lastly, the behavior of the system during its different phases is presented.

3.1 Basic Definitions

PreX introduces several new concepts in the context of exception handling. In current exception handling models, exceptions are raised, caught and handled. In PreX, exceptions can also be **predicted** with some degree of confidence within certain blocks of code. These blocks are called **prediction blocks**. Whenever an exception is predicted to occur within a prediction block, the application is notified of this prediction through an **alarm**, which is said to be **triggered**. Program flow is then interrupted and transferred to a **prevention block**, where some action can be taken, after which program execution continues as normal (similar to the way the resumption model works). Note that this alarm is not an exception, nor does it propagate like exceptions do. It is, instead, an event that signifies a potential exception.

At times, it might be useful to request that the exception handling model wait until it triggers the alarm. For example, a set of atomic operations should not be interrupted. Programmers can specify these blocks, which are called **no-alarm blocks**.

In summary, PreX introduces the following new concepts:

- **Prediction Block** – The region of code where programmers wish to be notified of exception predictions (to which they want to react). This matches the *try* block seen in current exception handling mechanisms, which is why it is also called the **try block**.
- **Alarm** – The indication that an exception *might* happen (i.e. it has been **predicted**) within a prediction block. Alarms are **triggered** by the exception handling mechanism.

- **Prevention Block** – The code block to which control is transferred after an alarm is triggered. These blocks can contain any code, but will usually try to prevent the exception from happening.
- **No-alarm Block** – A block of code (within a prediction block) where no alarm can be triggered. If an exception has been predicted during the execution of a no-alarm block, the corresponding alarm is only triggered once flow is outside of the aforementioned block. There can be any number of no-alarm blocks.

There are other concepts relating to PreX that will be introduced in the next sections and that help with the overall understanding of how PreX works.

3.2 Syntax & Semantics

An example of the syntax of PreX is shown in Figure 3.1. In this example we use a variant of Java code with the proposed syntactic changes. In the following sections, the detailed syntax and semantics of PreX used in Figure 3.1 are described in detail. The highlighted sections of code are discussed in Section 3.4.

```
1  // Connect to database and build list of data to write
2  DBConnection c = connectToDB();
3  List<Write> writes = fetchWrites();
4
5  // Try to predict exceptions using the "write-exceptions"
6  // prediction context
7  try("write-exceptions") {                                    (a)
8      for (int i = 0; i < writes.size(); i++) {
9          // Feed writes left to the prediction system
10         sample("writes_left", writes.size()-i);              (b)
11
12         // Don't allow alarms to be triggered
13         no_alarm {                                           (c)
14             print("Writing data!");
15             // Write the data through the connection
16             c.write(writes.get(i));
17         }
18     }
19 } prevent ( ConnectionTimeout, info ) {                       (d)
20     // info has information about the prediction
21     // (e.g. lead time)
22
23     // Sleep for some time to try to prevent an exception
24     sleep(1000);
25 } catch ( ConnectionTimeout e ) {
26     // The exception could not be avoided...                 (e)
27     error("Database timeout!");
28 }
```

FIGURE 3.1: Example of PreX using synchronous *try-catch-prevent*.

3.2.1 The *try-prevent-catch* and *try-prevent_ async-catch* constructs

PreX provides semantics similar to traditional *try-catch* blocks, although two different constructs are added to the language. The first is the *try-prevent-catch* construct, as seen below:

```
1 try(<prediction_context>) {
2     // ... Prediction Block.
3     // Exceptions can be caught and alarms can be triggered
4 } prevent ( <exception_name>, <information_object> ) {
5     // ... Prevention Block.
6     // Execution follows the resumption model.
7 } catch ( ... ) {
8     // ... Exception Handling code
9 }
```

Listing 3.1: The *try-prevent-catch* construct

The *try* block denotes the scope during which a program cares about alarms regarding some particular exception (i.e. it is the prediction block). If an alarm is not raised, and if the exception cannot be prevented, it is raised and caught by the code, and the traditional termination Exception Handling model is used, with program flow being transferred from within the *try* block to the *catch* block.

The second construct added by the model is similar, but uses the *prevent_ async* keyword instead of the *prevent* keyword. When an alarm is triggered, it can transfer execution flow to the prevention block in two different ways, depending on the construct used:

- **Synchronously**: execution within the *try* block is suspended and flow is transferred to the *prevent* block. In normal circumstances, the execution is then transferred back to the previous code within the *try* block (resumption model).
- **Asynchronously**: execution within the *try* block continues normally, and the *prevent_ async* block is executed asynchronously.

Thus, in PreX, triggering an alarm does not necessarily terminate the execution of code within the *try* block (as opposed to the *termination* model when an exception is raised). Instead, execution is resumed as if there had not been any interruption, because preventing an exception should not halt normal execution of the current code.

When using a synchronous approach, alarms are not triggered at just any point in time within the *try* block. Within this block, every program statement will be executed without any interruption from the exception handling mechanism (unless exceptions are

raised). The flow of execution only moves to the prevention block between statements, without the need for propagation in the call stack. In this sense, PreX uses an approach similar to static binding, meaning that alarms are not "propagated" as if they were exceptions. Instead, they are tied to the prevention block to which they belong. This approach makes sense if we consider that only the closest code, within the exact context of the particular exception, can know how to react to a specific prediction. Listing 3.2 shows an example of where alarms can be triggered (A, B, C, D, E and F) .

```
1 try(<prediction context>) {
2     function1(); A
3     c = a + b; B
4     if ( c > 0 && d < a ) {
5         C
6         function2(); D
7         function3(); E
8     } F
9 }
```

LISTING 3.2: Alarms can only be triggered in between statements (A, B, C, D, E and F)

3.2.2 The *no_alarm* keyword

If developers wish to prevent the triggering of alarms during the execution of a set of statements, a special *no_alarm* keyword can be used to denote a new scope within which alarms are not possible – the no-alarm block. In Figure 3.1, lines 13-16 belong to one of these scopes. In the asynchronous approach, as in the synchronous approach, alarms are only triggered when the program flow is outside the protected block. Thus, the *no_alarm* keyword can act as a synchronization primitive between code in the prevention block and code within the *try* block.

```
1 no_alarm {
2     // No-alarm block! No alarm can be triggered here.
3 }
```

LISTING 3.3: The *no_alarm* keyword starts a no-alarm block where alarms can't be triggered

3.2.3 The prediction context

Since no two systems are alike, the models used for prediction will have to be trained for specific deployments. Note that, for instance, *ConnectionTimeout* exceptions may be

different depending on the workload and the different systems involved. A write-heavy workload will have different characteristics when compared to a read-heavy workload. This motivates the need to distinguish between different kinds of predictions of *ConnectionTimeout* exceptions. This distinction is made by an argument to the *try* keyword that uniquely identifies the block of code that it encloses: *try(<prediction context>)*.

```
1 try(<prediction_context>) {
2     ...
3 }
```

LISTING 3.4: The *try* block accepts a *prediction context* as its argument

Write-heavy blocks can then use *try("write-exceptions")*, whereas read-heavy blocks can use *try("read-exceptions")* to handle two completely different models for the same kind of exception (*ConnectionTimeout*) under different contexts. This argument is a string called the **prediction context**. Several exceptions can be predicted within the same *prediction context* (and corresponding alarms triggered), and a *prediction context* binds training data and a prediction model to a unique name. To train the model, the system administrator may specify which *prediction context*s he/she wants the program to be trained in during a training phase. When in this training phase, no prediction mechanism is used in those *prediction context*s (i.e. no alarms can be triggered), although exceptions can still be caught. If the exception is raised and caught, this training mechanism is notified to adapt its prediction models.

3.2.4 The *sample* keyword

PreX allows programmers to periodically sample variables that they think will be useful for prediction, in addition to system variables monitored with custom probes. For instance, the remaining number of operations left might be useful in predicting connection timeouts. These variables can be supplied to the prediction system at any time using the *sample (<variable_name>,<variable_value>)* construct.

```
1 sample(<variable name>, <sample value>);
```

LISTING 3.5: The *sample* keyword can feed data into the prediction system within any prediction block

Note that if the *sample* keyword is used when in the training phase, it still feeds data to the prediction system.

3.2.5 The prevent block and the prediction information object

The prevent block is introduced with the *prevent* keyword. It requires two arguments:

- The first is the name of the exception to prevent, so that different prevent blocks be assigned to the same *try* block, similarly to how there can be different *catch* blocks in most programming languages.
- The second is the name to assign to the **prediction information object** within that prevent block. This object is created by the exception prevention mechanism and contains information regarding the alarm that has been triggered, such as the lead time, the data validity time or specific information regarding the prediction method chosen by the mechanism. This information might be used by application code to apply different prevention techniques.

In Listing 3.6, an example prevent block is shown for *TimeoutException* exceptions. The prediction information object can be accessed within the prevent code with the name *predInfo*.

```
1  prevent ( TimeoutException, predInfo ) {
2      // Block for preventing TimeoutExceptions, when
3      // triggered by an alarm.
4      // predInfo contains prediction information such a lead-time.
5  }
```

LISTING 3.6: Example prevent block, which requires the name/type of exception and a name to assign to the prediction information object

3.3 Architecture

PreX's architecture is suited for three different scenarios:

- **Prediction Scenario**: in which a set of models has already been trained and is used at run-time to predict failures.
- **Training Scenario**: in which no prediction happens and the models are trained for future usage (coinciding with the training phase).
- **Combined Scenario**: in which the previous scenarios might happen at the same time, providing a suitable system for dynamic environments where models must be retrained and conditions change.

Some components of PreX are shared in all of these scenarios, while some of them are only active in a particular scenario. Exceptions can be predicted using system-wide information. Several entities can share information used for prediction (more data, when appropriately filtered, implies better predictions). An entity need not be deployed in an independent machine on its own, and different entities might share the same machine. There are three main kinds of entities within a system using PreX:

- **Data gathering entities**: These entities feed periodic samples of data (e.g. memory and CPU usage) to the coordinator entities. Most of the prediction data comes from these entities, which don't execute any code that wants to be alerted of possible exceptions. The sampling rate for each of these variables is not pre-determined and may vary according to system load and characteristics.
- **Code entities**: Whenever a *try* block is entered, the exception prevention mechanism spawns a code entity that connects to the coordinator entity. Each code entity may want to register with the coordinator to be notified of predictions of certain kinds of exceptions within a *prediction context*. It is the responsibility of the code entity to trigger the alarm and transfer execution to the prevention block.
- **Coordinator entities**: A (potentially replicable) coordinator entity, responsible for aggregating the data from the other two types of entities and running the prediction system.

Each *try-prevent-catch* or *try-prevent_ async-catch* block spawns a new code entity. The coordinator entity is responsible for running the failure prediction methods for predicting exceptions. This behavior is described in Section 3.4 in more depth.

The behaviour of the code and coordinator entities is different during the training phase. During this phase, the code entity registers that it will be sending data and information regarding a given *prediction context* and a given exception. The coordinator then uses this data to train the model.

The different entities can be freely distributed among machines, and might all run on a single machine, although this might have a performance impact on the overall system. A typical deployment scenario is depicted in Figure 3.2, where the Coordinator Entity is allocated to an individual machine. Note, however, that all entities might be executed in the same machine, and that Figure 3.2 is merely an example.

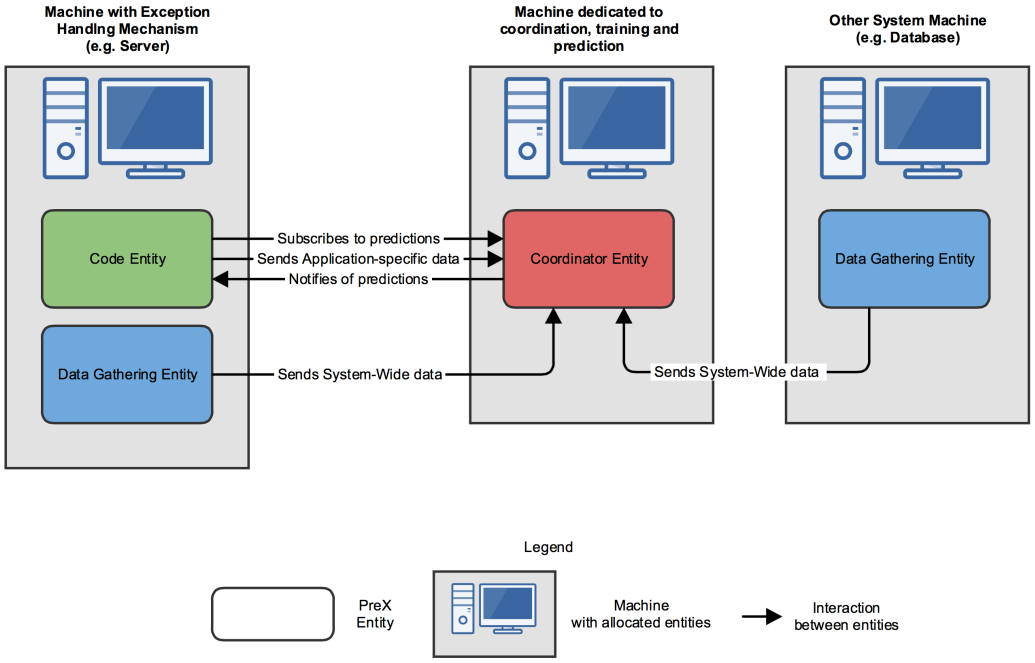

FIGURE 3.2: Example deployment of entities in a system. Another example involves deploying all entities in the same machine

Lastly, PreX is designed with the idea of extensibility. As such, implementations should offer an open protocol for interaction with the coordinator entity is specified so that other developers can build their own probes that feed data to the prediction system. Furthermore, this modularity means that if a new feature is ever needed, the data gathering entity does not need to be rebuilt and redeployed – instead, a new probe can be written, and data sent according to the open protocol. Security concerns can also be added atop this open protocol and are not the focus of the work in this thesis.

3.3.1 Prediction Scenario

In the **Prediction Scenario**, PreX's architecture is as depicted in Figure 3.3.

Several different **Probe**s might be executed in different machines of the system. These probes have a **System Data Gatherer**[1], responsible for gathering many different metrics, such as CPU, Memory, Disk and Network usage. This data is forwarded to a **Data Communication Module**, which transforms it and filters it to conform to a predefined standard, finally sending it to the Coordinator Entity. The Data Communication Module is independent of the System Data Gatherer.

[1]Note that any kind of probe can be used with PreX, and "System Data Gatherer" probes do not need to be limited to CPU, Memory and related information. When PreX is deployed in a system, the most appropriate system variables must be sampled with probes and fed to the mechanism.

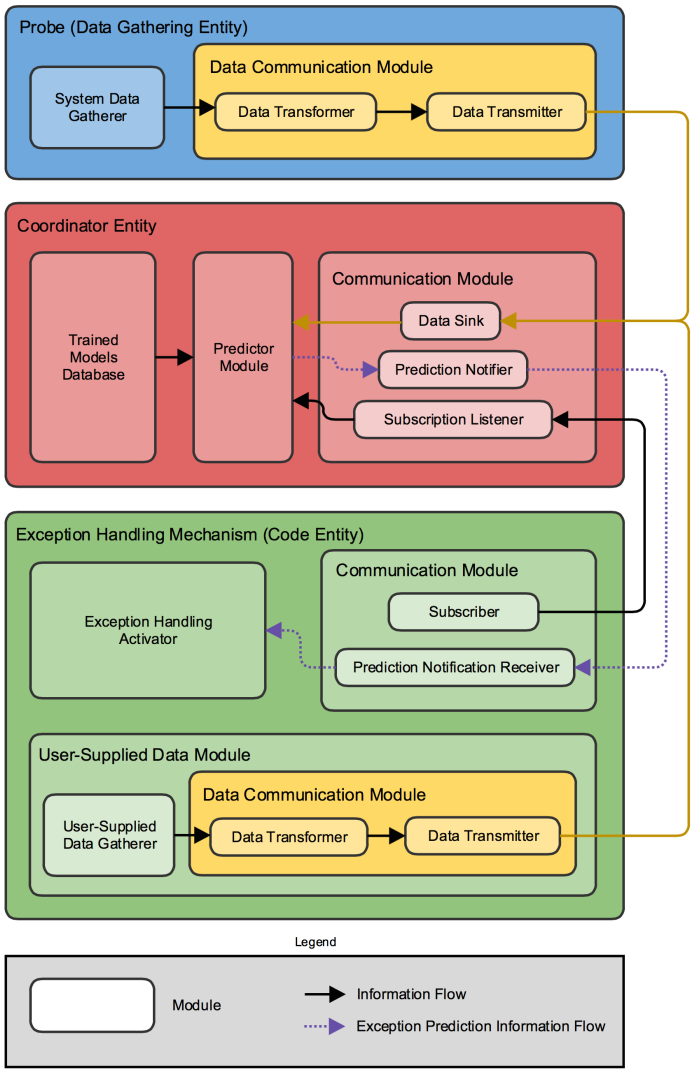

FIGURE 3.3: Components and Connectors diagram depicting PreX's architecture for the Prediction Scenario.

The **Coordinator Entity** is composed of three main modules:

- A **Trained Models Database** from where the models are read.
- A **Predictor Module**, which processes the data available and, using the trained models, determines if an exception has been predicted or not.
- A **Communication Module**. This module acts as the bridge between the Coordinator Entity and other entities. A **Data Sink** receives data and passes it to the Predictor Module. A **Prediction Notifier** module forwards prediction notifications to Code Entities. Lastly, a **Subscription Listener** awaits for Code Entities to notify that they intend to listen to some prediction, thus activating the Predictor Module.

Finally, the **Code Entity**, corresponding to the Exception Handling Mechanism itself, is composed of three major components:

- The **Exception Handling Activator**, responsible for running programmer-specified code whenever an exception is predicted, ensuring correct program state. This is the module that triggers an alarm and transfers execution to the prevention block.
- The **Communication Module**, acting as the bridge between this entity and the remaining entities. A **Subscriber** module informs the Coordinator Entity of the prediction context and exceptions that the Code Entity is interested in predicting. A **Prediction Notification Receiver** listens for exception prediction notifications from the Coordinator Entity, passing them to the Exception Handling Activator.
- The **User-Supplied Data Module**, a module very similar to a Probe, with the difference being that, unlike with Probes, this module gathers data supplied by the application developer using the *sample* keyword. Note how the **Data Communication Module** is the same in the Probe and User-Supplied Data Module.

3.3.2 Training Scenario

In the **Training Scenario**, PreX's architecture is as depicted in Figure 3.4.

This scenario shares some components with the previous scenario, but introduces additional ones. When training, data flows exactly as in the previous scenario. Thus, the **Probe** and the **User-Supplied Data Module** (of the Code Entity) are unchanged. Similarly, in the **Coordinator Entity**, the **Data Sink** does not change. The **Trained Models Database** is now the destination of the output of a **Training Module**, responsible for building new models. In the Code Entity, there is no longer the need for a Subscription Module, although exceptions must be detected with an **Exception Detector** and passed to the Coordinator Entity through an **Exception Notifier**.

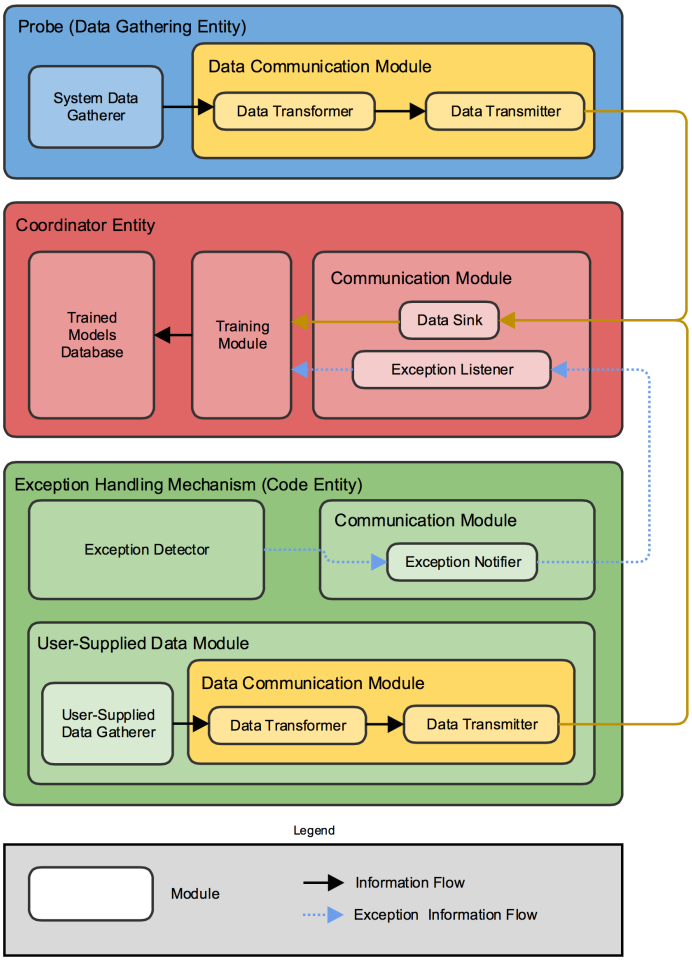

FIGURE 3.4: Components and Connectors diagram depicting PreX's architecture for the Training Scenario.

3.3.3 Combined Scenario

In the **Combined Scenario**, PreX's architecture is as depicted in Figure 3.5.

This scenario is the combination of the two previous scenarios, articulating all modules so that trained models can be retrained at run-time. If an exception is not prevented (even after being predicted), then it will eventually be raised and detected in the **Exception Detector**, which ultimately passes this information to the Coordinator Entity. Thus, new data and information about exceptions are used to train new models at run-time. This scenario is ideal for dynamic environments where base conditions change.

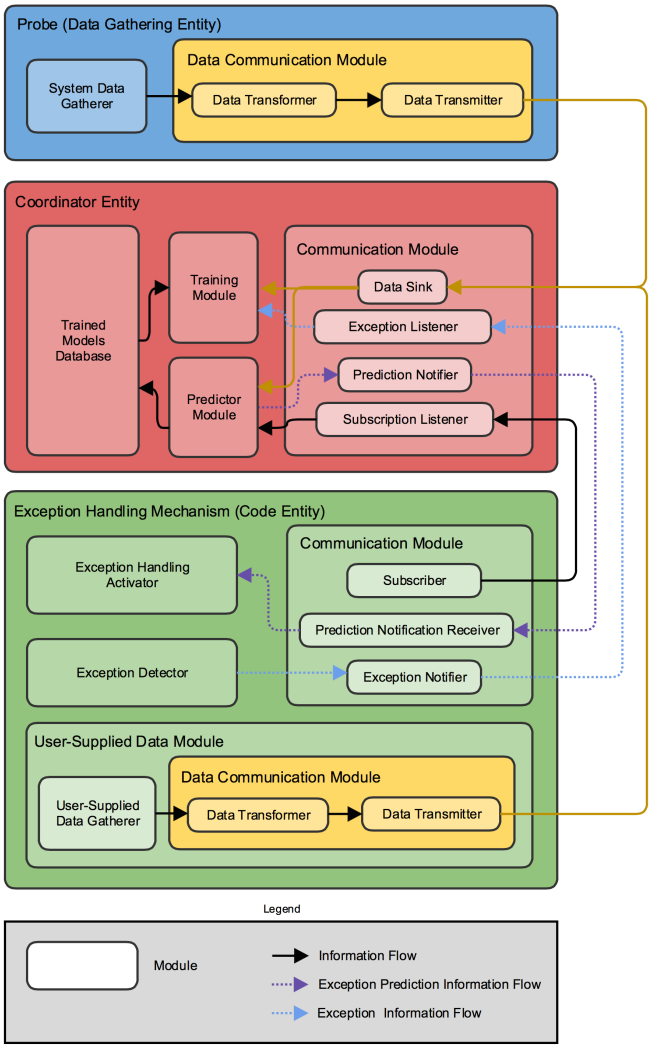

FIGURE 3.5: Components and Connectors diagram depicting PreX's architecture for the Combined Scenario.

3.4 Model Behaviour

To illustrate the behaviour of PreX, Figure 3.6 depicts the interaction between code, entities, and the prediction system for the code example in Figure 3.1 shown in a previous section. Notice that the code uses the synchronous prediction model. Also note that if the code was executed during the training phase, no alarms would be triggered, so section *d* would not be entered.

The example Listing 3.1 allows understanding of how the overall system behaves when deployed. In the example scenario, several typical data gathering entities would be placed: one at the database machine and one more for each of the client machines (executing the code). Furthermore, a coordinator entity might be placed somewhere within the system.

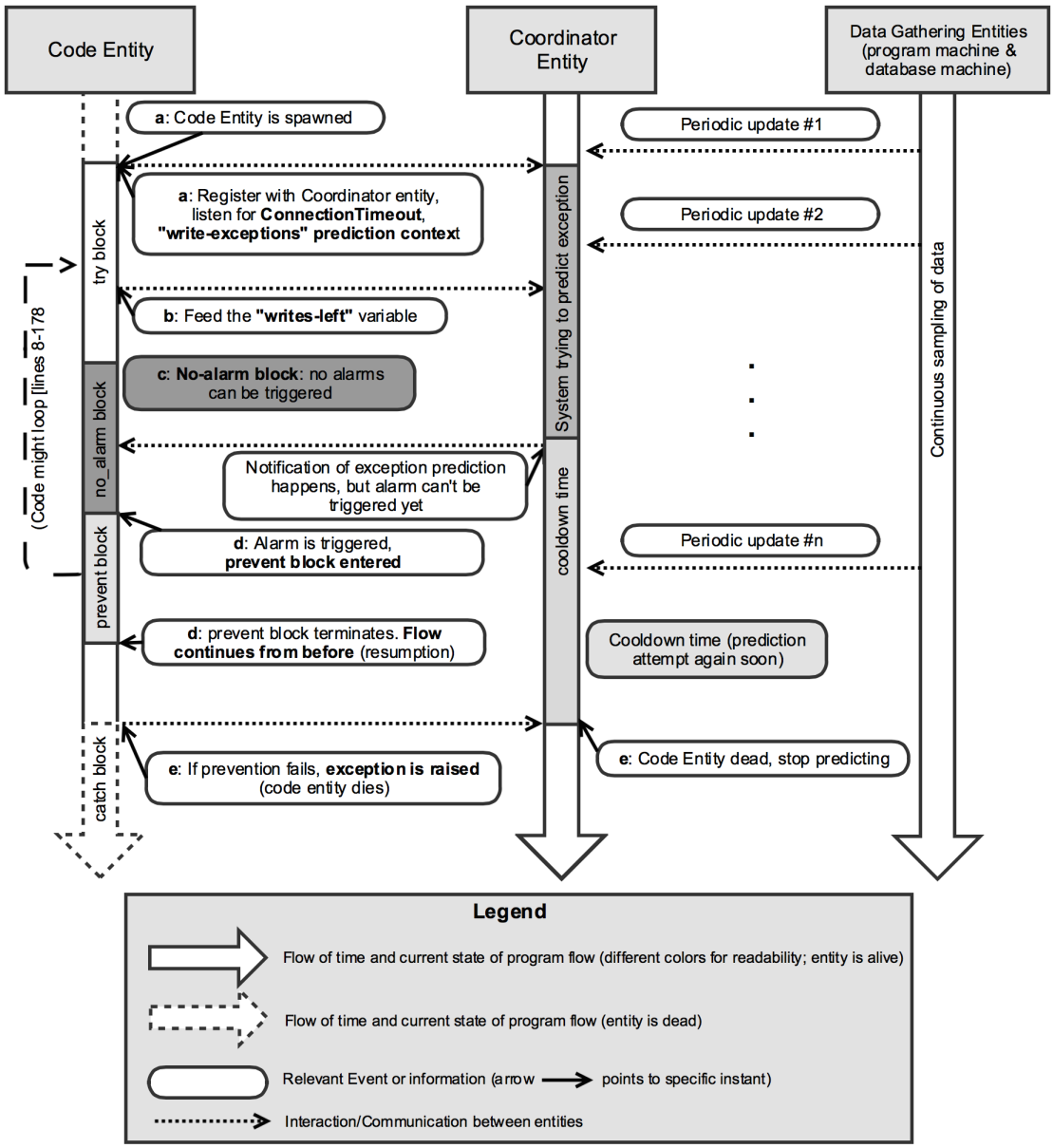

FIGURE 3.6: Interaction between code and entities through time for the code example in Figure 3.1. For simplicity, in this example, the loop only runs once.

When line 7 is reached, a code entity is spawned by the exception handling mechanism, registering with the coordinator (through the Subscriber module) that it wants to be warned of *ConnectionTimeout* exceptions within the "write-exceptions" *prediction context*. The coordinator node begins predicting exceptions based on the data being gathered by the data gathering entities (probes). At line 10, the *sample* keyword is used, making the code entity send a sample of data to the coordinator entity (which now uses this data together with the data form the data gathering entities). Normal operation continues at line 13.

Line 13 starts a *no_ alarm* block, indicating that even if notifications of exception predictions are received, alarms cannot be triggered (thus guaranteeing that it is executed atomically in this regard). In Figure 3.6, an exception is predicted during execution of the *no_ alarm* block. This information is sent to the *Code Entity*, which delays activating the Exception Handling mechanism. Once at line 17, outside of the *no_ alarm* block, the code is interrupted and execution continues in line 19, the prevent block (an alarm is triggered). A sleep instruction is executed in hopes of reducing the workload on the server (thus preventing the exception) and execution continues exactly where it stopped (resumption model), at line 18. This sequence of steps can happen many times during execution.

In the case that the exception could not be prevented, it may be raised at line 16 (the *write* method is the one that throws this exception), transferring program flow to line 25 as with the traditional Exception Handling mechanism of current programming languages. In either case (prediction or catching), the code entity terminates its execution at the end of the *try-prevent-catch* block, notifying the coordinator that it is no longer going to be interested in notifications.

If the *prevent_ async* keyword had been used, the behavior would be similar, but code in the *prevent* block would be run asynchronously and in parallel. The *no_ alarm* keyword can act as a synchronization primitive between the prevention handler and the code within the *try* block, guaranteeing that alarms would only be triggered once the main program code finished the *no_ alarm* block.

Consider now that the prediction model has not been trained yet. To perform training, no changes to the code are required. A system administrator selects the "write-exceptions" *prediction context* for training and runs the application. In this case, reaching line 7 still spawns a code entity, but it is only to feed data regarding the "write-exceptions" *prediction context* and the *ConnectionTimeout* exception. At line 10, data is sent to the coordinator as normal. Line 13 has no effect, since exceptions are not being predicted, and the only way of exiting the *try* block scope is to either successfully terminate the operation (i.e. a golden run) or by catching an exception. In both cases, the code entity notifies the coordinator and terminates itself afterwards.

3.5 Training and Prediction Methodology

PreX supports several prediction methods. Individual methods can be trained on the gathered data, and the most appropriate one can be selected. Nevertheless, PreX prediction methods have to operate on a basic set of features and follow a certain technique. For instance, it is different to use classification methods than it is to use regression methods. In this section we present the approach taken by PreX for prediction.

3.5.1 Features

As noted previously, Data Gathering Entities, as well as Code Entities, supply the prediction system with data. Each data point fed to the Coordinator Entity is characterized by a system-local timestamp and the data itself. The Coordinator Entity distinguishes features sent from different machines (the CPU Usage in machine A is a different feature than CPU Usage at machine B, even if measured at the exact same time). Each Data Gathering entity can sample different kinds of variables, such as those regarding CPU Usage, Network statistics, Disk usage/errors or Memory/Swap Usage. In total, this can amount to dozens of variables being sampled regularly (for example, $100ms$). All data is numeric or binary. As an example, in the preliminary results presented in Appendix B, 49 variables were sampled from one machine.

The PreX model (and the implementation presented in this thesis) are not harcoded to any features or probes. Different scenarios require inherently different data and, since the PreX protocol is open (see chapter 4), any probe can provide additional features used for prediction. Examples of features include: the CPU usage, the number of users logged into a machine, the most invoked HTTP method of a web-service (e.g. POST, PUT or GET).

3.5.2 Prediction Method

In Section 2.2.3, a classification of Online Failure Prediction methods proposed by Salfner et al. [5] was presented, together with several examples of applied prediction methods in published works. PreX's prediction method heavily influences the kind of algorithms and their performance, as well as the data pre-processing steps to be taken. As can be gathered from our chapter on the state of the art, there seems to be no definitive method for online failure prediction. Prediction algorithms can be considered as a kind of optimization problems where a number of input varibles (i.e. "features") are used to model the state (failure/no-failure or a probability of failure) of the system. Depending

on the type of failure and system, these input variables and their correlation with the external failure state of the program can vary greatly. In this sense, the "No Free Lunch Theorem" of Wolpert and Macready [70] shows that, indeed, no algorithm will be better than every other algorithm for all different kinds of predictions. Thus, if PreX intends to be broad, it must support several different prediction methods.

Different methods require different types of data. Some methods, such as the Eventset method proposed by Vilalta et al. [36], discard most temporal information regarding events. These methods also use Detected Error Reporting techniques (see Section 2.2.3.3). In the case of PreX, the only kind of error reporting is the exception itself, so these approaches are not appropriate. Additionally, PreX aims to include the time dimension within its prediction system, since we believe it is a critical part of the prediction process (Duetterich [71] and Irrera et al. [27] have shown the importance of the time dimension). Other methods (e.g. [29], [26] and [30]) use statistical tests to compare feature distributions with error-free states or error states. These statistical methods could be applied to the kind of data gathered by PreX, but they also make it harder to include the time-dimension for prediction. Additionally, these statistical methods often require more *a priori* knowledge of the system so that the statistical models can be built. Other approaches (e.g. [15], [16], and [18]) use regression and function approximation techniques. PreX could implement this approach, although it is more adequate to resource exhaustion scenarios and might not be appropriate for all kinds of exceptions that PreX intends to predict.

A failure prediction problem can also be modeled as a classification problem. These problems can be solved by state-of-the-art classifiers, and many authors have used classification methods for Online Failure Prediction (e.g. [22], [24], [25], [26] and [43]). Irrera et al. [27] presented a sliding window approach to incorporate the time dimension in a classification failure prediction problem. Their data was similar to the data used in this thesis (see Chapter 5) and their approach, which applies techniques presented by Duetterich [71], is the inspiration for our approach – PreX treats failure prediction as a **classification problem**.

3.5.3 Failure Prediction as a Classification Problem

In this section, we present the failure prediction problem as a classification problem. In addition, we present the rationale that lead to the final "feature set" used in PreX, which we further elaborate on the next section. This "feature set" does not specify which individual features (e.g. amount free memory or disk accesses) are used, but, instead,

specifies how to transform any amount of these individual features into a dataset ready for failure prediction.

If we consider that it is possible to sample data at precise instants $t_1, ..., t_m$, separated by intervals of time T, then at each time t_i, a **feature-vector** comprised of n features can be defined as:

$$x_i = (x_{i,1}, ..., x_{i,n})$$

This vector represents the system state at time t_i and has a corresponding label vector $y_i \in \{0, 1\}$ where 1 indicates a failure and 0 indicates a failure-free state. Over intervals of time T, there can be several feature vectors x_i and labels y_i that characterize the system state. A classification problem can then be formulated to, given each feature vector x_i, determine the hidden-class of the corresponding label $y_i \in \{0, 1\}$. Each individual feature $j \in 1, ..., n$ is denoted by $x_{i,j}$ (feature j of feature vector at time t_i).

However, this model has several limitations. The first of these limitations is the assumption that data can be sampled at precise instants. PreX's data gathering entities do not guarantee that data is sampled at exact times. One solution to this problem involves using approximate values. For example, if there is no value for $t_i = 0.3s$, but there is one for $t_i = 0.1s$, we can use that value. The problem with this approach is that some variables might have more plausible approximations than others, and simply "picking the closest one" might introduce error in the data[2]. An alternative, which we propose in PreX, is to use a window of size T to summarize the information regarding the features in the feature vector. If between $t_i = 1s$ and $t_{i+1} = 2s$ ($T = 1s$) there are many different values for feature j, one could take x_i to be the mean of these values (thus averaging feature j over the window of length T starting at instant t_i). A "simple" average might hide information useful in prediction, and PreX proposes that for each feature j, an additional set of features be constructed for each of the windows. Although implementations of the model may vary, possible features can be the *mean*, *standard deviation*, *maximum*, *minimum* and the *derivative*[3]. Thus, for each original feature j (e.g. CPU Usage), which can be sampled an arbitrary number of times within the interval $[t_i; t_{i+1}[$, a total of 5 features are generated. By introducing this step, we have developed an **augmented feature vector** (and a corresponding **augmented feature window**, or **time-window**) with five times the number of features (which we call the **augmented features**), representing a time interval instead of a specific time instant. We can apply

[2]This problem is, in a sense, similar to the problem of missing value imputation.

[3]We define the "derivative" as the quotient $\frac{x_f - x_s}{t_f - t_s}$ where x_s is the first measured value of the feature in the time window (measured at time t_s) and x_f is the last value of the feature in the time window (measured at time t_f).

the classification problem to the augmented feature vector, provided that the definition of the label y_i is "the program state (failure/non-failure) during the window $[t_i; t_{i+1}[$". We call the process that builds the augmented feature vector from the original data the **time-window construction**.

A second problem with the aforementioned model is the lack of time dimension for failure prediction. Samples are not correlated and are specific to time t_i. The proposed changes, with the augmented feature vector, already incorporate a sense of time, in particular the derivative feature. However, this information alone does not provide enough information about the time dimension. As a solution, we propose to apply the sliding window presented by Irrera et al. [27] to our augmented feature vector. Thus, for any window $[t_i; t_{i+1}[$ with augmented feature vector x_i, we can construct a sliding window w_i:

$$w_i = (\underbrace{x_{i-(k-1),1}\ , ...,\ x_{i-(k-1),n}}_{x_{i-(k-1)}}\ ,\ \underbrace{x_{i-(k-2),1}\ , ...,\ x_{i-(k-2),n}}_{x_{i-(k-2)}},\ \underbrace{...}_{...},\ \underbrace{x_{i,1}\ , ...,\ x_{i,n}}_{x_i})$$

Where k, the size of the sliding window (which we call the **merge window**), is the number of windows to be concatenated in window w_i. We call this step **window-merge**. With both of these steps, the final feature vector, w_i, is constructed. This provides a data validity time $\Delta t_d = T \times k$.

For our classification problem, a label l_i is necessary. We have defined w_i as the concatenation of several augmented feature vector windows $x_{i-(k-1)}, ..., x_i$, but have not defined what to do with their labels $y_{i-(k-1)}, ..., y_i$. Irrera et al. consider only the last value y_i, discarding potential failures measured previously. An alternative would be to define $l_i = 1$ if *any* of $y_{i-(k-1)}, ..., y_i$ was 1 (in other words, a merge window would be labeled as "failure" if any of its concatenated windows was labeled as "failure"). Both of these approaches tie the merged window to a failure within that window of size $T \times k$, which implies that, at run-time, they do not allow us to predict failures outside the scope of this window. To compensate for this, we change the label l_i of w_i to be 1 if there is a failure in any of the augmented feature vectors of the *next* merged window and 0 if there are none. Hence, the lead time, Δt_l is T (the difference in time between merged windows). A further generalization that allows for higher values of Δt_l can be achieved by defining l_i to be 1 if there is a failure in any of the augmented feature vectors of the $t - th$ next merged window, making $\Delta t_l = t \times T$. The prediction period is $\Delta t_p = T$.

3.5.4 Overview of feature set construction

In the previous section, we presented the rationale behind most of the construction of the feature set to be used with a classifier. In this section, we formalize this process.

Data Gathering Entities gather data at arbitrary times (usually with some desired frequency, such as 100 ms or 1000 ms). This data is condensed in windows of size T, called **augmented feature windows**, which are represented by **augmented feature vectors** $x_1, ..., x_n$. Window x_1 contains condensed/augmented feature information (in the current version of PreX, as detailed in Chapter 4, the *mean*, *standard deviation*, *maximum*, *minimum*, *number of samples* and *derivative* of the data are taken) for the interval $[0; T[$. Window x_2 contains condensed/augmented feature information for the interval $[T; 2T[$ and so forth. If a failure/exception was recorded within the timespan of a window, that window is considered as a failure window. If there are m variables (e.g. CPU Usage, Memory Usage, etc), the augmented feature vector is of size $5 \times m$, since each variable produces 6 *augmented features*. This step, the **time-window construction**, is shown in Figure 3.7.

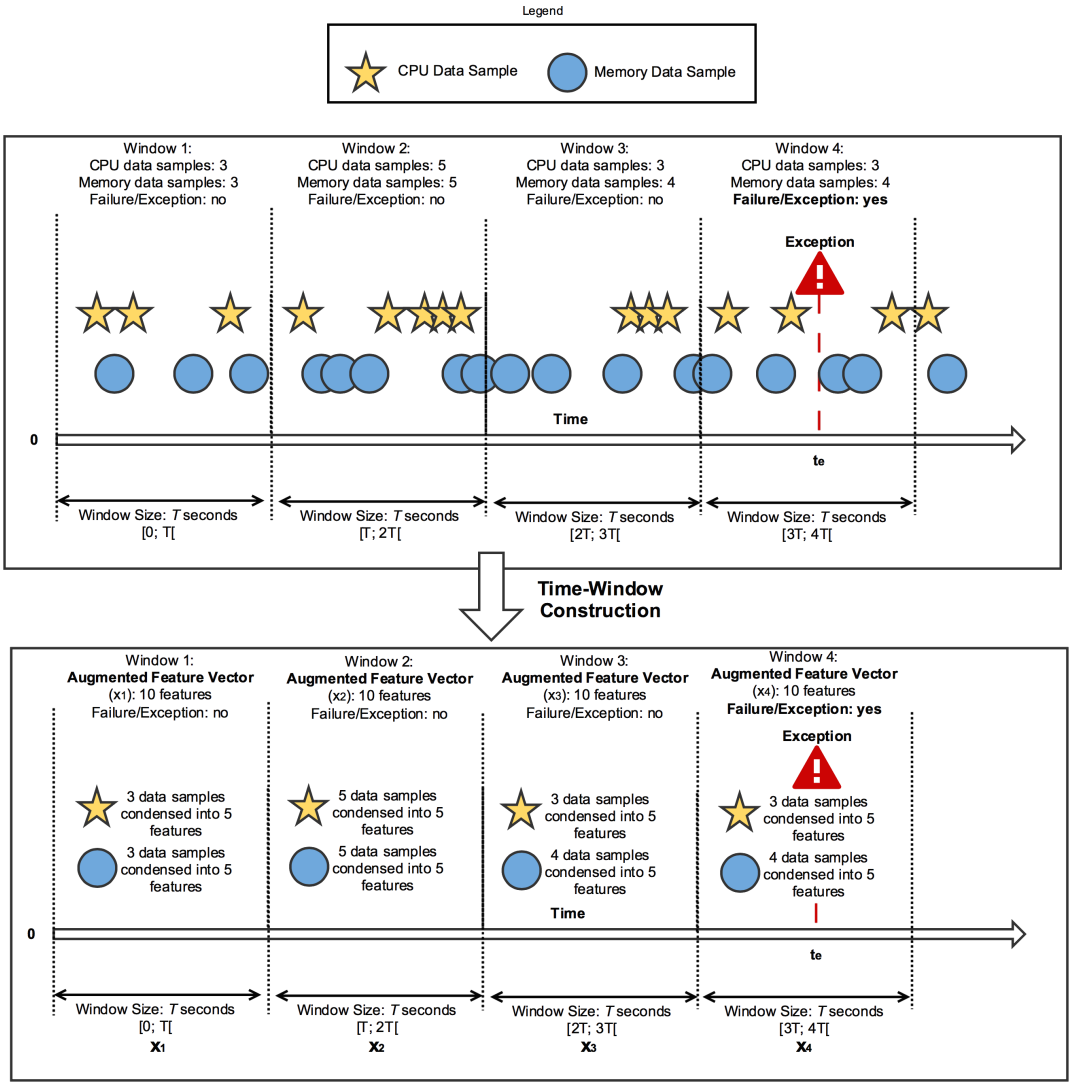

FIGURE 3.7: Time-Window Construction groups and summarizes variables according to "time windows" of size T. Note that in this figure there are 5 summarized features instead of 6.

The augmented feature windows are then merged in the **window merging process**. A configurable parameter, $k \in \{1, \ldots\}$, specifies how many windows should me merged. The input vector w_i is formed by taking

$$w_i = (\underbrace{x_{i-(k-1),1}, \ldots, x_{i-(k-1),n}}_{x_{i-(k-1)}}, \underbrace{x_{i-(k-2),1}, \ldots, x_{i-(k-2),n}}_{x_{i-(k-2)}}, \underbrace{\ldots}_{\ldots}, \underbrace{x_{i,1}, \ldots, x_{i,n}}_{x_i})$$

and using a sliding window process for each of the original augmented feature windows. Each window w_i is then assigned a label, l_i, defined as

$$l_i = \begin{cases} 1, & \text{if there was an exception in the } t\text{-th next augmented feature window} \\ 0, & \text{otherwise} \end{cases}$$

where $t \in \{1, \ldots\}$ is another configurable parameter. This step is shown in Figure 3.8.

A classifier can then be trained to predict the labels l_i based on the windows w_i, ensuring a lead time $\Delta t_l = t \times T$ with a data validity time of $\Delta t_d = T \times k$ and a prediction period of $\Delta t_p = T$. An implementation of PreX trains several different classifiers using this input data and chooses the best for each prediction context[4]. Thus, the configurable data for feature set construction is given by the tuple

$$(T, k, t)$$

where T is the window-size, k is the number of windows to merge/concatenate and t is a "look-ahead" parameter to determine how much in advance a failure is to be predicted. Obvious trade-offs exist among these parameters, the classification performance and accuracy. These trade-offs were studied during the development of the thesis, and can be seen in Chapter 5 and Appendix B. Also note that for the same dataset, higher values for k reduce the number of instances available for classifier training, potentially impacting accuracy.

The example shown in Figures 3.7 and 3.8 presents a representation of the feature set construction process with $k = 2$, $t = 1$ and 2 features. Also note that in these figures, the number of augmented features is 5 instead of 6, although the algorithm is the same.

[4]Fore more information regarding the choice of "best classifier", see Chapter 4.

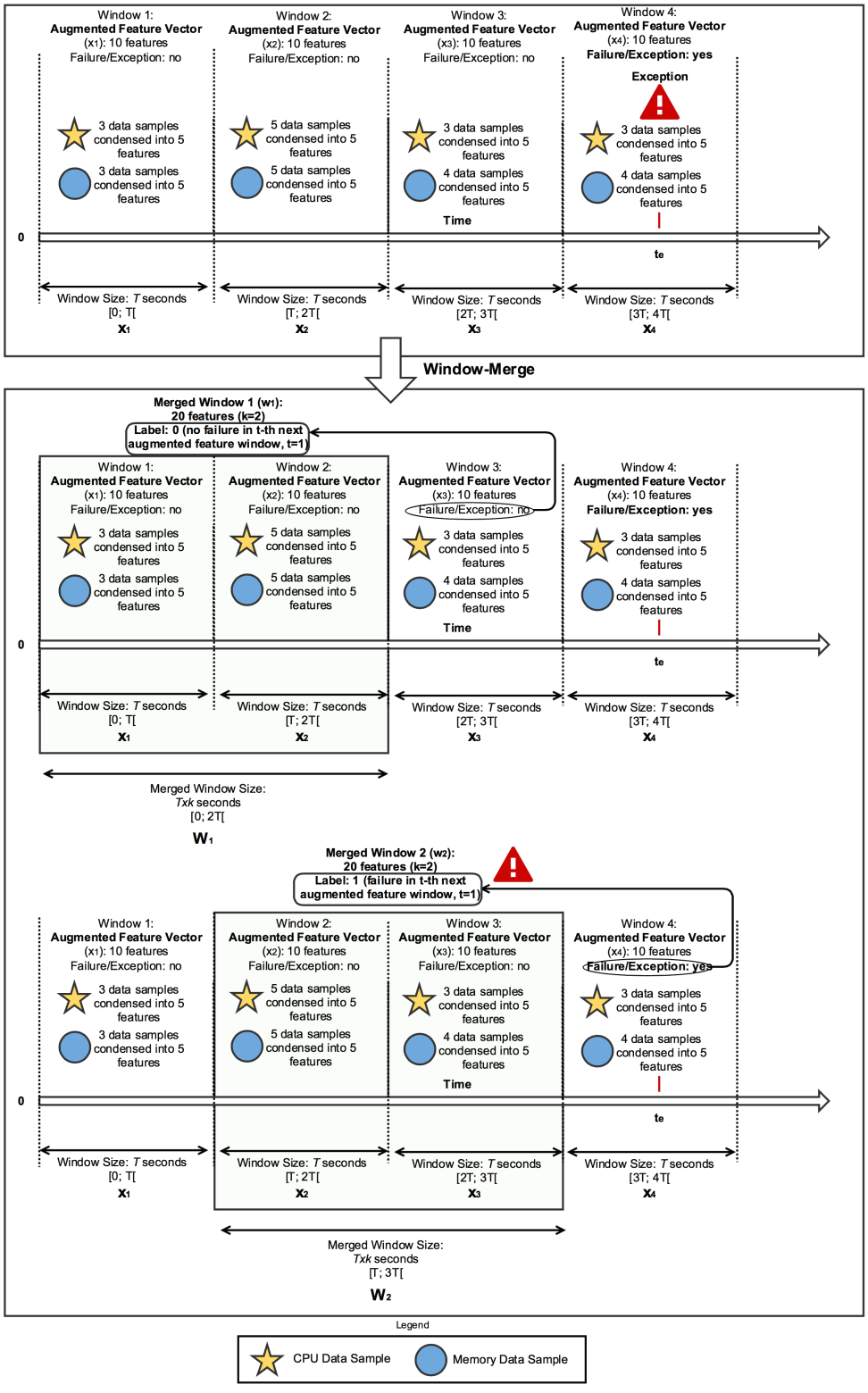

FIGURE 3.8: Window Merge merges augmented feature windows using a sliding window process. Each of the sliding windows, composed of merged augmented feature windows, is then passed to a classifier. In this example, $k = 2$ and $t = 1$. Note that in this figure there are 5 summarized features instead of 6.

It should be noted that data will not always be available (some windows might have no data at all regarding some features, or the user-supplied variables might be supplied irregularly). Thus, a missing value imputation method can be used. Implementations of PreX might consider using one of the following methods:

- Imputation with global average (if the source distribution is random, then replacing missing values with the average of that feature will not introduce any bias [72]).
- Imputation with average of last n samples.
- Imputation with global median (the median is a more outlier-robust approach [73]).
- Imputation with median of last n samples.
- Imputation using the K-nearest neighbours algorithm using the remaining available features as a model (as proposed by [74]).

Finally, a form of feature selection can be used. As we have seen, after the time-window construction and window-merge steps, the number of features increases dramatically (by a factor of $k \times t$), and the classification problem might become unfeasible. Several strategies can be used for feature selection, such as the three-step feature selection process seen in [27]: (i) removal of null/constant features; (ii) using a classical linear correlation metric (such as the Pearson correlation coefficient); and (iii) a classical approach involving wrapper or filter methods. Alternatively, a strategy similar to the one proposed by Irrera et al. [46] can be used, where changes in features are compared between failure and failure-free executions of a program, based on the concept of a symptom. However, note that PreX is not a generic failure prediction system: it is designed for an application-specific domain and, as such, the variables used for prediction can be specified for each different prediction context, reducing the amount of features used in the training process. In Chapter 4, the approach taken by our PreX implementation regarding missing value imputation and feature selection is described.

3.6 Preliminary Experiments

The main idea behind PreX is that an alarm of a potential exception might offer significant advantages for developers. However, as discussed in the state of the art (Chapter 2), there has been no work done on the prediction of exceptions at such a fine-grained level. A first step in the validation of the model, hence, involves verifying if it is possible to reliably predict exceptions.

A set of preliminary experiments, shown in full detail in Appendix B were performed to assess the predictive capabilities of the model. These preliminary experiments used a scenario where exceptions were triggered within the TPC-W benchmark. These experiments only intended to validate the main principles of the model, and did not involve a full implementation of PreX as specified in previous sections.

The results of these experiments demonstrated that, indeed, it was possible to predict exceptions, and, consequently, that the first part of the PreX model – prediction – was sound. In these preliminary experiments, a J48 classifier achieved a false negative rate of 9.90% for $T = 2500$, $k = 4$ and $t = 1$, while maintaining a very low false positive rate (0.12%).

3.7 Simulation

The PreX model involves two distinct stages: prediction and prevention. The latter stage is heavily dependent on the application domain, meant to be written by the programmer (when an alarm is triggered). Nevertheless, it is important to assess the usefulness of the proposed model with regards to prevention. If prevention is not useful, then prediction also loses its uselfuness. In this section, we present a simulation experiment used to validate the prevention capabilities of PreX.

Using simulation tools, we can quickly change model parameters and assess their overall impact on the system. For example, it is possible to assert the impact of too many false negatives on the system. Thus, without the need for a full experiment, it becomes feasible to evaluate the model's prevention capabilities, insofar as we are aware of the simulation limitations. In the next sections, we present the simulated scenario, the simulation parameters, its implementation, limitations and results.

3.7.1 Simulated Scenario

The simulated scenario closely resembles the one presented in the preliminary experiments mentioned in the previous section and shown in Appendix B and was developed as an attempt of modeling this scenario. A number of simulated clients are running concurrently, overloading a simulated server that accesses a database (which, in our preliminary experiments, was the TPC-W Server).

In the case of TPC-W, used in the preliminary results seen in Appendix B, exceptions happen due to connection pool exhaustion. However, real-world applications are often

protected against this kind of situation, so we opted by modeling the exceptions differently and more realistically. As the server gets overloaded with clients, the time to run queries increases, eventually becoming impractical, with some database management systems triggering an exception in server code when the query exceeds a fixed time. This is the scenario which we model in our simulation: if a query reaches a certain time limit (the query *max_time*), an exception is raised in the server code. In addition, the query time should depend on the number of active clients.

If we include PreX in this scenario, it now becomes possible for alarms to be triggered, signifying the prediction of a potential exception. When this occurs, a preventive action can be taken. Note that due to existence of false negatives, it is possible that this action is taken even if the query time is far from the *max_time*.

The preventive action is written by the application developer, and might involve complex procedures. However, we intended to assess the impact of a simple action first, so that complicated preventive actions were not unnecessarily coded. To this end, we modeled our clients to sleep for a random amount of time, uniformly distributed in a fixed interval. By sleeping, the clients essentially delay their execution and re-schedule it to a later time, according to a uniform distribution. The rationale behind this is that heavy/lengthy queries are scattered across a random uniform interval, effectively reducing the load on the database and throttling the rate of requests. Consequently, faster queries are prioritized, allowing these clients to disconnect and freeing more CPU room for the execution of the heavy queries, possibly reducing the number of exceptions and increasing the throughput of successful operations. This idea is revisited in the experiments detailed in Chapter 5.

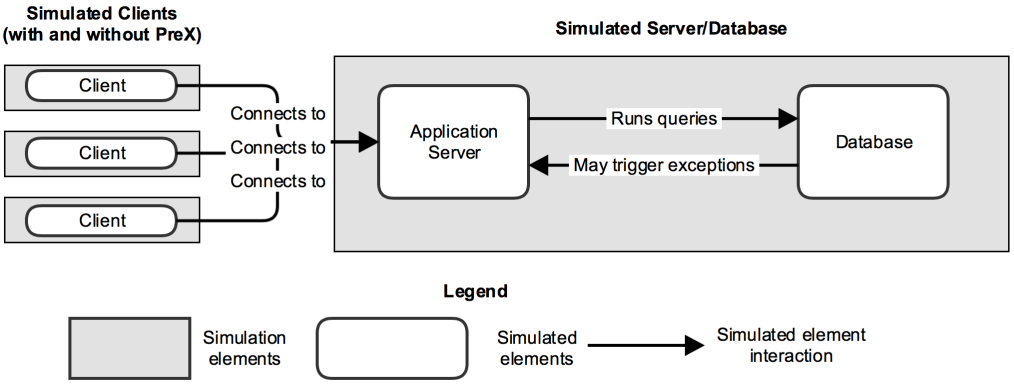

FIGURE 3.9: Simulation scenario. Each client is an independent agent.

3.7.2 Simulation Parameters

Figures 3.10 and 3.11 show that there are a number of steps in the simulation of each client that depend on simulation parameters. Some of these parameters, such as the time to run each query, depend on the application domain, and should be selected to closely resemble real-world behaviour. Other parameters, such as the time to sleep in a recovery action, or the model accuracy, are configurable parts of our model. It is, then, useful to observe how the simulation outcome changes depending on these latter parameters.

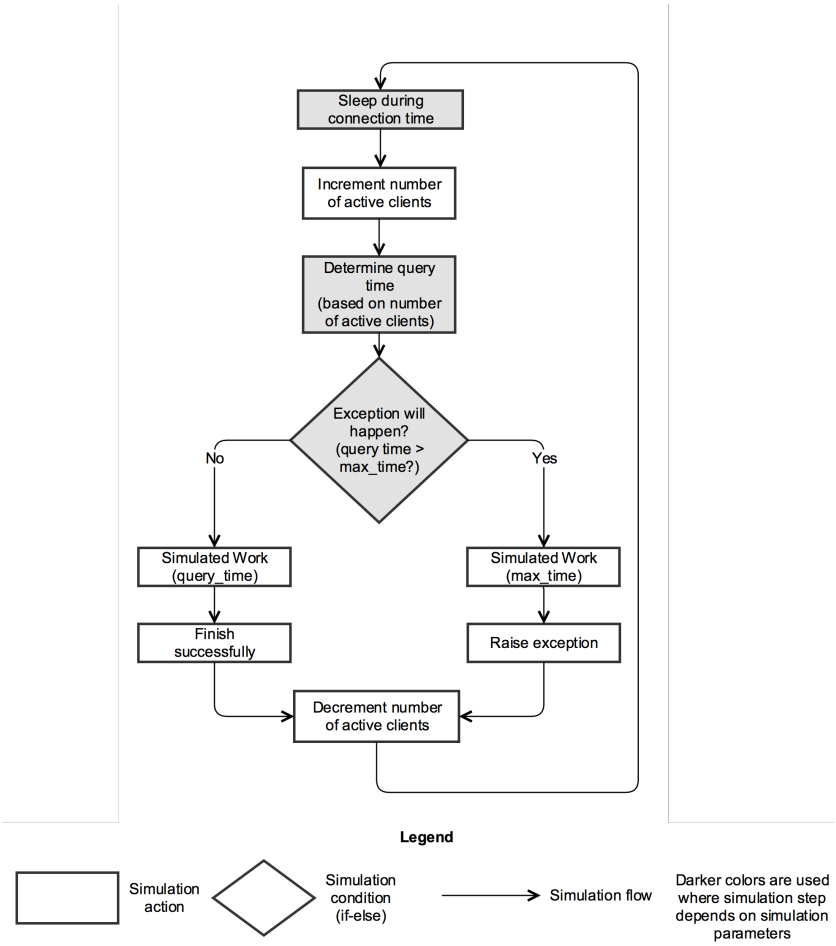

FIGURE 3.10: Flowchart of the behavior of each individual simulated client when PreX is not used.

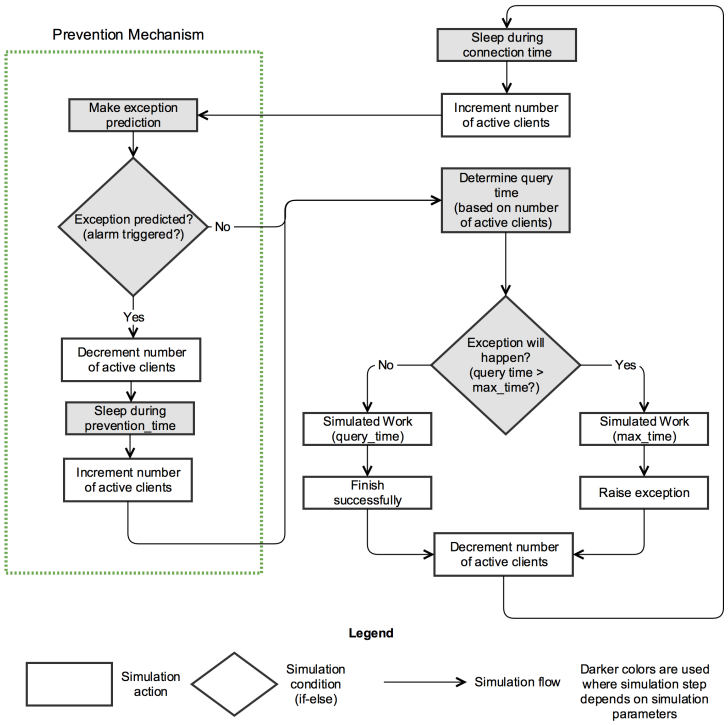

FIGURE 3.11: Flowchart of the behavior of each individual simulated client when PreX is used.

The simulation parameters are, then:

- **Number of clients**. They represent the number of active clients connecting and overloading the server. This is similar to the number of virtual clients used by the TPC-W load generator (seen in Appendix B). It is mostly through this parameter that different workloads can be simulated.

- **Connection time**. This parameter represents the time that it takes for a client to start a connection to the server, before it runs any queries. In practice, lower values mean that more clients can connect in a smaller interval, leading to higher loads on the server. We determined this parameter from the experimental data gathered in the preliminary experiments seen in Appendix B.

- **Query time**. The query time, that is, the time to run a query, depends on the number of currently connected clients. More clients should lead to a longer query time. This parameter is essential to the simulation, and must closely resemble realistic environments. We adjusted our experimental scenario to measure the query time according to the number of clients. These experiments revealed a linear relationship between the query time and the number of clients. Other articles where

TPC-W is used have shown that this linearity exists [75–77]. As such, we modeled the query time according to a linear expression of the form:

$$query\_time = base\_time + k \times number\_of\_clients + \mathcal{N}(m, \sigma)$$

where the *base_time* and k are deployment-specific attributes determined empirically and where m and σ are the mean and standard deviation of a normal distribution used to add some noise and randomness to the *query time*. m and σ were chosen to add very little noise, since the query times did not exhibit a high standard deviation.

- **Maximum query time (max_time)**. This value represents the maximum time that a query can take before the database raises an exception in the server code. Lower values lead to more exceptions. This parameter was configured to have "sensible" values for an operation in an e-commerce web-site.

- **Exception prediction accuracy**. This parameter models the accuracy of the PreX model. In our simulation, it is represented by a tuple (FNR, FPR), representing the false negative rate and the false positive rate, respectively. This way, depending on whether the query time exceeds the max_time (i.e. an exception will happen), we can accurately simulate the outcome of PreX's predictions. It is interesting to experiment with different values and determine their effect on the simulation metrics.

- **Prevention sleep time (prevention_time)**. This parameter regards the prevention mechanism presented in the previous section. When a client receives an alarm of a potential exception, it sleeps for a given *prevention_time* to try to prevent the exception and reduce the load on the server. The prevention_time is, thus another interesting parameter to test for different values. In this work we defined it as a value sampled form a uniform distribution between 33.3% and 100% of the max_time (the rationale is that we should wait some time, at most max_time until we try again, but preferably less).

- **Simulation time**. The simulation time should not have a significant effect in the simulation, as long as it is sufficient for simulation stabilization. Indeed, it was selected so as to allow for this stabilization.

3.7.3 Implementation and metrics

The simulation was implemented in Python 3 using the SimPy[5] library. The clients were implemented as a simulation object which yielded execution whenever a simulation step should wait for a given time (i.e. a "timeout"). The server was simulated using a shared resource where the number of active clients was kept, thus influencing the query time.

SimPy is a unitless simulation framework. However, considering that each simulation step/unit is a millisecond, we can simulate with millisecond precision. To reach this goal, all the empirical parameters (e.g. the query base time) were measured with millisecond precision. As noted in the previous section, the simulation was run until after it had stabilized and long enough to gather enough data, totalling 5 simulated minutes.

To analyze the simulation and evaluate the proposed model, several performance metrics were gathered. These allow analysis of the number of exceptions, number of successful operations and successful operation throughput.

3.7.4 Limitations

Although the use of simulation software offers development advantages, as well as a fast way to study different model parameters, it is not without its limitations. The simulation is only as good as its underlying model. In our model of the TPC-W scenario (see Appendix B), we put aside the ramp-up period and scheduled all clients to start executing concurrently. This limitation might impact the generality of our results. Similarly, we have assumed that there is independence of connection time and query time. This makes sense when a load balancer is used, forcing the bottleneck to be at the database layer, but it was not necessarily the case with the TPC-W setup.

This simulation does not distinguish between query types. This means that read queries are treated exactly the same as write queries. There are scenarios where this is the case, but the TPC-W workload used in the preliminary experiments contained a mix of both queries. It may nevertheless still be possible that the linear relationship between the number of clients and the query time holds. However, more advanced prevention actions, such as prioritizing write queries (usually shorter and more business-critical in e-commerce websites) over read queries, are not allowed with this simulation model.

Finally, this simulation does not deal with dynamic environments, where the performance accuracy might change at run-time. However, the results allow us to understand what would happen to the system if the performance accuracy suddenly changed. In this

[5] `https://simpy.readthedocs.io`

section we present the results for the preliminary experiments, validating the model's predictive capabilities, as well as the results for the simulation, validating the model's preventive usefulness.

3.7.5 Results

To analyse the effectiveness of the proposed prevention action and, hence, validate the usefulness of the proposed model, several measurements were made.

Figure 3.12 shows the operation throughput as a function of the number of clients (n), for different scenarios. The baseline scenario corresponds to normal operation, when PreX is not used. The remaining scenarios show the use of PreX and the prevention action with varying degrees of prediction accuracy. The accuracy of the preliminary results is used, in its 10-fold cross-validation variant (FNR=0.0152, FPR=0.0889) and its "test on one, validate on another" variant (FNR=0.233, FPR=0.0701). Additionally, there are three scenarios with varying accuracy which reveal how the system behaves: FPR=FNR=0.5; FPR=0.1,FNR=0.9 and FPR=0.9,FNR=0.1. In total, the 7 different scenarios allow an evaluation of PreX. Figure 3.13 shows the total number of recorded exceptions for the same scenarios.

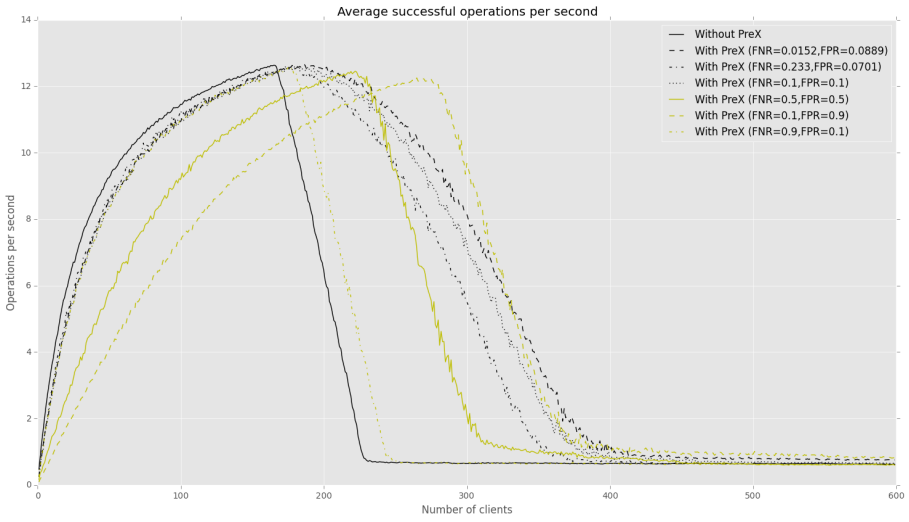

FIGURE 3.12: Successful operation throughput with and without PreX, based on number of clients, for different prediction accuracy models.

It is clear from Figures 3.12 and 3.13 and that PreX offers increased throughput when the server is overloaded. Due to false positives, the prevention mechanism leads to a lower throughput in the absence of exceptions ($n < 180$). However, when exceptions start happening ($n = 180$), the throughput abruptly falls in the absence of PreX. In

contrast, PreX delays the appearance of exceptions and offers increased throughput, offering advantages to application developers which intend to increase the reliability of their systems.

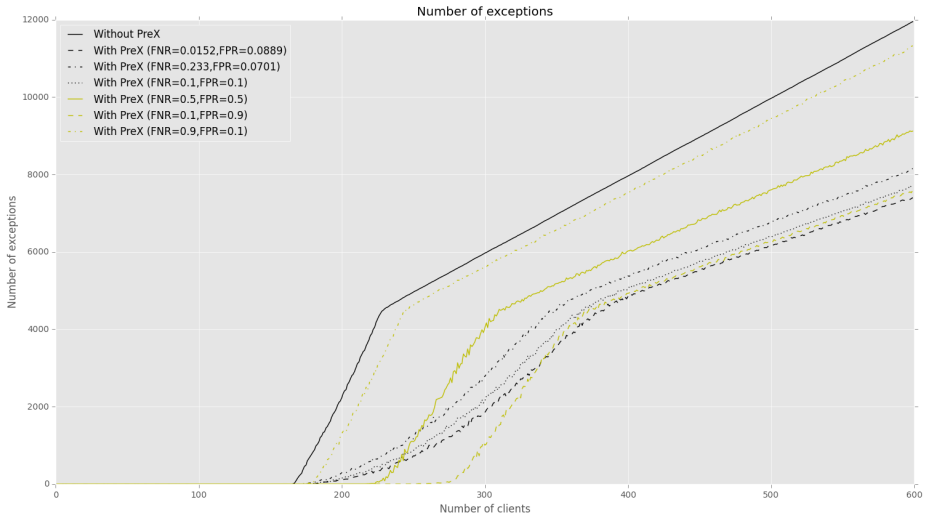

FIGURE 3.13: Total number of exceptions with and without PreX, based on number of clients, for different prediction accuracy models.

An increase in the false positive rate naturally leads to a decrease in operation throughput, delaying the appearance of exceptions. When this value is very high, the throughput can decrease by 50%, but exceptions are also delayed. This implies that the preventive action is also useful even when an exception is not bound to happen. However, it is clear that it is most useful when exceptions are accurately predicted (e.g. FNR=FPR=0.1). The area under curve seems larger when there is an adequate balance of FNR and FPR (higher prediction accuracy). This is the case with the empirical data used (FNR=0.233, FPR=0.0701). Thus, the simulation shows that the TPC-W scenario, whose prediction model we already know, is an ideal use-case for PreX and the proposed prevention action.

Figures 3.14 and 3.15 show the throughput and number of exceptions when using PreX with varying degrees of false positive rate. The false negative rate is fixed at the empirically determined value for TPC-W (0.233). Three different scenarios (with and without PreX) are represented: for 128, 192 and 256 clients. If we look at Figures 3.12 and 3.13 we see that these three workloads have clearly different characteristics. The first represents the absence of exceptions. The last represents a full server overload. The second one is a balance between both, in the transition phase where the server can still accommodate some clients, but already catches exceptions. It makes sense, then, to look at these three different scenarios.

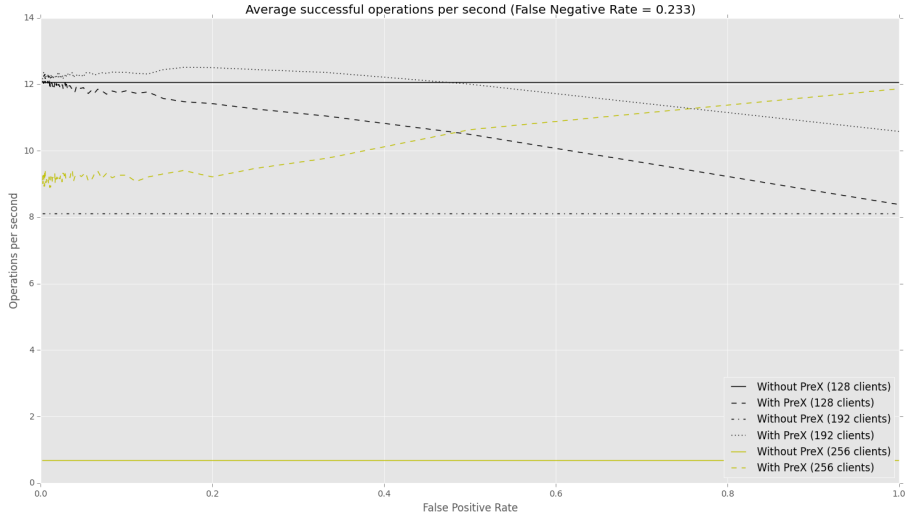

FIGURE 3.14: Successful operation throughput with and without PreX, based on the false positive rate, for different workloads.

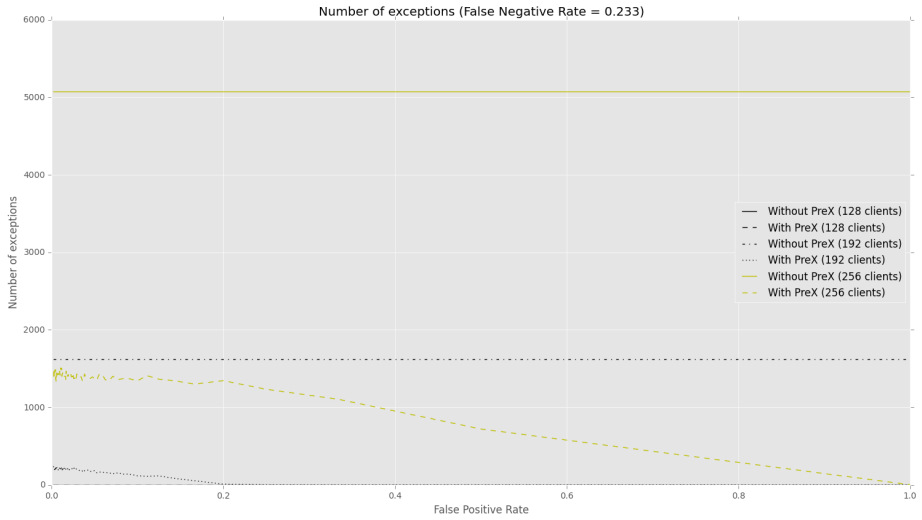

FIGURE 3.15: Total number of exceptions with and without PreX, based on the false positive rate, for different workloads.

As we have seen before, a higher false positive rate has a negative effect in the operation throughput when exceptions are rare. However, in the scenario where the server is overloaded (256 clients), higher values actually have a positive effect, once again strengthening the idea that the preventive action is useful even before the exception is predicted. In general, there are less exceptions when the preventive action is used more often, but this often means that the overall throughput decreases (since operations are being adjourned for a later time).

Figures 3.14 and 3.15 show similar plots, regarding the throughput and number of exceptions when using PreX with varying degrees of false negative rate.

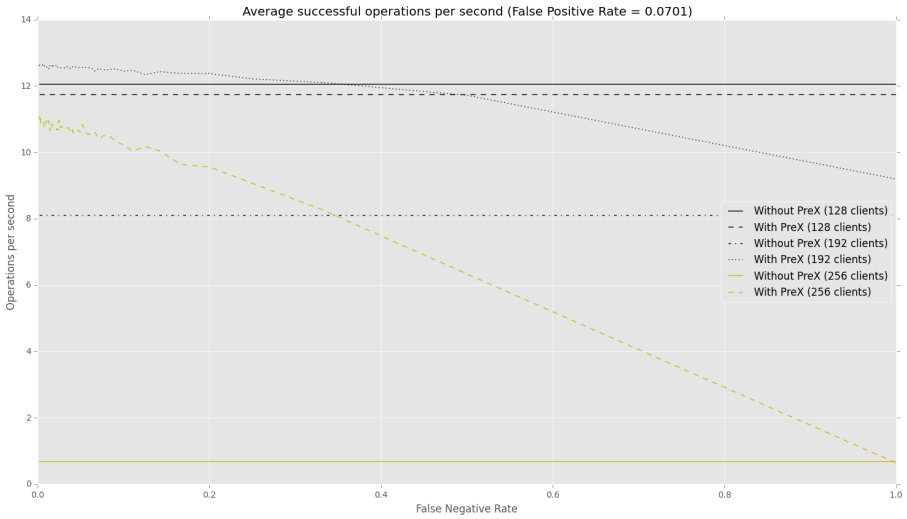

FIGURE 3.16: Successful operation throughput with and without PreX, based on the false negative rate, for different workloads.

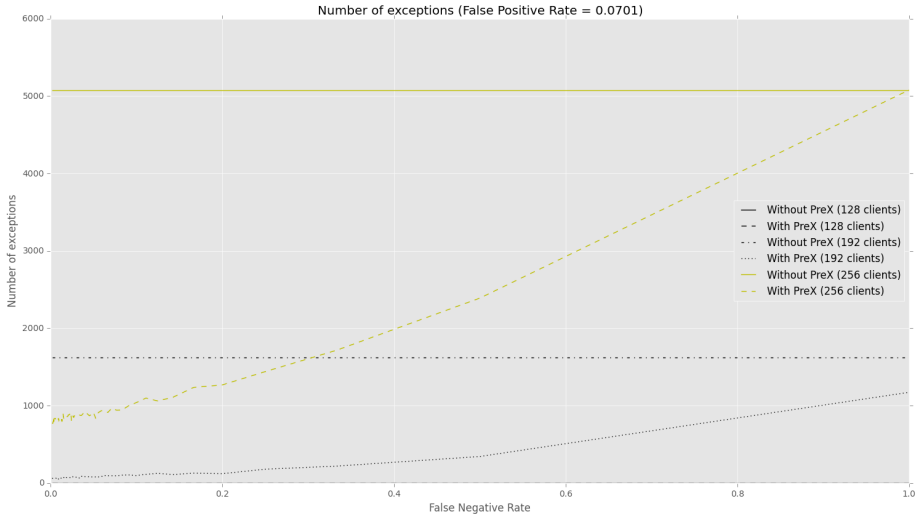

FIGURE 3.17: Total number of exceptions with and without PreX, based on the false negative rate, for different workloads.

The analysis of the model's behaviour with regards to false negative rate is more straightforward. An increase in false negative rate leads to more exceptions, because the prevention mechanism is less used. Nevertheless, it is interesting to note that even when there are many false negatives (i.e. exceptions are often not predicted), the proposed model offers advantages. For example, a false negative rate of 0.8 still represents a three-fold

increase in operation throughput for the heaviest workload (256 clients). This shows that PreX offers new reliability options which are useful to developers.

Lastly, Figures 3.18, 3.19 and 3.20 show the throughput during each second of three simulated scenarios. Each of these represents a different workload, once again using 128, 192 and 256 clients. The prediction accuracy used the one determined empirically in the previous section (FNR=0.233, FPR=0.0701), since these are the most realistic values currently available.

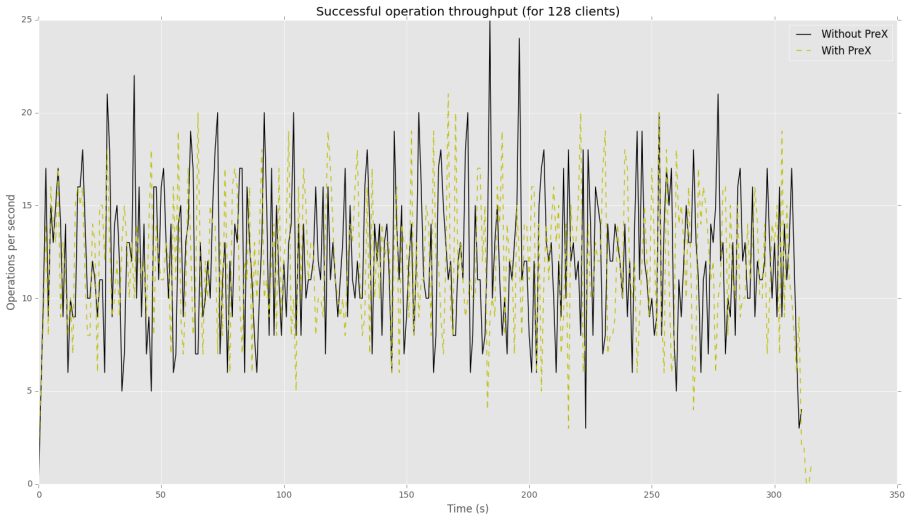

FIGURE 3.18: Successful operation throughput in one simulation with and without PreX, for a workload with no exceptions, using the empirically determined false negative and false positive rates.

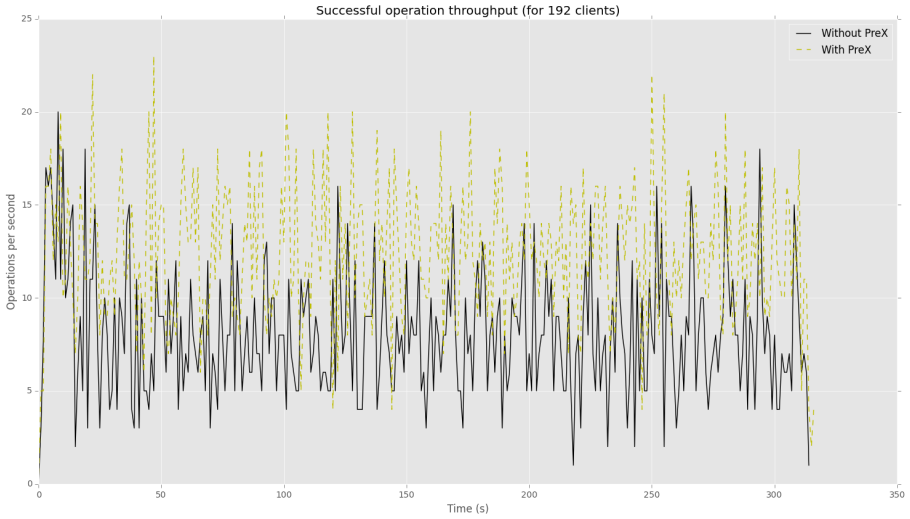

FIGURE 3.19: Successful operation throughput in one simulation with and without PreX, for a workload with some exceptions, using the empirically determined false negative and false positive rates.

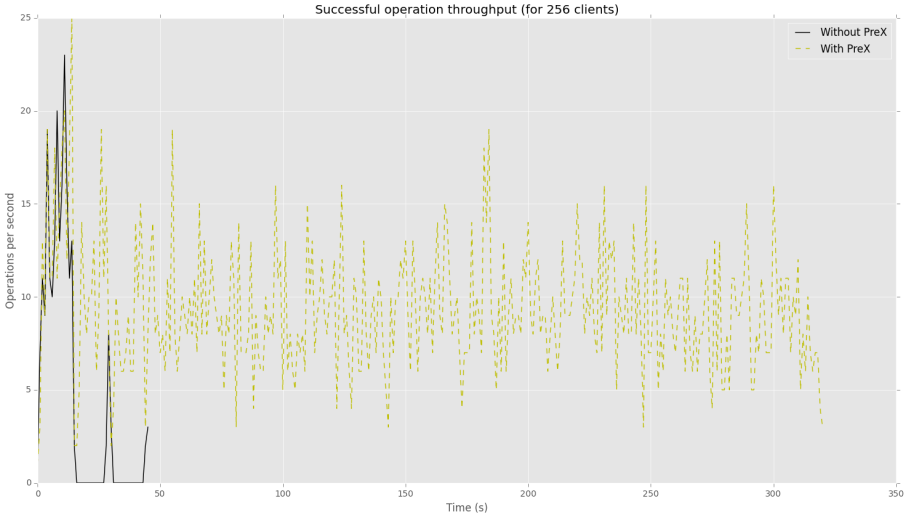

FIGURE 3.20: Successful operation throughput in one simulation with and without PreX, for a workload with many exceptions, using the empirically determined false negative and false positive rates.

It can be concluded, from Figures 3.18, 3.19 and 3.20, that, whenever there are exceptions, PreX offers an increase in operation throughput during the simulation. For example, in the case of 192 clients (Figure 3.19), this increase is marginal, but guarantees that, in the end, more operations will have completed successfully. In the case of 256 clients (Figure 3.20), the original simulation flatlines its throughput due to the constant overload of clients. By using PreX, it is possible to continue operation, even if some exceptions are triggered. Lastly, in the workload with no exceptions (Figure 3.18), we see the effect of false positives marginally decreasing the operation throughput when PreX is used. The increase in throughput is a direct consequence of the uniform redistribution of load which reduces the amount of concurrent clients and, ultimately, increases individual query performance (if there are less simultaneous clients, the overall query performance increases). The effectiveness of PreX's preventive capabilities is evaluated in a real-world scenario in chapter 5.

Chapter 4

Implementation

The proposed model was implemented as a prototype. This prototype consists of four different applications:

- **Coordinator**: The coordinator entity as presented in the model. Developed in Java 8 as a standalone application.
- **Client Library**: The client library designed to be used by applications wishing to make use of the new model. It can be used to feed data to the system (acting as a probe) or to introduce the new *try-prevent-catch* blocks. It is a Java 8 library that can be easily plugged into existing Java 8 projects with a small number of changes to original application code.
- **Probe**: A data gathering entity (probe) as presented in the model. Developed in Java 8 as a standalone application, adapted from previous work and modified to communicate with the coordinator.
- **Administration Application**: An administration application used to control and communicate with the coordinator for tasks such as scheduling training of new models or adjusting prediction parameters. This was developed in Java 8, supporting both a CLI and GUI interface.

The implementation, thus, uses a Java 8 library. This choice allows easy integration in projects, without the need for additional parsers or JVM modifications. A library is simple to integrate in current projects, even those which have not yet been ported to Java 8. In section 4.2.3 we present the advantages and disadvantages of this approach.

In this chapter, we present example usage of the implemented model, the architecture and main design choices when developing each of these applications, as well as an overview of the protocol used between them.

4.1 Introductory Concepts

To better understand the implementation details of PreX, in this section, we present a set of core concepts used throughout this chapter.

The coordinator and client libraries exchange **protocol messages**, which are serializable Java classes exchanged through the *ObjectOutput/InputStream* mechanisms. Regular clients[1] can: (i) ask the coordinator to start/stop making predictions or (ii) supply data to the prediction system. Data is supplied in the form of **samples**. A sample represents a unique data point and is characterized by a name, a source, a timestamp and a value. Samples with the same source come from the same client (e.g. from the same probe), and samples with the same timestamp happened at the same time.

When the coordinator builds the set of features used to train prediction of a given exception in a given prediction context, it does not consider that samples with the same name belong to the same feature. Indeed, the **unique identifier of a sample** is a pair <sample name, sample source>. This allows samples of the same kind from different machines (e.g. the CPU usage from machine A and the CPU usage from machine B).

The training process involves the algorithm described in section 3.5. This algorithm requires a starting time and stopping time to build the different time-windows. As such, the coordinator must be able to identify when an experiment "started" and when an experiment "ended". In between the start and end of each experiment, the algorithm can be applied. Within the PreX implementation, these are known as **runs**: each run is characterized by a start and finishing timestamp. The system can continue to work when a run is not currently active, but that data is not used for training (it might be used for prediction). It is the responsibility of the coordinator to maintain a database of data (samples) and **prediction models**, which are characterized by the (T, k, t) parameters described in Section 3.5, a Weka binary model, and the model's performance metrics for the most recent training data. The data stored in the database must then be summarized with the window-merge algorithm into what is known as a **summarized dataset**: a dataset, for a specific combination of (T, k, t) parameters, that has been through the time-window construction and window-merge algorithms.

An architectural overview of the implementation is seen in Figure 4.1. There are two main components: the coordinator (server) and the client library used to build probes and the *try-prevent-catch*. This architecture is very similar to the proposed architecture for model implementations previously presented in section 3.3.

[1] Regular clients are applications using the model and probes. Other clients, such as the administration application, can perform other tasks besides those of regular clients.

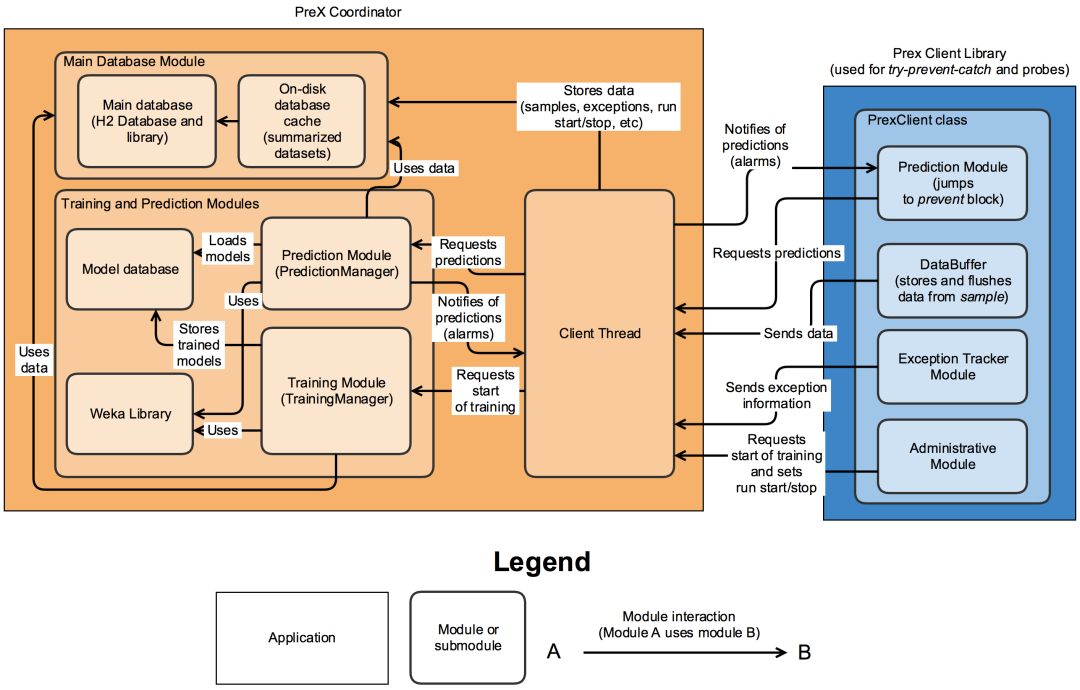

FIGURE 4.1: Overview of the implementation architecture

4.2 Client library

The client library attempts to isolate most of the communication mechanisms between the coordinator and any clients. It makes uses of Java 8 functionality, such as functional interfaces, and encourages the use of lambda functions to emulate the addition of new syntactic elements to the Java language. It can also be used to only supply data to the coordinator, effectively acting as a probe. In the following sections, we show how this library can be used in existing applications to implement the PreX model, how it can be used to implement Probes, and compare it with the model itself.

4.2.1 Try-Prevent-Catch

If an application wishes to use the *try-prevent-catch* construct in *synchronous* mode, it first needs to create an instance of the ***PrexClient*** class. This class accepts four parameters:

- A unique identifier for this client
- The coordinator hostname
- The coordinator port

- A sample buffer size (defaults to 1)

The unique identifier is used by probes or when the *sample* keyword is used. It allows each sample of a feature to be uniquely identified and tied to an entity (source). The sample buffer size determines how many samples can be buffered before they are flushed to the coordinator, allowing tuning of network performance. Since most applications will use the *PrexClient* class with *try-prevent-catch* blocks, the default value of 1 means that *sample* keyword data is flushed instantly.

Once an instance of the *PrexClient* class has been obtained, a chaining design pattern can be used to emulate *try-prevent-catch* blocks as in the example in Listing 4.1.

```
1  PrexClient client = new PrexClient(<id>, <host>, <port>);
2
3  client
4  .Try( <prediction context>, (t) -> {
5      // ... code which must call t.check();
6  })
7  .Prevent ( <exception type>.class, (predictionInfoObject) -> {
8      // ... code
9  })
10 .Catch ( <exception type>.class, (exception) -> {
11     // ... code
12 }).sync();
```

Listing 4.1: Example syntax of the *try-prevent-catch* implementation

Evidently, the graphical disposition of code is a mere suggestion, since these are only methods being called through the chaining design pattern. No code is actually run until the *sync* method is called, indicating the *synchronous* version of the model, as opposed to the asynchronous one.

Within the *try* block, one must manually invoke *t.check()* whenever the programmer wishes the code to potentially jump to the prevent block. This is a consequence of implementing the approach as a Java library without additional changes to the bytecode of the function, JVM changes or a pre-processor. If the programmer does not invoke t.check(), then the prevent block can never be accessed (although the *catch* block can).

The prevent function must accept an instance of a PredictionInformationObject. This object corresponds to the model's prediction information object and, in the current implementation of PreX, offers access to the lead-time. Thus, within the prevent block, it is possible to act differently based on how long ago the prediction was made, or how close the exception is to happen, according to the alarm.

Note also that PreX supports the exception hierarchical tree. If the user specifies that it wants to catch or prevent exceptions of type *Java.lang.Exception*, then any subclass of this class is also considered by the system. This is achieved through reflective code.

Thus, in practice, very little changes are required when replacing *try-catch* blocks with this implementation. The main required changes are:

- An instance of the *PrexClient* class should be obtained and used within the function with the *try-prevent-catch* block. This instance can be re-used.
- Exceptions have to be appended with the *.class* suffix.
- The *check* method has to be invoked within the *try* block whenever it is useful to execute to check for exception alarms. The *sample* method can also be invoked.
- A prediction context must be supplied and a prevention action must be written.
- Since code is effectively usually contained in lambda functions , it might be necessary to convert non-final variables to final variables.
- It is also necessary to store any return values in variables outside of the lambda functions, adding an extra return statement after the *sync* method call.

An academic example of the implementation can be seen in listing 4.2:

```
1  PrexClient c = new PrexClient("client1", "localhost", 1610);
2
3  c
4  .Try( "test", (t) -> {
5      Connection conn = connectionPoool.getConnection();
6
7      while ( data = getDataToWrite() ) {
8          t.check(); // Might jump into prevent block!
9          data.writeToDB(conn);
10     }
11 })
12 .Prevent ( SQLException.class, (predictionInfoObject) -> {
13     try { Thread.sleep(1000); } catch (Exception e) {}
14 })
15 .Catch ( SQLException.class, (exception) -> {
16     System.err.println("There was an error writing the data!");
17 }).sync();
18 conn.close();
```

LISTING 4.2: Academic example of the *try-prevent-catch* implementation

In this example, the client identifies itself as "*client1*" and connects to a coordinator running at *localhost:1610*. It then continuously fetches data to write to a database. Since the *writeToDB* method might raise a *SQLException* exception, the programmer introduces a call to *check* before writing to the database, so that the prevention mechanism can be activated if any alarm is triggered. The prevention action involves a temporary sleep to reduce overload on the database.

The previous code would look very similar if using Java's native *try-catch*, showing that this approach adds little extra verbosity:

```
1  try {
2      Connection conn = connectionPoool.getConnection();
3
4      while ( data = getDataToWrite() ) {
5          data.writeToDB(conn);
6      }
7  } catch (SQLException e) {
8      System.err.println("There was an error writing the data!");
9  }
10 conn.close();
```

Listing 4.3: Equivalent native Java *try-catch* implementation

An *asynchronous* version of the *try-prevent-catch* construct can be used by calling *async* instead of *sync* at the end of the chain. Note that in this scenario, the calls to *check()* have no effect.

4.2.2 Probes

The *PrexClient* class is also the base class used by any probes. To build a probe, one must first instantiate this class, making sure to provide a unique source which does not change between executions, since it will be attached to any samples provided by this probe. If a probe is designed to feed many different samples and features in a short amount of time, it should not use the default sample buffer size of 1, instead changing it to a more suitable value which does not overflow the network with samples.

Once an instance has been made available, the probe can send data to the coordinator at any time with the *sample* method. An example of code using this functionality of the client library, adapted from one of the project's probes[2], is seen in Listing 4.4.

[2]Available at `https://github.com/Jorl17/prex-probe`.

```
1  class Probe {
2      private PrexClient client;
3
4      public Probe(String src, String host, int port) {
5          client = new PrexClient(src, host, port, 100/* buffer size */);
6      }
7      // ...
8      public void log(DataPoint point) {
9          // Iterate features and supply them to the prediction system
10         for (Feature f : point.getFeatures())
11             client.sample(f.getFeatureName(), f.asFloat());
12     }
13     //...
14 }
```

Listing 4.4: Example Probe code (adapted from PreX's probes)

The code in Listing 4.4 shows the instantiation of a *PrexClient* in line 5 with a sample buffer of 100 samples. In lines 10-11, several features are fed to the sample buffer, which will be flushed whenever 100 samples have been filled. This example shows that it is easy to build probes and integrate them with a PreX coordinator, since no additional work is required[3].

4.2.3 Comparison with model

The model present in previous chapters of this thesis is not fully implemented with the proposed approach, although the provided implementation is functionally equivalent.

This implementation uses native Java features, making extensive use of higher-order functions (functions which accept other functions as arguments), generics and reflective code (for run-time exception type inheritance checking). These allow code to look very similar to traditional exception handling without extensive project rewrite. Indeed, our experiments, presented in section 5, involved adapting the Shopizer code and porting it from Java 7 to Java 8, a task which was not difficult, in spite of this being a complex web application.

The main alternatives to the proposed approach were:

- Implementing JVM and compiler support for the new model

[3]In future work, security considerations might make this process lengthier, by adding authentication mechanisms.

- Implementing a compilation pre-processor that integrated current code with the existing library
- Using instrumentation techniques to remove the need for calling *check*

Modifying the JVM would force projects to use this modified compiler and runtime, making the implementation harder to use. Similarly, a pre-processor would demand extra effort on behalf of the developer. Instrumentation techniques would allow for automatic insertion of *check* calls, but would hinder performance. In addition, since the *no-alarm* keyword could not be implemented that way[4], automatically inserting these calls would remove the ability to temporarily stop alarms.

The main disadvantage of the chosen implementation is that it does not allow for the full model to be implemented. In particular, explicit calls to *check* are needed and the *no_ alarm* keyword is not present (although its functionality is, in the form of explicit *check* calls). The programmer is also forced to manually create and monitor the *Prex-Client* class, which would not be needed with modifications to the compiler and runtime. Lastly, as seen in section 5.4.3, there is some small overhead that this implementation adds and that could possibly be removed by using a modified runtime and compiler.

4.3 Protocol

The communication between Coordinator and clients follows a protocol using the Java *ObjectOutputStream* and *ObjectInputStream* streams. There is a pre-defined abstract *Message* class which all protocol messages should extend. This *Message* class only has to carry the source (unique identifier) of the message.

In total, the following messages are implemented in the protocol to exchange information between all entities:

- **BufferedSamplesMessage**: Supply a set of samples contained in a buffer to the coordinator.
- **RecordedExceptionMessage**: Notify the coordinator of an exception that has happened, so that it can be stored for future model training.
- **StartListeningToPredictionsMessage**: Notify the coordinator that a given client wishes to listen to predictions for a given exception under a given prediction

[4]It would be possible to wrap the *no-alarm* code in another lambda function, but this would make code harder to develop due to scopes issues (e.g. forcing variables to be made final).

context. The coordinator should reply with the current status of the prediction (*true* or *false*).

- **StopListeningToPredictionsMessage**: Notify the coordinator that a given client wishes to stop listening to predictions for a given exception under a given prediction context.
- **ExceptionPredictionStateMessage**: Notify the client that a given prediction status has changed (either from *true* to *false* or the reverse). It is up to the client library to keep track of the current state by listening to these messages.
- **TrainMessage**: Trigger the training of predictions for a given exception under a given prediction context. The coordinator should train several models, compare them with the current best (if it exists) on current data, and elect a new winner. Used by the administrator application.
- **AddPredictionContextSampleIDsMessage**: Notify the coordinator that it should add sample IDs (<name,src> sample pairs) to a given prediction context. Used by the administrator application.
- **RemovePredictionContextSampleIDsMessage**: Notify the coordinator that it should remove sample IDs (<name,src> sample pairs) from a given prediction context. Used by the administrator application.

In practice, the development of new probes and sources of data does not require understanding the internal protocol of the implementation, since it is abstracted through higher level functions provided to application developers.

4.4 Coordinator

The coordinator is implemented as a standalone server with an integrated H2 [78] database. It listens to clients which can instruct it to perform specific actions (such as starting the training process), provide data (probes) or request information regarding exceptions. Each client is handled by a separate independent thread. By default, the coordinator listens on TCP port 1610.

4.4.1 Database

The coordinator must be able to store all the samples provided by clients at any time. These samples are then meant to be used both for training, which can be performed offline, or in real-time, during online predictions. For this purpose, a relational database (the H2 Embedded database used in MySQL compatibility mode) was chosen. H2 offers the benefit of being tightly integrated with the Java language and being easy to use as a self-contained embedded database. By relying on the database, issues regarding concurrency, data consistency and storage are made easier. However, not all of the Coordinator's structures are appropriate for storage in a database. For example, trained models are mere binary data, compatible with the Weka Explorer tool, and for this reason do not need to be stored in the database. Thus, the database used in the Coordinator is used only for Samples, Exceptions, Prediction Contexts and Runs. An entity-relationship diagram of the coordinator database can be seen in Figure 4.2.

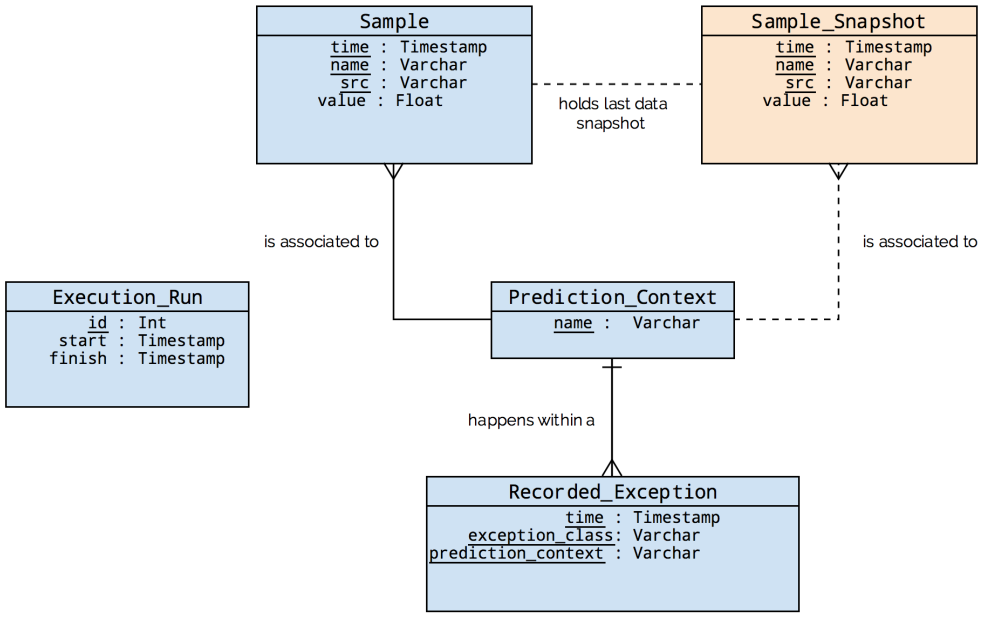

FIGURE 4.2: Entity-Relationship diagram of the coordinator database. The *Sample_Snapshot* table acts as a snapshot of the most recent data in the *Sample* table for performance reasons.

In total, there are 4 different entities: *Sample*, *Prediction_Context*, *Recorded_Exception* and *Execution_Run*. A prediction context is associated to several samples, and several samples can be used by different prediction contexts. Each recorded exception must have been recorded and reported within a *try* block and, hence, is bound to one single prediction context. The *Execution_Run* table stores the start and end of each different run. Its IDs are incrementing and auto-generated, so that at any given time a model can claim that it is trained with data from the "n-th first runs". It should be noted that

there are two distinct tables for samples: the *Sample* and *Sample_Snapshot* tables. The first table contains all the recorded sample data. However, since samples must also be accessed in real-time, an additional table (*Sample_Snapshot*) was created to store only the most recent samples, usually pre-configured to store the samples of the last minute. This makes the real-time access to recent data much faster, particularly in situations where there are thousands of samples arriving at the coordinator every second.

4.4.2 Training

The training process can be triggered with a specific message, usually sent using the administration client. When the coordinator is asked to train the prediction of an exception under a given prediction context, it begins a CPU-intensive task meant to train several models.

The several models use the algorithm presented in section 3.5 and, therefore, might require different values for T, k and t. As a consequence, their datasets (instances used for training) are different, although they are made from the same data. Each dataset is made by applying the Time-Window Construction and Window-Merge algorithms to each run stored in the server. Since the *Sample* table grows into very large values, cached versions of these datasets are stored as binary files. This allows models to bypass heavy queries to the database, accelerating the training process. In section 3.5 we noted that the process of *window-merging* required a set of summarized features to be extracted for each of the original features in the data. In this implementation of PreX, these are: the number of samples, the mean, the maximum, the minimum, the standard deviation and the derivative. These metrics offer a range of information regarding the diversity of samples, as well as time-based information (number of samples and derivative).

In section 3.5.4, it was noted that PreX implementations should consider missing value imputation techniques. In its current implementation, PreX does not perform missing value imputation, representing missing values with NaN or a a constant value (typically -1). This decision stems from the fact that most classification methods already offer their own missing value imputation solutions, which are often best suited for their methods. In addition, missing value imputation techniques would affect the performance of the prediction system, which must act in real-time to make quick predictions within a tight window. It is more efficient to let each of the classifiers deal with the issue of missing values. Furthermore, preliminary experiments with offline data showed that even when each of the missing value imputations presented in section 3.5.4 was used, results did not change (in particular due to the fact that there is little missing data in our experiments).

Section 3.5.4 also noted that PreX implementations might need to perform feature selection in order to improve classifier performance and accuracy. In the experiments with the full PreX model, detailed in chapter 5, several of the feature selection methods offered by Weka were tested[5], but these did not significantly change the accuracy of the provided models. This is a consequence of the fact that our models were already highly accurate and our features did not have a high overlap. Implementing feature selection processes would add extra development time for little gain in the scenarios which we explored. More importantly, since each prediction context can select which samples it "cares about", the task of feature selection loses part of its value, since the system administrator should select features which are relevant. Nonetheless, in a future version of PreX, a feature selection process might offer advantages if there is a much larger number of variables to consider (something which was not the case in the experiments detailed in chapter 5).

A training request does not need to happen only once and, in dynamic environments, it happens several times during application execution. This means that it is often not necessary to rebuild the full dataset, but only include data from more recent runs. The coordinator checks which runs have already been used to build datasets and only adds new data as needed, making the overall training process more efficient. It should be noted that in this process the dataset is automatically balanced using the supervised instance resampling filter found in Weka, although the system administrator is encouraged to have a balance of "golden-runs"

At the same time that the different (T, k, t) datasets are being created (or read from disk), several threads are spawned training models using the Weka Java library. The coordinator then waits up to a maximum of a configurable amount of time for all the models to finish training. If any models cannot be trained during this time, they are aborted and discarded. Whichever models have successfully finished training are ranked and the new best model is chosen. If training had already happened before, the current best is loaded and evaluated on the new data to assert if it should remain the current model or if it should be replaced.

The (T, k, t) parameter combinations can be supplied when the training request is made, though a set of default values is used. The prediction models are currently hardcoded and offer a variety of classifiers with different parameters. These are:

- The *J48* classifier under its default configuration
- The *RandomTree* classifier with depth limit set to 2, 3, 5 and unlimited.

[5]The CorrelationAttributeEval, PrincipalComponents, OneRAttributeEval and WrapperSubsetEval methods were used.

- The *REPTree* classifier with depth limit set to 2, 3, 5 and unlimited.
- The *MultilayerPerceptron* with default parameters, training time = 100, and three distinct network configurations: one with as many neurons as features in a hidden layer, one with two hidden layers and as many neurons as features per layer, and one with only 10 neurons per layer, often having higher error but training faster.
- The *RandomForest* classifier under its default configuration.
- The *IBk* classifier with k=5,30,100.

These classifiers offer a broad range of methods which can be trained in realistic time frames. It should be noted that in the original implementation there were other types of classifiers, such as Naive Bayes classifiers, but they often underperformed and took longer to train.

The decision tree algorithms have depth limits defined wherever possible to increase generality and avoid overfitting. The IBk (K-means) classifier uses values of K which are computationally feasible, since in our experiments values for K>100 did not allow for real-time classification.

The models are all validated using 10-fold cross validation to avoid overfitting and ensure a fair comparison. They are then compared according to the following pseudo-code:

1. If both A and B have a high F-measure value[6] (≥ 0.75), then pick the one with the lowest False Negative Rate.
2. Otherwise, pick the one with the highest f-measure value.

The main idea behind this algorithm is that we first prefer to pick models which are accurate (exhibit a high f-measure value) but, if both models can be categorized as "sufficiently accurate" (which we define as having an f-measure greater than or equal to 0.75), then we should prefer the one that predicts exceptions the most, thus having the lowest false negative rate. Other factors could be taken into account when ranking the models. For example, it would make sense to give precedence to the models with higher lead-time or with a smaller value of k (the preliminary experiments show that higher values of k tend to lead to models that strongly overfit to the data). However, since the training parameters (T, k, t) can be adjusted by the user, these were left out of the

[6]The F-measure value is the harmonic mean of recall and precision. A value of 0.75 ensures a "sufficiently" good model can be used, with an appropriate balance of both types of error (for example, achieved by precision and recall of 0.75). However, since the F-measure does not explicitly take into account the true negative rate, and since our model is particularly interested in preventive false negatives, another comparison metric is used after the models are "sufficeintly good".

ranking process – it is therefore up to the user to provide "sensible" parameters. Lastly, an adjusted F-measure with different weights on precision and recall could be considered, but the proposed two-step approach initially filters "bad" models and then focuses on those with the better characteristics, instead of attempting to do these different processes in one step.

4.4.3 Prediction

Predictions are managed within the Coordinator by a *PredictionManager*. The *PredictionManager* keeps track of current clients interested in predictions, as well as current threads making predictions. Whenever a client wants to "start listening to predictions for a given exception and prediction context", the *PredictionManager* first checks if there is already a thread doing these predictions and, if there isn't, creates it.

The prediction threads start by loading the currently best model available. They then remain dormant for a period of $T \times k$, so that enough samples can be gathered for prediction. Once this time has passed, the Time-Window Construction and Window-Merge algorithms are applied to the last $T \times k$ seconds of data and fed to the Weka model, which outputs a prediction. If the prediction changes (it starts at its current value, false at the start), the prediction thread notifies the *PredictionManager*, which in turn notifies each of the interested clients of this change.

In dynamic environments, the best models might change at run-time. If such is the case, the *PredictionManager* is notified and propagates this information to any relevant prediction threads. These detect the change, load the new model and restart the procedure.

Whenever all clients announce that they no longer wants to listen to predictions (most likely because they have all exited the *try* block), the *PredictionManager* waits a predetermined amount of time (currently 5 seconds) before graciously terminating the appropriate prediction thread. This "grace period" greatly improves performance in situations where clients are constantly entering and leaving a *try* block, such as when it is nested within a loop or in recurrent actions (e.g. a web-server satisfying several similar requests). In effect, the same thread is reused and the overhead of constantly creating and terminating threads (or even re-using them from a thread-pool) is eliminated. In addition, if no such "grace period" existed, then it might happen that no predictions would ever be made in a scenario where clients have a life-span smaller than the prediction $T \times k$ (a situation found, for example, in the experiments seen in chapter 5).

4.5 Administration Application

Some actions within the PreX system, such as training, marking runs as started, or adding and removing associations between features and prediction contexts, have to be triggered by an outside source. While the *PrexClient* class offers all of this functionality, the recommended way of performing these actions is by using the administration application.

The administration application offers a GUI seen in Figure 4.3. An administrator can connect to any coordinator and: (i) start and stop a run; (ii) trigger the training of a model; (iii) Perform changes to sample prediction context associations. The GUI can be used with any number of coordinators by simply changing the host and port and reconnecting, thus avoiding the need for different program instances at different times. In a real-world scenario, an administrator would mostly use the GUI to mark an application run as started or stopped.

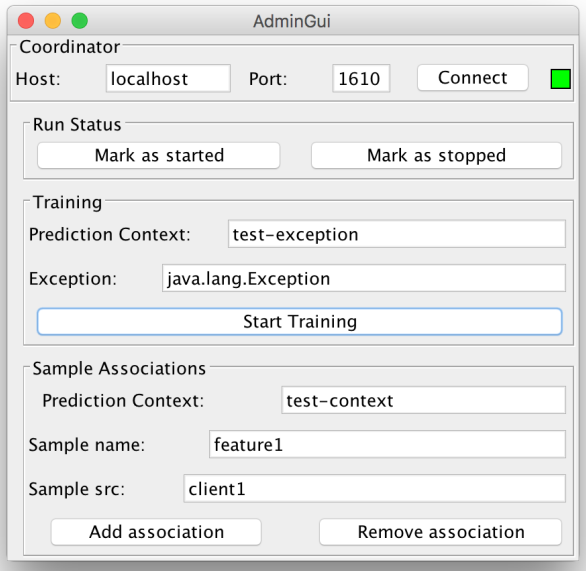

FIGURE 4.3: Administration GUI to interact with the coordinator.

In addition to the GUI, the same application can be controlled through a series of command line options, offering the same functionality. This allows easy automation of all of these tasks. Indeed, to perform the validation experiments shown in chapter 5, the CLI interface was used. In total, there are 9 different command line options (`-start-run`, `-end-run`, `-prediction-context`, `-train`, `-coordinator-host`, `-coordinator-port`, `-sample-src`, `-sample-name`).

Chapter 5

Validation

The preliminary results (Appendix B) demonstrated that it was possible to predict exceptions with high degrees of accuracy. However, these experiments did not shed any light with regards to the preventive capabilities of PreX. The preventive mechanism offered by PreX was shown to be useful using a simulation detailed in Section 3.7, but neither of these experiments validated the model as a whole, in its final implementation detailed in chapter 4. Thus, a set of additional experiments were designed to validate the model and its implementation.

These experiments focused on validating the model simultaneously in its two main components: prediction and prevention. In addition, they were performed with full implementations of this model, as described in chapter 4. PreX was tested on a real enterprise e-commerce solution – the open-source Shopizer application – with the intent of showing that the use of this model can increase system reliability and availability at a fraction of the cost of other approaches. The implementation of PreX used in these experiments can be used with any existing Java project and is available at `https://github.com/Jorl17/prex`.

A modified instance of the open-source e-commerce Java software Shopizer [79] was configured, using a MariaDB database as the backend. The Shopizer e-commerce application is one of the largest open-source e-commerce platforms and is under constant development.

A workload was configured using jMeter [80] to simulate Web users navigating through the virtual shop, eventually overloading the database. As a consequence, since queries were pre-configured to timeout in the database after a period of time, during the execution of the workload, several exceptions are raised. PreX was then used to predict and prevent as many of these exceptions as possible.

The goal of the experiments was to test the following **research hypotheses**:

1. **H1**: *PreX is able to predict exceptions with False Positive Rates and False Negative Rates below 20%.*
2. **H2**: *PreX is able to prevent exceptions in an effective way.*
3. **H3**: *PreX enables the usage of new and differentiated recovery strategies, not available to traditional EH models.*
4. **H4**: *It is possible to define a (T, k, t) parameter configuration that will deliver good performance.*
5. **H5**: *The exception prediction algorithms are sufficiently robust to withstand slight differences in environment conditions without affecting accuracy*
6. **H6**: *PreX does not impact performance by more than 5%.*

To test question 1, an offline training of classifiers is sufficient. These classifiers can then be used at run-time to test question 2. At the same time, traditional recovery strategies should be employed to be compared with the preventive mechanism and test question 3. Finally, hypothesis 4 can be partially tested by exploring different parameters in the offline training. To test hypothesis 5, we can train models in different scenarios, with different workloads, and validate them on one another. Finally, the last hypothesis concerns the impact of PreX on the overall performance of the system. Even when no prediction is being made, but data is being sampled, there might be some overhead added by PreX. By comparing the execution of the original unmodified Shopizer with a modified Shopizer, where PreX is used but has no prediction models trained, this comparison can be made.

In the following sections we present the experimental setup and methodology, as well as the results of these experiments.

5.1 Experimental Setup

The setup consisted of 4 machines, simulating a 3-tiered architecture. The machines all had the same hardware and software, shown in Table 5.1, and were named *prex1*, *prex2*, *prex3* and *prex4*.

TABLE 5.1: Summary of the machine specifications for the validation experiments

CPU + OS	**RAM**	**HDD**	**Name**
Intel Core i3 3.10GHz CentOS 7 (64 bit)	4GB	WD5000AAKX 7200 RPM 16MB Cache SATA 6.0GB/s	*prex1*
Intel Core i3 3.10GHz CentOS 7 (64 bit)	4GB	WD5000AAKX 7200 RPM 16MB Cache SATA 6.0GB/s	*prex2*
Intel Core i3 3.10GHz CentOS 7 (64 bit)	4GB	WD5000AAKX 7200 RPM 16MB Cache SATA 6.0GB/s	*prex3*
Intel Core i3 3.10GHz CentOS 7 (64 bit)	4GB	WD5000AAKX 7200 RPM 16MB Cache SATA 6.0GB/s	*prex4*

An instance of MariaDB 10 was configured in the *prex1* machine, to act as the data source for Shopizer. Shopizer itself was downloaded from the official repository and installed in *prex2*. By default, the application comes with a limited set of mock data (6 products). Since we wanted to create high workloads at the database layer, we modified Shopizer so that each product was inserted, with slightly different versions, 1000 times. Hence, in total, there were 60.000 products.

An instance of jMeter was installed in the prex3 and prex4 machines, so that they could be used to induce heavy loads on the other two machines, simulating the client layer. In addition to this, the PreX coordinator was deployed in *prex3*. The coordinator is responsible for training and prediction, which is bound to consume a significant amount of CPU time. Thus, it was deployed in the machine with the least CPU usage (*prex3*), so as to reduce its overhead on the system.

The PreX probes communicated with the coordinator to feed real-time data used for training and predicting[1]. The modified Shopizer application included the PreX client library and the *try-prevent-catch* mechanism, also communicating with the coordinator as described in section 4.3. Lastly, an administrator application was used from within the *prex3* machine to control the experiments. Figure 5.1 shows the overall experimental setup.

[1]These probes sampled dozens of variables regarding CPU, RAM, Network and Disk access, similarly to the probes used in Appendix B.

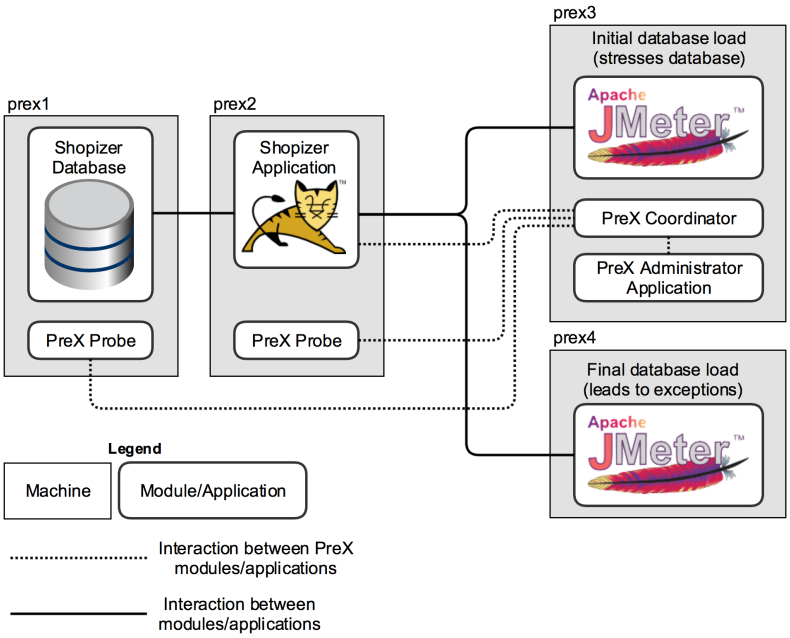

FIGURE 5.1: Experimental Setup for the full implementation experiments.

5.2 Experimental Methodology

The experimental procedure consisted of two stages: training and preventing. The first stage involved running a series of tests where exceptions were triggered, using a simulation of several users navigating in Shopizer (with jMeter). After each test was completed, the machines were reset to its original state and the experiments repeated, so that enough data was available to accurately train several validation models. The trained models (whose training was triggered by the administration application) were then used by PreX in a second stage, repeating the previous steps, but using a method to try to prevent exceptions, detailed further below. Thus, the full model was tested, from prediction to prevention, covering all the different blocks that are part of the *try-prevent-catch* mechanism

The default configurations for Shopizer allowed queries to take a very long time, making tests unpractical and unrealistic. To counter this situation, MariaDB was configured to timeout unfinished queries after 15 seconds. Under heavy workloads, this triggers exceptions in the Shopizer application, which were used for training.

The workload was generated by jMeter instances running in the remaining two machines (*prex3* and *prex4*). When each test started, the prex3 machine introduced a heavy workload simulating 128 clients clicking through the website at a random time after the test start (between 35 - 50 seconds). This number was just enough to strain resources on the database machine, making its four cores reach 99-100% usage, but still without

raising exceptions. At a random time (ranging between 1 to 2 minutes to let the original workload stabilize), the *prex4* machine introduced an additional workload, using a ramp-up time, with the intent of pushing the database beyond its limits and raising exceptions. The random time makes the scenario more realistic and less predictable, simulating a sudden burst of interest in the website, as is usually observable in Christmas and Black Friday.

The aforementioned workload introduced by the *prex4* machine was not always the same. Three different workloads were selected so that different scenarios could be simulated. The goal of the experiment was to validate the accuracy of models trained on one workload and being used to predict exceptions in another. The three different workloads are shown in Table 5.2. After 3-5 minutes, the test was stopped and then the procedure was redone until all tests had finished.

TABLE 5.2: Characteristics of the three different workloads

Workload/Scenario	Number of Clients	Ramp-Up time (seconds)	Notes
Heavy Load	256	100	Simulates quick high influx of clients
Medium Load	128	100	Simulates quick moderate influx of clients
Heavy Load with long ramp-up	256	400	Simulates slow high influx of clients

For each stage (training and preventing), each test was run 15 times, which totals 45 full tests (15 for each of the three workloads presented), each containing data for about 10 minutes of execution, with dozens of exceptions recorded.

Besides a training phase, the predictive exception mechanism also involves a prevention phase wherein the developer might specify code to run in case of an alarm – this is the code within the *prevent* block. In our scenario, the e-commerce website was flooded with too many requests (see Table 5.2), but not all of them are of the same value for the company. For example, updates and inserts are usually associated with transactions and important business operations, whereas read operations are more geared towards navigation of the online catalog (where most of the time is spent). Therefore, it makes sense that, upon detecting a possible exception, indicating a heavy workload, a mechanism should prioritize inserts and updates over reads. This way, the important business operations can take precedence, increasing the monetary gain of the company during an overload of clients. This was the mechanism implemented in our experiments.

Upon detecting that an exception was imminent, the modified Shopizer application waited a random interval (depending on the lead-time) before executing the lengthy read operations that lead to exceptions. This has several consequences:

- The insert and update queries are executed without any delay, thus being prioritized and increasing the company's monetary gain.
- The heavy read operations were scattered across a random uniform interval, effectively reducing the load on the database (throttling the rate of requests).
- The Shopizer application, which uses a threadpool, would not be stuck waiting for the lengthy read operations to finish. Instead, it could process other, smaller operations, effectively increasing the overall throughput of the application.
- Due to the reduced load on the database, exceptions might be prevented.

This preventive action is the same as described in the simulation experiment detailed in section 3.7. Although that simulated scenario was geared towards the TPC-W benchmark, the rationale behind the preventive action remains the same and, as the results will show, it is an appropriate action to take when an alarm is triggered.

After running the first tests, where PreX had not yet trained any models, the system was started once more and the coordinator was instructed to train predictions for the aforementioned exception and prediction context. As described in Section 3.5.4, the prediction algorithm involves three parameters: T, k and t. For this experiment, we trained algorithms only for $t = 1$, since it is likely that windows closer to the exception contain more valuable information (it might be interesting to experiment with other values for t in future work). The (T, k, t) parameters submitted for training are seen in Table 5.3. The values were selected to achieve lead-times of 10 and 15 seconds, which are enough for a corrective action to take place. 15 seconds is the time it takes for the database to timeout the queries and trigger an exception, thus making it an interesting value to explore.

TABLE 5.3: All combinations of variables used in the full experiments

T **(Time-Window size)**	k **(Number of merged windows)**	t **(Number of windows to look-ahead)**
2500 ms	4	1
5000 ms	2	1
7500 ms	2	1
10000 ms	1	1
15000 ms	1	1

The PreX model trained each of the classifiers presented in section 4.4.2, for each of these windows. In total, 84 classifiers had to be trained, cross-validated and compared to one another.

Once the system had completed training and determined the best model, the experiments were redone. No code changes were necessary, since the only difference was that the coordinator now had trained models which it could automatically use to perform online predictions.

A final step in the experimental procedure involved running the original Shopizer code unmodified and comparing it with the modified Shopizer code when no predictions were enabled. Since these experiments were focused on performance metrics, they need not be executed for the different kinds of workloads. Thus, they were only focused on the "Heavy Load" workload. However, these experiments were done at a later time and, due to slightly different machine configurations, resulted in different throughput and exception values. Consequently, in order to allow for a fair comparison, these results could not be directly compared with those of the previous runs. The solution was to re-run the experiments with PreX and the "Heavy Load" workload. These were done with 30 runs instead of 15, to further eliminate outliers in the data. Therefore, in this second set of runs, each test was run 30 times and consisted of the following steps: (i) the machines, probes and applications were started; (ii) the administration client marked the start of a run; (iii) the *prex3* workload was introduced; (iv) the *prex4* workload was introduced; (v) the administration client marked the stop of a run; (vi) the applications (probes, coordinator, Shopizer and database) were stopped. The same random intervals as described previously were used between runs, and it should be noted that steps (ii) and (v) were omitted when the unmodified Shopizer instance was used.

In summary, four different sets of runs were carried: (i) runs with PreX, without trained models; (ii) runs with PreX, and with trained models; (iii) runs without PreX (normal Shopizer code) and (iv) run with PreX, without trained models (re-done). Runs (i) and (ii) were done at the same time and compared against each-other to validate the full model in its predictive and preventive capabilities, whereas runs (iii) and (iv) were also done at the same time and compared against each other to assess PreX's impact on performance.

5.3 Comparison Metrics

To compare the preventive exception model with the prior exception handling model, Shopizer was modified to retry the failed operations to a maximum of three times. This means that even if exceptions were caught, the operation itself might be successful on a following retry attempt. Thus, this recovery mechanism can be compared with the aforementioned prevention mechanism. This is summarized in Table 5.4. Note that in the final experiments, where Shopizer was run unmodified and compared with PreX

(without trained models), the recovery action was not used (it would not be a fair comparison).

TABLE 5.4: Comparison of recovery and prevention strategies

Scenario	Strategy	Notes
Without PreX	Retry operation until a maximum of three attempts is reached.	Common recovery strategy
With PreX	Sleep for a random amount of time between 5-15 seconds when alarm (of exception) is triggered (for read queries).	Simple prevention strategy offered by PreX. Distributes load and prioritizes updates.

We introduced profiling code into Shopizer. As such, it was possible to register when heavy operations were started, when they ended, when an exception was caught and, at a later stage, weather the prevention technique was being used. This allowed us to collect the following metrics on each test:

- The count of successfully completed operations (even if they had to be retried).
- The count of exceptions.
- The count of unsuccessful operations (i.e. operations which never finished, exceeding the limit of retries).
- Average count of successfully completed operations per second (successful operation throughput).

With these metrics, it is possible to assess the effectiveness of the proposed model. For example, it is possible to know if the model prevents exceptions at the cost of performance (i.e. throughput), or if it is inefficient (i.e. prevents few exceptions).

5.4 Results

In this section we present the results of the experiments. These results focus on three distinct aspects: (i) first, there is a discussion of the offline analysis of the best classifier performance for several (T, k, t) parameter combinations; (ii) secondly, we present and analyse the results of the prevention mechanism, comparing it with other recovery strategies, once the best of these combinations had been selected; (iii) thirdly, we analyse the performance impact of using PreX, even when i has no trained models available.

5.4.1 Classifier Performance (Offline Analysis)

Table 5.5 presents the results of the different (T, k, t) parameter combinations. FPR stands for False Positive Rate, indicating the percentage of "false warnings". FNR indicates the False Negative Rate, or the amount of "missed warnings". It is worse to have a high FNR than to have a high FPR, since "missed warnings" mean an exception was not predicted, possibly having consequences for business. In addition to performing 10-fold cross-validation for each of the workloads (heavy workload, medium workload, heavy workload with long ramp-up), some of the datasets were used for training and validating on other datasets. This allows for an assessment of the generality of the predictor (i.e. is a good predictor for the heavy workload an equally good predictor for the medium workload?).

Table 5.5: Offline Analysis classifier performance (DNF means Did Not Finish)

		Dataset													
		heavy load (10-fold CV)		**heavy load with long ramp-up (10-fold CV)**		**medium load (10-fold CV)**		**train heavy load val. heavy load w/ long ramp-up**		**train high-load validate medium load**		**train medium load validate heavy load**		**train heavy load w/ long ramp-up val. medium load**	
T	k	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR
1000	10	0.37	0.20	0.33	0.10	0.38	0.20	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF
2000	5	0.15	0.13	0.16	0.14	0.16	0.14	0.17	0.06	0.20	0.16	0.33	0.10	0.99	0.00
2500	4	0.10	0.11	0.20	0.02	0.10	0.11	0.09	0.06	0.11	0.13	0.01	0.01	1.00	0.01
5000	2	0.03	0.04	0.19	0.02	0.04	0.05	0.20	0.04	0.08	0.06	0.12	0.03	1.00	0.00
7500	2	0.01	0.02	0.16	0.01	0.01	0.03	0.01	0.04	0.01	0.03	0.09	0.02	0.16	0.00
10000	1	0.02	0.03	0.12	0.02	0.02	0.03	0.01	0.04	0.01	0.04	0.08	0.02	0.99	0.00
15000	1	0.03	0.05	0.02	0.03	0.02	0.03	0.02	0.03	0.03	0.03	0.06	0.06	0.00	0.00

In general, the best results were achieved with the $(7500, 2)$ and $(15000, 1)$ pairs. In other words, using windows of 7500 ms, grouped in pairs of two, or windows of 15000, without grouping, achieved the best prediction accuracy. This remains the case even when datasets are validated on one another (i.e. training with one and validating on another). It is also clear that an increase in windows quickly makes the data too noisy (i.e. adds too many features and reduces the amount of information about each one), rendering poor results (these results are in line with those of Ivano et al. [27]).

It should be noted that when the heavy workload with long ramp-up dataset is used for training (and even simply using 10-fold cross validation), the results are bad (exceptions are almost never predicted). This is a consequence of less data on exceptions for this dataset. Since the ramp-up time is high, very few exceptions were recorded on the dataset, making it unbalanced and affecting the classifier performance. At the time when these experiments were run, the PreX implementation did not yet offer the functionality of resampling the data to make it balanced, although such functionality was available in the later experiments used to compare PreX's performance impact.

These results show that it is possible to predict these exceptions, and that predictors for different scenarios are accurate on each other's datasets. Since the best results were reached with the $(7500, 2)$ pair (excluding an outlier for the heavy load with long ramp-up time), the next phase of the experiments – prevention – was performed using this predictor.

The best classifier was a Decision Tree classifier (using the C4.5 algorithm, as provided by the J48 classifier in Weka [81]) and used a combination of variables from both machines, namely:

- TCP Passive Opens (from *prex2*)
- CPU Combined Usage (from *prex1* and *prex2*)
- CPU User Time (from *prex1*)
- Used Memory (from *prex1*)

The number of passive opens variable makes sense, as it increases when more clients connect. Similarly, the CPU usage easily allows one to understand in which situation the system is in. If the CPU usage in the database machine increases, consistently reaching 100%, then it is at its maximum load (and in particular, it might be spending much time doing user-mode processing in the database), possibly leading to an exception. In much the same way, if the CPU usage in the server machine decreases, it might indicate that there is a bottleneck in the database (the server would be waiting for the database to finish). When all of these variables are combined with the decision tree, then they allow for the results shown in Table 5.5. It should be noted that no human had to create rules for triggering alert events, everything was machine generated from the data collected at run-time.

As noted in section 5.2, the "Heavy Load" experiments were run a second time due to differences in machine configurations and to allow for a fair comparison with an unmodified shopizer. In these second experiments, the best classifier was obtained using the *REPTree* algorithm. It had an F-measure of 0.945, FPR of 0.13 and FNR of 0.0017. This classifier had a lead-time of 10 seconds, with $T = 5000$ and $k = 2$. Table 5.6 shows a ranking of the 10 best classifiers found in the last batch of experiments.

TABLE 5.6: Best classifiers found in the last batch of experiments

Rank	**Classifier Algorithm**	**F-Measure**	**FPR**	**FNR**	(T, k, t)
1	REPTree	0.945	0.13	0.0018	(5000,2,1)
2	REPTree	0.932	0.015	0.0026	(10000,1,1)
3	REPTree	0.976	0.05	0.0067	(10000,1,1)
4	MultilayerPerceptron	0.861	0.33	0.009	(10000,1,1)
5	MultilayerPerceptron	0.965	0.06	0.019	(10000,1,1)
6	J48	0.976	0.028	0.023	(15000,1,1)
7	RandomTree	0.859	0.345	0.030	(5000,2,1)
8	REPTree	0.942	0.081	0.049	(7500,2,1)
9	RandomTree	0.915	0.096	0.084	(15000,1,1)
10	J48	0.94	0.081	0.049	(5000,2,1)

The top 10 classifiers all used very small values of k, showing that higher values of k lead to bad performance. Indeed, the performance of classifiers where $k = 4$ had, at times, an F-Measure of only 0.5. From Table 5.6 we can also see the comparison algorithm at work: the algorithm with the best F-Measure is ranked sixth due to its higher value for false negative rate. It would be interesting to explore other ranking algorithms as art of future work. The table also allows us to conclude that the best classifiers are based on decision tree algorithms and neural networks. It is also interesting to note that the best classifier was not the one with the highest lead-time. There is an inherent bias when choosing classifiers with $t = 1$ and high lead-times, because if $t = 1$ and the lead-time increases, then so does the size of the prediction window (i.e. the prediction period). A bigger prediction window increases the likelihood that there is an exception in that window and, henceforth, the less precise each model has to be[2].

5.4.2 Prevention Mechanism Results (Online Analysis)

Table 5.7 shows the results of the prevention mechanism. Using the exception mechanism significantly decreased the number of exceptions (by factors between 40 and 100), with some increase (2.2%) in successful operation throughput for the Heavy Load and Medium Load scenario. In the last scenario, throughput decreased by 1%, although this scenario has far less exceptions and, thus, the mechanism is less useful and might add some overhead[3]. These results show that the prediction mechanism worked, detecting exceptions as soon as the database machine started being overloaded with connections. Once these exceptions were being predicted, the prevention action was used with success.

[2]In other words, if the prediction window is sufficiently large, the dataset quickly becomes saturated with "windows with exceptions" and the models become biased towards predicting exceptions, even with data resampling.

[3]The idea that overhead might be at the root of a decrease in performance is further explored in 5.4.3

TABLE 5.7: Comparison of scenarios with and without the prevention mechanism

Scenario	Successful operations (mean)	Unsuccessful operations (mean)	Exceptions (mean)	Successful operations per second (mean)
Heavy Load	4294.6	1.134	207.34	12.65
Heavy Load with PreX	4832.27	0.0	3.8	12.93
Medium Load	4462.33	1.067	200.93	12.64
Medium Load with PreX	4507.4	0.0	2.93	12.91
Heavy Load with long ramp-up	4286.0	0.47	132.2	12.81
Heavy Load with long ramp-up with PreX	4566.73	0.0	1.2	12.62

The effects of the prevention mechanism can be seen in Figure 5.2, where two plots of different runs of the experiment (for the heavy workload) are shown: one with the mechanism and one without it. Throughput starts by increasing, until it peaks (maximum capacity is reached). Afterwards, without the prevention mechanism, several exceptions are raised, although the throughput is roughly the same. When the prevention mechanism is used, there is a slight increase in throughput, and a significant decrease in the number of exceptions. For the Heavy Load scenario, exceptions are decreased by a factor of 54, a change from 207.34 to 3.8 average exceptions per test. For the Medium Load scenario, this factor is 69, a change from 200.93 to 2.93 average exceptions per test. Finally, in the last scenario, there is a decrease in exceptions by a factor of 110, a change from 132.2 to 1.2 average exceptions per test. Note that the prediction mechanism sometimes stopped predicting exceptions (possible false negatives), leading to an increase in operations completed successfully without the preventive action, but also leading to the few exceptions that still happen. This is an indication that if a better prediction model had been found, more exceptions might be prevented. Also note that there are no false negatives before the *prex4* workload begins.

The increase in successful operation throughput can be attributed to a more efficient resource usage. Since the workload is being distributed, and since less time is being spent retrying failed queries, then the overall throughput increases. In addition, note that these results do not reflect the throughput of other operations. For example, since read operations are effectively being "deprioritized", an increase in the throughput of write operations is expected. Thus, for this scenario, the use of the proposed exception prevention mechanism is a valid and valuable technique, with very little additional effort

for the programmer (he/she only needs to code the recovery mechanism, train the model by running the system, and then enable the trained model).

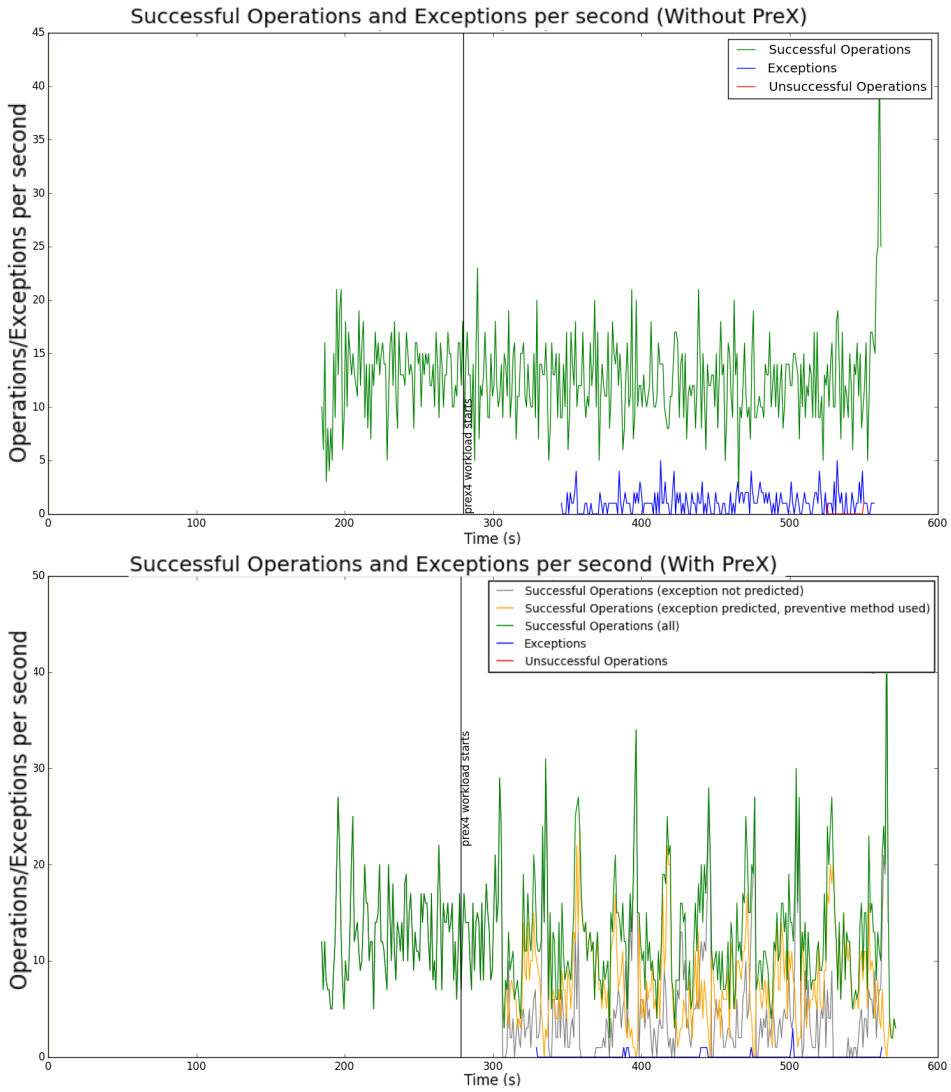

FIGURE 5.2: Operation and exception throughput with and without the prevention mechanism.

Note that the results shown in Table 5.7 and Figure 5.2 compare PreX with and without trained models. Another interesting run to inspect is shown in Figure 5.3. In this run, at around 155 seconds, the coordinator stops triggering alarms. As a consequence, the preventive action is not taken and some exceptions happen. At around 195 seconds, the alarms start being triggered again, temporarily stopping between 210 and 230 seconds. This shows that the predictions aren't perfect, and that there is a clear relationship between prediction, preventive action and exceptions: if no prediction happens (and, hence, no preventive action takes place), then exceptions happen as they would if PreX was not used.

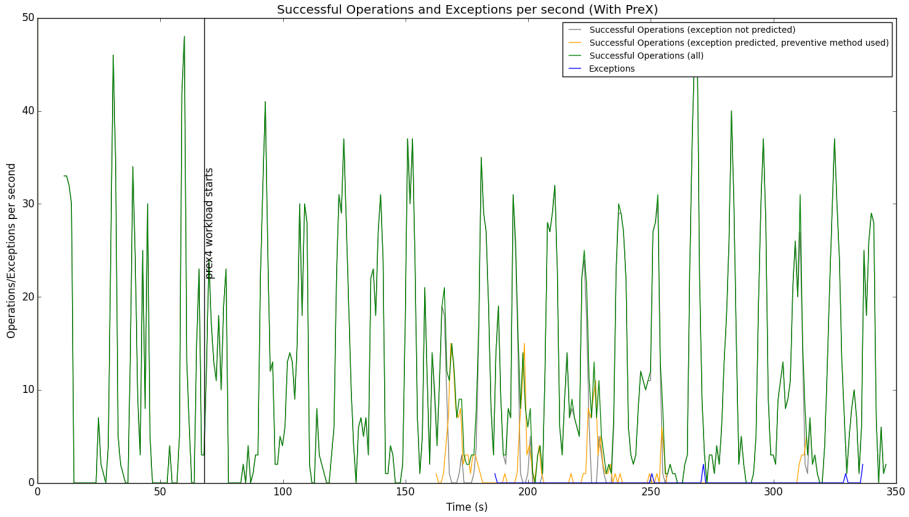

FIGURE 5.3: Operation and exception throughput in one of the experimental runs where PreX was used (with trained models). In this run, there were some false negatives and exceptions.

5.4.3 Impact on performance

One of the main goals of the validation experiments was to evaluate the impact of PreX on the overall system performance (due to possible overhead). The comparison of runs with and without the PreX system (with predictions disabled) allows to perform this evaluation[4]. Table 5.8 shows the results of this comparison.

TABLE 5.8: Comparison of scenarios with predictions enabled or disabled, when using PreX

Scenario	Successful operations (mean)	Exceptions (mean)	Successful operations per second (mean)
Without PreX	4201.5	203.5	12.59
With PreX (no predictions)	3865.4	117.4	12.08

PreX does indeed add some overhead to the system. The operation throughput decreases from 12.59 to 12.08 operations per second, a 4.1% difference. This is also reflected in a decrease of the total number of exceptions. The ratio of exceptions to successful operations with PreX is 0.030, whereas it becomes 0.048 when the system is not used at all. Given this difference, a decrease in the number of exceptions points towards a bottleneck in the *prex2* machine. Indeed, if there was some overhead in the *prex1*

[4]Note that the results "With PreX (no predictions)" are different from those shown in the previous section because, as noted previously, these experiments were re-run independently under slightly different machine configurations.

machine (where the database is hosted), the number of exceptions would increase with PreX because the database would have less CPU time to process the queries and more of these queries would timeout. Similarly, if the overhead was the same in both machines, the ratio of exceptions to successful operations would not change, because both machines would suffer a similar performance hit.

The results are not surprising. The usage of PreX is bound to add some overhead to the system. Firstly, one must consider that probes add increased network traffic. This could become an issue and reduce the total number of operations if the network could not handle the amount of traffic generated by jMeter and the probes at the same time. However, such a scenario is unlikely, as the machines shared a gigabit connection and the data used in these tests is not sufficient to bottleneck the connection. Probes also add increased CPU load to the machines where they are located (*prex1* and *prex2*), but this was inspected during each of the tests and never exceeded 1%. The main reason for PreX's overhead is more likely attributed to the computational overhead of using the PreX library. Although the *try-prevent-catch* mechanism featured in the implementation (Chapter 4) offers a structure which is graphically similar to a native *try-catch*, it makes use of lambda functions which must be created for each execution of their parent function. in addition, these functions are not directly invoked, but the client must be instantiated and exchange an initial protocol message with the coordinator, which further delays the execution of the relevant Shopizer code. Thus, in a native implementation of the PreX model this overhead would be eliminated or greatly reduced.

Further experiments could be performed to assert if these are the true sources of overhead in the system. For example, another scenario could be constructed where the same PreX library client could be re-used, instead of one being created for each operation. This would eliminate the overhead of the initial protocol message with the coordinator. These experiments are a good candidate for future work.

5.4.4 Answer to Research Hypotheses

If we recall the research hypotheses presented previously (see section 5), we can conclude that:

1. **H1**: the hypothesis is accepted. PreX is able to predict exceptions for our scenario with satisfactory accuracy, displaying False Positive Rates of no less than 15% and False Negative Rates of no less than 19%. The best results showed FPR and FNR in the 3-5% range.

2. **H2**: the hypothesis is accepted. PreX can be used to prevent exceptions in an effective way. The number of exceptions dramatically decreased in all three scenarios, by factors of 54, 68 and 110.

3. **H3**: the hypothesis is accepted. PreX offers new techniques that can compete with current revitalization strategies. Using the new model to prevent exceptions, even with a simplistic approach, the successful operation throughput increased about 2% in two of the three scenarios, when compared with a retry strategy. The increase in throughput is significant, with $p < .01$ using a dependent T-Test.

4. **H4**: the hypothesis is accepted. The values for (T, k, t) have a high impact on prediction performance. High values for k, coupled with low values for T lead to worse results. The $(7500, 2, 1)$ and $(15000, 1, 1)$ configurations provided the best performance.

5. **H5**: the hypothesis is accepted. The prediction models showed generality, since when applied to different workloads, performance was still adequate. More different scenarios should also be the focus of future work.

6. **H6**: the hypothesis is accepted. Using PreX, even when no models have been trained, decreases performance by an average of 4.1%.

Chapter 6

Planning and development

The research developed in this thesis is a "High Risk, High Reward" kind of research. As such, it was planned with a focus on careful risk analysis, a thoroughly planned architecture and a publication plan with several milestones. During the first semester, work mostly focused on the state of the art, risk analysis, preliminary experiments (to validate the model's predictive capabilities, available in Appendix B) and the publication plan. In the second semester, work focused on validation of the model's preventive capabilities and in the implementation of the full model. In the following sections, we present an overview of the thesis plan, the risk analysis and contingency plan, and the publication plan. We also note additional research and professional activities carried out during the development of the thesis.

6.1 Overview

The implementation of PreX followed an adaptation of the waterfall model, as shown in Figure 6.1. The first step in the development of PreX was the study of the state of the art. Afterwards, the PreX model was designed, as well as its architecture. The validation experiments were defined, identifying evaluation scenarios, exceptions to predict, and countermeasures and techniques allowed by PreX. These three stages then allowed for a more iterative development methodology. In a first stage, the PreX prototype was implemented implemented for offline prediction of exceptions, performing offline validation and returning to the implementation stage to adjust the prototype. Once offline prediction has reached a mature state, the prototype can then be adjusted for online prediction, using another *implement-validate-implement* iterative cycle.

This adapted waterfall model allowed for a strong focus on theoretical aspects of PreX, providing the foundation for a solid model with promising applicability to real-world

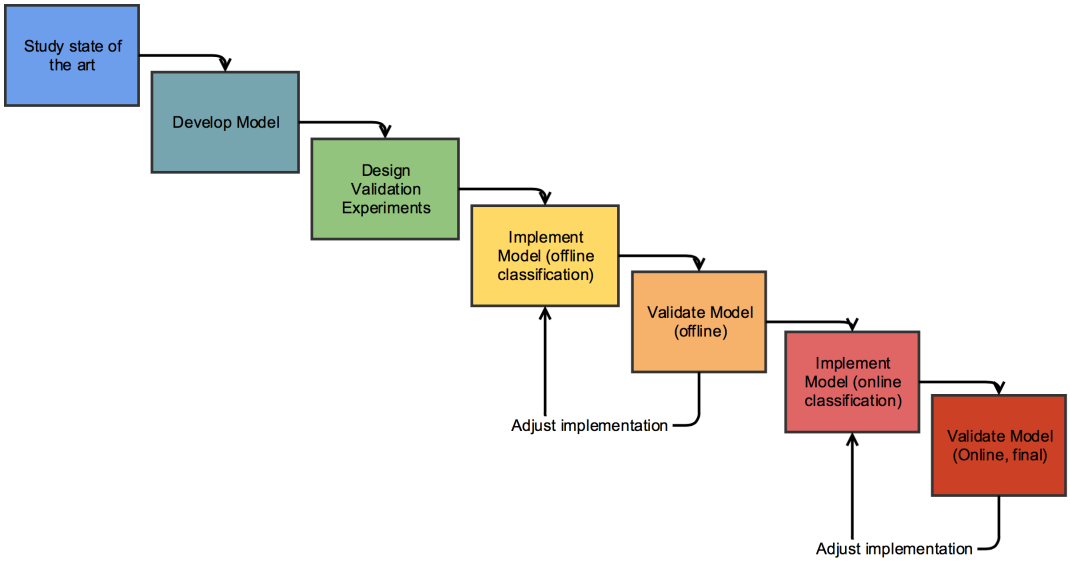

FIGURE 6.1: Project development methodology for PreX.

scenarios. Once this model had been fully defined, the two-phase implementation using iterative development allowed for regular (e.g. weekly) evaluation of the project progress, as well as easy adjustments. A traditional waterfall approach would, for example, only validate the online model. This would delay validation and make adjustments to the implementation harder, since it would comprise much more functionality and lines of code. Thus, this methodology combined the robustness of a well-thought model design process with the benefits of quick development, testing and validation processes. In addition, since online prediction involves the execution of the preventive actions, this division in two-stages clearly separated prediction from reaction to prediction.

In the first semester, the work on PreX produced one published article [82]. The development plan used during that semester can be seen in Figure 6.2, as a Gantt chart. Work was mostly focused on surveying the literature and development of the PreX model, as well the overall architecture of the solution.

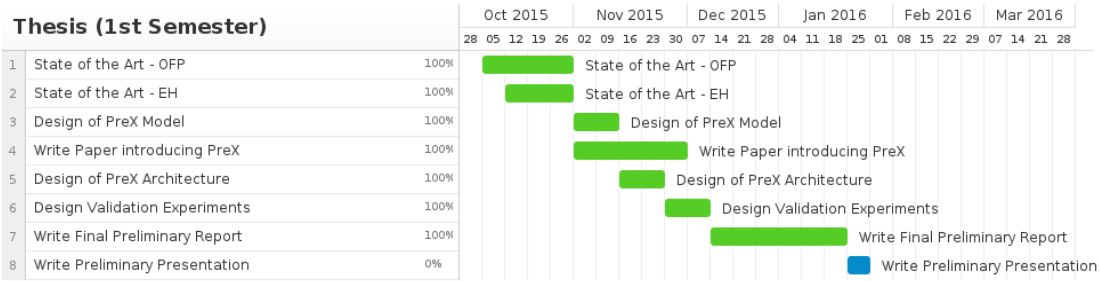

FIGURE 6.2: Development plan (Gantt chart) used during the first semester.

In the second semester, work mostly focused on the implementation of the PreX architecture with a prototype, as well as its final validation. This was done in the aforementioned two stages: first, using offline prediction and, afterwards, using online prediction and preventive actions. In this semester, the PreX model was implemented and validated with two experiments (detailed in chapter 5) and a simulation (detailed in section 3.7), producing different journal articles (see Appendix A).

The work carried out during the second semester can be seen in Figure 6.3, as a Gantt chart. In this chart, the secondment that took place in Brazil (see section 6.4) is also highlightedl

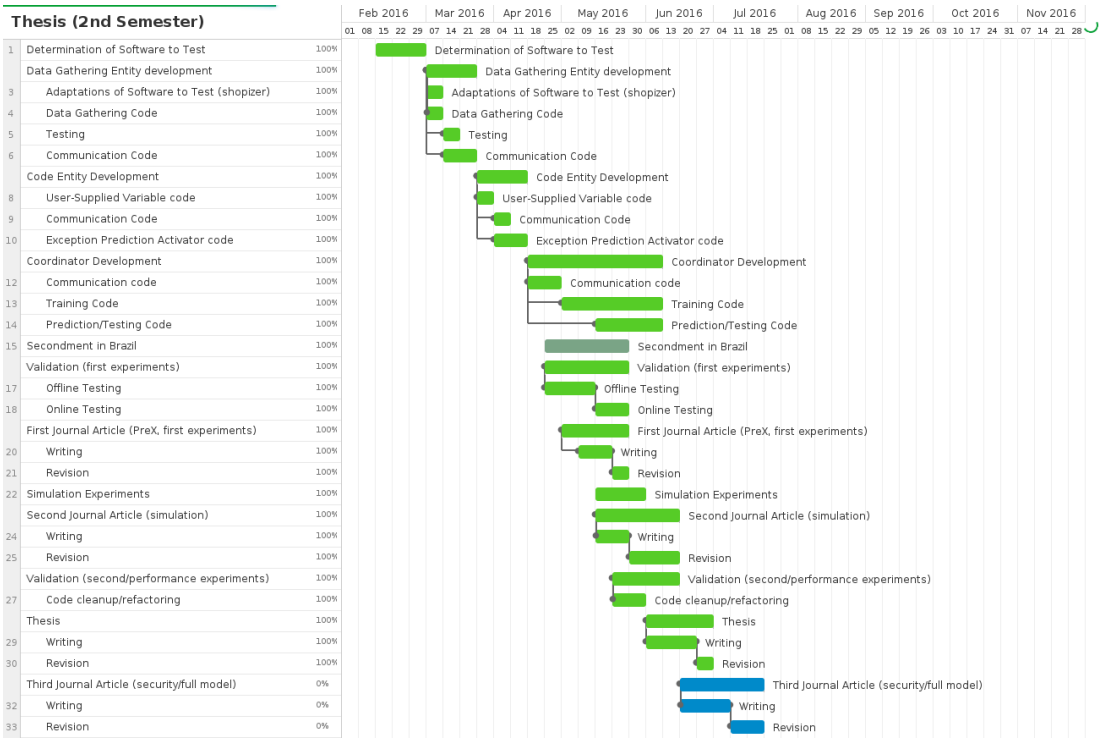

FIGURE 6.3: Work carried out during the second semester, as a Gantt chart.

6.2 Risk Analysis and Contingency Plan

An important part of the development of the thesis involved carefully analysing possible risks and devising ways to prevent them from happening or mitigating their effects. Since this was a "High Risk, High Reward" research, a focus on early risk analysis was extremely important. For this analysis, the several development stages were inspected, looking for potential risks and analysing them according to a likelihood and impact scale. Each risk was then assessed according to its potential impact on the thesis and probability of happening. The impact is graded as Low, Medium or High, with Low implying that the thesis is not greatly impacted by the risk, and High implying that if such risk happens the

thesis may be strongly affected by it. The probability is graded in percentage intervals $[0, 25[$, $[25, 50[$, $[50, 75[$ and $[75, 100]$. The results of this risk analysis can be seen in Table 6.1. For each of the determined risks, a set of mitigation and preventive actions were developed, forming a contingency plan. These can be seen in Table 6.2.

TABLE 6.1: Risk analysis according to impact and probability

ID	Risk	Impact	Probability
1	The model's preventive capabilities might not offer any advantages for developers, rendering its main premise worthless.	Low	[25, 50[
2	Real-world exception data may not be available, affecting model validation and evaluation.	High	[75, 100]
3	The model's predictive capabilities might not be sufficiently good to justify its usage, limiting its preventive capabilities.	High	[25, 50[
4	The model involves the development of a distributed real-time adaptive machine learning system, which may be too difficult to implement and delay the thesis.	Medium	[25, 50[

Four major risks were identified, two of which have a high impact on the thesis. Risk 1 concerns the model's preventive capabilities, which might not offer any real advantage to developers. If there is no use in acting on an exception before it happens, then the premise of the model is flawed and there is little contribution that this thesis can provide. In order to mitigate this risk, a simulation was developed that assessed the overall impact of the model in a real-world system[1]. In addition, several revitalization strategies made available by PreX were designed early on, to make sure that the model was viable. Risk 2 notes that if there is no real-world data available, then the model might not be properly validated and evaluated. Since this risk has a high impact and probability, it is the one on which most effort should be focused. Indeed, early during the development of the thesis a mitigation plan was developed: if there were no data available, a set of real-world experiments, with existing platforms, were to be developed, producing the necessary dataset. Risk 3 notes that even if the model's preventive capabilities are useful, they are limited by its predictive capabilities. To mitigate this risk, a lengthy survey of the state of the art in online failure prediction was conducted, with early preliminary experiments answering the question "can we predict exceptions?", showing that the model could offer accurate predictions. Lastly, Risk 4 is the only one regarding implementation details. Since this model involves developing a distributed system capable of learning, adapting and making predictions in real-time, at the same time having to be tightly integrated with an existing language and the existing exception handling model, it could happen that it would be too difficult to implement. In order to prevent such difficulty from arising, the PreX architecture was carefully planned in the initial stages of the project, so that major implementation roadblocks were caught early on and avoided.

[1] If it was verified that the model was not useful, then the focus of the thesis would have to be changed.

Each of the mitigation/prevention plans devised were essential in the development of the thesis. For example, by acknowledging that real-world data would probably be lacking, an initial focus on preliminary experiments resulted in a published paper [82]. Similarly, the early focus on PreX's architecture significantly reduced its implementation complexity and, indeed, the final implemented architecture bears a high resemblance with the designed architecture.

TABLE 6.2: Mitigation and Prevention actions for each of the identified risks

ID	Risk	Mitigation/Prevention Plan
1	The model's preventive capabilities might not offer any advantages for developers, rendering its main premise worthless.	Perform a simulation experiment to validate the preventive usefulness; design preventive actions made possible through PreX early during the thesis
2	Real-world exception data may not be available, affecting model validation and evaluation.	Plan and execute real-world experiments with existing platforms (e.g. Shopizer).
3	The model's predictive capabilities might not be sufficiently good to justify its usage, limiting its preventive capabilities.	Survey state of the art in online failure prediction and perform early preliminary experiments.
4	The model involves the development of a distributed real-time adaptive machine learning system, which may be too difficult to implement and delay the thesis.	Design a solid and robust architecture very early.

6.3 Publication plan and milestones

The thesis was developed with a publication plan and a set of milestones in mind. The goal was to publish at least a paper in a conference and one in a high-impact journal. As the semesters progressed, these goals were adapted to include other publishing milestones.

The conference paper was submitted, accepted, and published [82] in the WorldCIST'16 conference held in Recife, Brazil. This conference was chosen after analysis of several conferences in the area of software engineering and/or failure prediction which were accepting papers during the first semester. The main conferences in the areas of software engineering and/or failure prediction were surveyed during the first semester, at the end of October, with the intent of submitting an article during the month of November. This would allow for quick feedback on the main research ideas behind PreX.

After the conference paper had been published, a set of additional publishing milestones were defined[2]. In total, the following milestones were used as guidelines for the thesis work:

- **(1st milestone, reached)** November: Submission of short-paper or preliminary results paper to a conference. This paper has been published [82].
- **(2nd milestone, reached)** April: Submission of a full-paper to a journal with a validation of the full model on real-world data. This paper has been submitted [83].
- **(3rd milestone, reached)** May/June: Submission of a full-paper to a journal with a validation of the preventive model's usefulness. This paper has been submitted [84].
- **(4th milestone, ongoing)** July: Submission of a full-paper to a journal with a validation of the of the full, generic, implementation of PreX, possibly in the are of security. This paper is under development.

As a result of these milestones, in addition to the article published in the WorldCIST conference, an article was submitted to the *Journal of Systems and Software*[3] [83], another article was submitted to the *Advances in Knowledge and Information Software Management* journal [84] and, lastly, a final paper is still being prepared, detailing the full implementation of PreX. All of the published and submitted (under the process of peer-revision) papers can be seen as Appendixes to the thesis. Note that the last milestone is still ongoing, and a new journal article is still being written for submission during July.

6.4 Secondment in Brazil

As part of a collaboration with the Universidade Federal de Alagoas (UFAL) in Maceió, Brazil, some of the research in this thesis was done during a one-month secondment in Maceió in the context of the DEVASSES project. During this stay, we collaborated with professor Baldoíno Fonseca dos Santos Neto and his research team, presenting our work and providing feedback and suggestions on their research. The research team also provided valuable feedback for this thesis.

[2]It should be noted that one of these milestones (set in May/June) was the outcome of an invitation to submit an extended version of the WorldCIST'16 paper to the *Advances in Knowledge and Information Software Management* journal.

[3]This journal was chosen after a survey of several top-tier journals where the research in this thesis could be published.

The secondment took place between April and May, 2016, partially overlapping with the date of the WorldCIST'16 conference, allowing for the presentation of the published article. Most of the experiments presented in chapter 5 were done in Brazil.

6.5 Teaching at the University of Coimbra

Parallel with the thesis, during the second semester, work was also done as a monitor for the "Acertar o Rumo" programme of the University of Coimbra. This involved weekly classes of 3 hours with a class of 22 students for the "Programação Avançada em Java" course. There were a total of 16 classes.

This experience, in parallel with the research work being developed and the secondment in Brazil, helped develop academic, research and teaching skills, ultimately helping the development of the thesis.

Chapter 7

Conclusions and Future Work

PreX is a new Exception Handling model that defies nowadays' Exception Handling preconceptions. Current research in exception handling and online failure prediction shows that a fine-grained system for predicting exceptions is currently missing. Instead of catching exceptions, this model proposes that the system, as a whole, actively work towards predicting and preventing exceptions. Applications can then be more resilient, robust, reliable and have increased performance.

In this thesis, we presented PreX as a new preventive model and offered its first implementation, made publicly available at `https://github.com/Jorl17/prex`. The model was validated with different experiments and a simulations.

The results show that PreX, as a model and, in particular, as implemented in this thesis, can be a useful tool to increase the reliability of systems. It is easy to integrate the model into an existing Java project without drastically changing the layout of *try-catch* blocks. A GUI tool is supplied to allow easy interaction with the system. The real-world experiments performed with this implementation and the open-source Java e-commerce Shopizer solution showed that exceptions could be dramatically reduced (by an order of magnitude), with reduced impact in performance. This solution is an inexpensive way of increasing reliability of services, which would ordinarily be increased with higher monetary costs (e.g. by adding replication).

The thesis also shed some insight into ideal (T, k, t) parameters of the proposed failure prediction method. It is clear that high values of k hinder training time and prediction accuracy, although small values might offer the benefit of more temporal information. The values of T are also subject to a problem-dependent sweet-spot and should be carefully examined by system administrators looking to use this model. The new model offers new tools for developers: not only can they proactively use some of their existing

recovery techniques, but they can also employ new techniques made available before an exception happens. These new tools allow for more reliable systems, built with a proactive and self-aware environment.

PreX was fully validated. Its predictive capabilities were validated with two distinct experiments, whereas its preventive capabilities were validated both with a simulation and with a real-world scenario. The main goal of this thesis was, thus, achieved: a new model was proposed, implemented and fully validated, demonstrating its usefulness as a new tool for application developers.

The implementation presented in this thesis is only the first implementation of the proposed model. There is room for much future work. There are other scenarios where it can be applied, particularly in the security domain, where it is useful for applications to selectively shutdown only part of their services if they detect an attack, instead of blocking clients or temporarily denying all access. All of the experiments in this work focused on a value of $t = 1$ to validate the usefulness of the model. However, now that PreX was shown to be useful, work can continue on other values for $t = 1$ (i.e. how reliably can the lead-time be increased with higher values for t?). The current implementation used a range of different classifiers (16), but there are certainly other classifiers and parameter configurations which could be explored as future work. It would also be interesting to deploy PreX in an enterprise environment and evaluate its impact in more "realistic scenarios". Finally, while our experiments focused mostly on exceptions within windows of 5-15 seconds, it would be interesting to validate the model on much larger timescales, where it might be shown that higher values of k offer benefit in this scenario.

On a more personal level, this year was marked by many professional experiences. I continued my research in the area of NoSQL systems and worked on the PreX model and its implementation, facing many fascinating design challenges and decisions. During this time I took a one-month secondment in Brazil, where I had the opportunity of collaborating with wonderful colleagues and professors who provided feedback and additional insight into this thesis. I also had the pleasure of interacting with highly motivated students during weekly classes as a monitor in the "Acertar o Rumo" programme. All of these activities worked together to keep me motivated and learning something new everyday, which I am extremely grateful for.

Appendix A

Publications during the development of the thesis

A.1 Publications

- João Ricardo Lourenço, Bruno Cabral, and Jorge Bernardino. **A predictive model for exception handling**. In *Proc. of the 16th World Conference on Information Systems (WorldCIST)*, 2016 (upcoming).
- João Ricardo Lourenço, Bruno Cabral, Jorge Bernardino and Marco Vieira. **Comparing NoSQL databases with a relational database: performance and space**. In *International Journal of Big Data*, 2016 (upcoming).

A.2 Submissions (under revision)

- João Ricardo Lourenço, Bruno Cabral, and Jorge Bernardino. **A predictive model for exception handling**. Submitted to: *Journal of Systems and Software.*
- João Ricardo Lourenço, Bruno Cabral, and Jorge Bernardino. **Predicting and preventing exceptions for increasing reliability**. Submitted to: *Advances in Knowledge and Information Software Management.*

Appendix B

Preliminary Experiments

To demonstrate that PreX is a feasible model, some preliminary experiments with the TPC-W benchmark were conducted in the first semester. These experiments only intended to validate the main principles of the model, and did not involve a full implementation of PreX as specified in previous sections.

The TPC-W benchmark [85] simulates the activities of a retail store Web site. It defines a set of 14 user transactions that are either of browsing or ordering types. TPC-W has a workload generator that emulates the behavior of users according to pre-specified configurations. In our experiments, the connection pool of the TPC-W server was exhausted due to the overload of clients, resulting in *NullPointerException*s in several different sections of code. Using failure prediction methods, we attempted to predict these exceptions.

In the following sections, a description of the preliminary experiments is given. These experiments were performed offline, using only a limited portion of PreX's architecture.

B.1 Experimental Setup

The setup consisted of three virtual machines running the Crunchbang Linux distribution (Irrera et al. [47] showed that virtualization did not significantly influence failure prediction results). The three machines were allocated with 1 GB of RAM and a single virtual CPU core. They communicated through local network bound to the host machine. One of the machines ran the TPC-W Server, the other the MySQL database, and the third the TPC-W load-generator. In this third machine, a custom-built data gathering tool was placed, sampling data at the rate of 100 samples per second. The

full list of variables is presented in Section B.2. The experimental setup can be seen in Figure B.1.

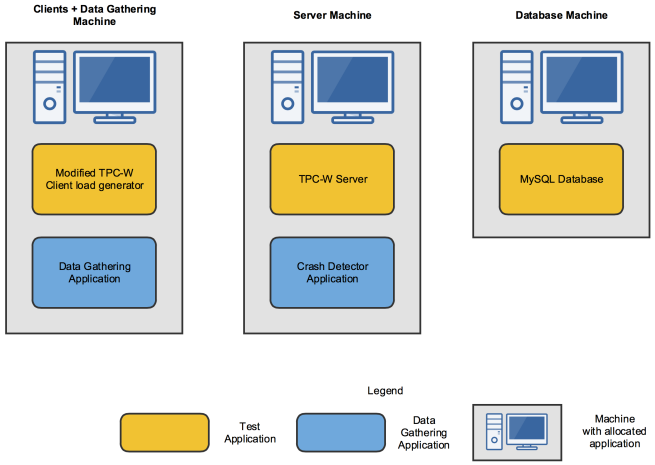

FIGURE B.1: Experimental setup of the preliminary experiment.

B.2 Variables used for prediction

The experiments used a custom-built Java data gathering tool developed on top of the System Information Gatherer And Reporter (SIGAR)[1] library. This library is cross-platform and exposes functionality through Java bindings, making it ideal for quick development and integration with the Java language. In total, 49 variables were sampled, regarding CPU, Memory, Network, Disk, JVM and system statistics. The full list of sampled variables can be seen in Table B.1. Since the Null Pointer exceptions were being caused by connection pool exhaustion in the TPC-W Server, many variables associated with network were sampled.

B.3 Data preprocessing

The SIGAR library does not always provide values for some of its variables, leading to missing values. In these preliminary experiments, these were replaced with the previous recorded value at run-time by our Java application. In future PreX versions, a proper missing value imputation method will be used.

After recording several *golden-runs* (executions with no failure), as well as several executions with failure, the final data was passed to a pre-processing window-based algorithm.

[1] `https://github.com/hyperic/sigar`.

TABLE B.1: Full list of variables sampled for the preliminary experiments

Variable	**Type**	**Subsystem**
CPU Usage (%)	float	CPU
Memory Usage (%)	float	Memory
Swap Usage (%)	float	Memory
Total Swap	int	Memory
Open Connections	int	Network (TCP)
In Errors	int	Network (TCP)
In Segments	int	Network (TCP)
Out Segments	int	Network (TCP)
Retransmitted Segments	int	Network (TCP)
Attempt fails	int	Network (TCP)
Established connections	int	Network (TCP)
Established connection resets	int	Network (TCP)
Passive opens	int	Network (TCP)
RxBytes	int	Network (Interface)
RxDropped	int	Network (Interface)
RxErrors	int	Network (Interface)
RxFrame	int	Network (Interface)
RxOverruns	int	Network (Interface)
RxPackets	int	Network (Interface)
Interface Speed	int	Network (Interface)
TxBytes	int	Network (Interface)
TxCarrier	int	Network (Interface)
TxCollisions	int	Network (Interface)
TxDropped	int	Network (Interface)
TxErrors	int	Network (Interface)
TxOverruns	int	Network (Interface)
TxPackets	int	Network (Interface)
Available free bytes	int	Disk
Total Free bytes	int	Disk
Total Number of Bytes	int	Disk
Disk Queue	int	Disk
Number of disk reads	int	Disk
Number of disk writes	int	Disk
Disk Usage (%)	float	Disk
JVM Major Faults	int	Monitored Process JVM
JVM Minor Faults	int	Monitored Process JVM
JVM resident memory	int	Monitored Process JVM
JVM shared memory	int	Monitored Process JVM
JVM total memory	int	Monitored Process JVM
JVM Number of Threads	int	Monitored Process JVM
JVM Open File Count	int	Monitored Process JVM
JVM Heap Usage	int	Monitored Process JVM
Number of Idle Processes	int	System Processes
Number of Running Processes	int	System Processes
Number of Sleeping Processes	int	System Processes
Number of Stopped Processes	int	System Processes
Number of Threads	int	System Processes
Number of Zombie Processes	int	System Processes
Total Number of Processes	int	System Processes

This algorithm involved the two stages of PreX's feature set construction algorithm described in Section 3.5.4: (i) *time-window construction* and (ii) *window-merge*. This algorithm has three configurable parameters:

- T: The size of each *time-window* within which data is summarized with the *mean*, *standard deviation*, *maximum*, *minimum* and the *derivative* in the *time-window construction* step.
- k: The number of *time-window*s to merge in the *window-merge* step.
- t: The number of *time-window*ss to "look-ahead" for prediction and label assignment during the *window-merge* step.

The ideal values of these parameters is impossible to know a priori, with some values leading to better predictions than others. These values also affect the lead-time (Δt_l) and data validity time (Δt_d), since $\Delta t_l = t \times T$ and $\Delta t_d = T \times k$. Thus, it is valuable to test several combinations of these parameters. To this end, a lead-time of 10 seconds with $t = 1$ was selected. 10 (seconds) is a "sensible" value to predict this kind of exception and allow for potential counter-measures, such as reducing the rate of requests. In addition, by choosing $t = 1$, the data used for prediction is the one closest to the exceptions themselves, which might provide better results. Future experiments involving $t \neq 1$ would allow to truly assess the impact of this parameter.

Thus, several different parameter combinations were tested, leading to different datasets used for classification. The full list of tested values can be seen in Table B.1. Note also that, since there are 49 variables, each *time-window* contains $49 \times 5 = 245$ variables/features.

TABLE B.2: All combinations of variables used in the preliminary experiments

T **(Time-Window size)**	k **(Number of merged windows)**	t **(number of windows to look-ahead)**
1000 ms	10	1
2000 ms	5	1
2500 ms	4	1
5000 ms	2	1

B.4 Classifier Selection

The experiments were done using the WEKA[2] toolkit, specifically, the Weka Knowledge Explorer Graphical User Interface. The dataset was very large, with some combinations

[2] `http://www.cs.waikato.ac.nz/ml/weka/`

of parameters yielding as much as 2450 features. Since it was not the focus of these preliminary experiments, no data reduction or feature selection was performed. This combination of factors limited the availability of some classifiers, which took too much time to train on the datasets of as much as 3030 instances.

As these are the first experiments with PreX, a decision was made to mostly try different kinds of classifiers. Thus, the following WEKA classifiers were selected:

- **J48**: An implementation of the C4.5 algorithm used for building decision trees [86].
- **RandomTree**: A decision-tree classifier built with K randomly selected attributes at each node. Fast to train, should not achieve particularly good results, useful for baseline comparisons.
- **Logistic**: An implementation of a modified version of Cessie and van Houwelingen's [87] multionomial logistic regression model.
- **Naive Bayes**: A Naive Bayes classifier using estimator classes. Numeric estimator precision values are chosen by WEKA based on analysis of the training data.
- **SMO**: An implementation of John Platt's sequential minimal optimization algorithm for training a support vector [88] classifier.

The default WEKA parameters were chosen for all classifiers, as precise parameter choie is reserved for future work.

B.5 Dataset Generation

The data were generated by executing a modified version of the TPC-W framework. The TPC-W framework allows for a certain number of clients, Remote Browser Emulators (RBEs) to be configured. A high number of RBEs leads to quicker Null Pointer Execeptions because the connection pool is exhausted faster. In addition, low values might not lead to any exception at all. Thus, the number of RBEs is also a parameter worth investigating. However, since PreX should be flexible and work under different environments, it was decided that data would be gathered for different RBEs and fed to the classification algorithm regardless of the number of RBEs. The classifiers should then, hopefully, be more than general than if experiments were limited to a fixed number of RBEs, at the possible cost of classification accuracy.

The experiments were done twice and at different times. The idea was to assess the impact of different system characteristics on the classification process. A classifier could be trained for the first batch of experiments and then tested on the second batch, and vice-versa. If each of the two datasets had be generated in consecutive executions of the TPC-W framework, data within each dataset could implicitly be skewed by the system characteristics at that point in time, leading to corresponding skewed results and classifiers – it is therefore interesting to assess the performance of classifiers trained with one dataset and validated with the other, generated at different times.

An initial empirical analysis of the relationship between exception occurrence and the number of RBEs was performed by gradually increasing the number of RBEs until exceptions started happening. A value of 400 RBEs was found to occasionally lead to exceptions, with successive increases leading to faster and faster exceptions. Using this original empirical analysis, and looking to gather data regarding both failure and failure-free executions, the experiments were executed for the following 9 numbers of RBEs: 400,500,600,700,800,900,1000,1100,1200. At 1200 RBEs, exceptions were almost always triggered. However, when the experiments were done a second time, the impact of the different system characteristics was instantly noticeable, leading to exceptions much earlier in the process. As a result, for this second run, executions started with the number of RBEs set to 200.

For each of these values of RBEs, 30 different executions were required, so as to gather more data and attenuate possible outliers in the data. Thus, for example, for the first dataset, a total of 270 (9×30) executions were monitored. TPC-W executions where no exception was registered were stopped after 3 minutes, since it was verified that exceptions were never triggered after this time.

In summary, two datasets were generated at different times, under different system characteristics[3]. Each of these datasets consisted of about 270 TPC-W executions (with 9 different numbers of RBEs), including golden-runs and failure runs. The two datasets shared clearly different characteristics, as a simple ad-hoc analysis of the data showed.

Once the TPC-W executions had been finalized, the data was processed in *Matlab*, with the aforementioned data preprocessing steps, and validated in two different ways: using 10-fold cross validation, and training with the first dataset and validating on the second.

[3]The second dataset was generated after the virtual machines had been used for day-to-day routines, such as web-browser and software development.

B.6 Results

Table B.3 presents the results for both datasets using 10-fold cross-validation [4]. FNR stands for False Negative Rate and FPR stands for False Positive Rate. Both of these values are given as a percentage. The former measures the exceptions that were not correctly predicted by the classifier, whereas the latter measures situations where exceptions were incorrectly predicted. In the case of PreX, FNR is of much more value, since false negatives lead to exceptions, whereas false positives might only induce corrective action without it being necessary.

TABLE B.3: Preliminary results for the two datasets using 10-fold cross-validation.

			Classifier									
			J48		**Logistic**		**NaiveBayes**		**RandomTree**		**SMO**	
Dataset	T	k	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR
1	1000	10	46.88	0.27	15.625	0.55	6.25	3.07	62.5	0.59	15.625	0.17
1	2000	5	20.97	0.17	14.512	0.51	4.84	1.35	17.75	0.48	1.61	0.13
1	2500	4	14.46	0.24	8.434	0.55	2.41	4.42	20.49	0.52	6.03	0.13
1	5000	2	25	0.68	25	1.71	0.83	13.4	25.83	1.03	25.83	0.13
2	1000	10	16.13	0.25	19.36	0.88	9.68	0.87	32.25	0.75	6.445	0.37
2	2000	5	11.37	0.186	20.45	1.43	6.82	4.35	25	0.43	18.18	0.31
2	2500	4	9.80	0.186	N/A	N/A	7.84	1.87	9.81	0.43	1.96	0.19
2	5000	2	25	1.11	27.5	2.48	1.25	8.882	25	1.30	26.25	0.12

The results show that, using 10-fold cross-validation, a Naive Bayes classifier produced the best false negative rate of 0.83% and a false positive rate of 13.4%. In other words, the algorithm only failed to predict failures 0.83% of the time, and inaccurately predicted a failure 13.4% of the time (note that the authors of [25] think that such a value, in their work, would be "too high"). This result was achieved for $T = 5000$ and $k = 2$. In general, most classifiers[5] become better at predicting exceptions for lower values of k and higher values of T. This should be further focused on in future work, but it might be due to smaller feature vectors and, thus, better training. Note that the best classifiers, regardless of the combination of T and k, seem to be NaiveBayes and SMO. Additionally, note that, again, the best results for the second dataset are achieved for $T = 5000$ and $k = 2$, using the same WEKA classifier (NaiveBayes).

These values encouraged future work which became part of this thesis. Although preliminary, they showed that it is possible to predict exceptions with a lead-time of 10 seconds. However, as we have noted before, an additional validation step was performed, where classifiers were trained on one dataset and validated on another. Table B.4 presents these results. The first column indicates which dataset was used for training (the other one being used for validation). Unfortunately, the training process for this validation

[4] The data for the second dataset using the Logistic classifier is missing, symbolized by the use of "N/A".

[5] With the exception of the RandomTree classifier, which selects random features.

proved too computationally expensive and, thus, there are many results which could not be computed in time. These are marked with *DNF* (Did Not Finish).

TABLE B.4: Preliminary results for the two datasets using one dataset for training and the other for validation.

			Classifier									
			J48		**Logistic**		**NaiveBayes**		**RandomTree**		**SMO**	
Dataset	T	k	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR
1	1000	10	51.61	0.18	100	0.55	DNF	DNF	41.93	0.12	25.81	1.06
1	2000	5	18.18	0.43	DNF	DNF	DNF	DNF	9.09	0.48	0	73.54
1	2500	4	9.80	0.12	DNF	DNF	1.45	41.18	74.5	0.52	100	0
1	5000	2	45	0.12	DNF	DNF	DNF	DNF	22.5	1.03	DNF	DNF
2	1000	10	15.62	8.99	100	0	DNF	DNF	68.75	0.75	100	0
2	2000	5	32.25	2.97	0	100	30.65	5.44	52	0.43	0	100
2	2500	4	13.25	1.80	N/A	N/A	25.4	2.55	35	0.43	100	0
2	5000	2	100	0.21	0	100	24	7.01	30	5.15	0	100

These results give us valuable insight. The original results were clearly a consequence of overfitting to the particular system state and characteristics. In particular, the SMO classifier, which was one of the best classifiers in the previous results, now presents false negative rates of either 0% or 100% indicating that the classifier always classified the data as belonging to only one class. The best result, for the SMO classifier, is a false negative rate of 25.81%. In general, most classifiers show worse performance under this scenario, but the best result is still achieved by a NaiveBayes classifier, with a false negative rate of 1.45 for $T = 2500$ and $k = 4$ when training with the first dataset and classifiying with the second. However, when the datasets are swapped, the false negative rate drops to 25.4%. In addition, the 1.45 false negative rate had a corresponding false positive rate of 41.18%, which is excessive for practical scenarios.

The J48 classifier achieves a false negative rate of 9.90% for $T = 2500$ and $k = 4$, while maintaining a very low false positive rate (0.12%). If the datasets are swapped, the results are similar, with a false negative rate of 13.25% and a false positive rate of 1.80%. Thus, this seems to be the best result using this form of validation. In fact, the J48 classifier seems to produce the best results. The NaiveBayes classifier also produces results which fail to predict at most 31% of exceptions.

Appendix C

Publication: A predictive model for exception handling. In Proc. of the 16h World Conference on Information Systems (WorldCIST), 2016

A predictive model for exception handling

João Ricardo Lourenço[1], Bruno Cabral[1], and Jorge Bernardino[12]

[1] CISUC – Centre of Informatics and Systems of the University of Coimbra
FCTUC – University of Coimbra, 3030-290 Coimbra, Portugal
[2] ISEC – Superior Institute of Engineering of Coimbra
Polytechnic Institute of Coimbra, 3030-190 Coimbra, Portugal
joaoml@student.dei.uc.pt, bcabral@dei.uc.pt, jorge@isec.pt

Abstract. The exception handling mechanism has been one of the most used reliability tools in programming languages for over four decades. Nearly all modern languages have some form of "try-catch" model for exception handling and encourage its use. Nevertheless, this model has not seen significant change, even in the face of new challenges, such as concurrent programming and the advent of reactive programming. As it stands, the current model is reactive, rather than proactive — exceptions are raised, caught, and handled. We propose an alternative exception handling model — PreX — where exceptions are no longer caught but, rather, predicted and prevented. Online Failure Prediction techniques generally work at a very high level, showing potential for prediction of program crashes. However, these techniques have never been at the hands of the programmers as an effective tool to improve software quality. By applying recent advances in Online Failure Prediction, PreX aims to fully prevent exceptions, bringing failure prediction techniques to a much more fine-grained level that the programmer can control. Predicting exceptions enables a range of preventive measures that enhance the reliability and robustness of a system, offering new revitalization strategies to developers.

Keywords: Exception Handling, Online Failure Prediction, Self-Healing

1 Introduction

The Exception Handling (EH) mechanism was proposed by Goodenough [1] and has been one of the most used reliability tools in programming languages for more than four decades. This model has gone unchanged, even in the face of concurrent software and programming languages for multi-core platforms (e.g. Scala, Erlang and Elixir). Thus, the sequential Exception Handling model remains the preferred Exception Handling model [2].

However, the ubiquity of the Exception Handling mechanism for error recovery does not imply its most correct or desirable usage. Most of the time, developers use EH language constructs as a way of hiding problems, performing log activities or informing the user of unexpected behavior, rather than recovering from it autonomously [3]. This kind of use of EH might be considered a

symptom of a design flaw in the mechanism – the system only acts when it is too late, thus making the problem unavoidable.

In this paper, we propose a new approach to Exception Handling, by providing the means for developers to act on an exception before it happens, thus broadening the range of their revitalization strategies. The approach reshapes the concept of "try-catch" blocks, so that programmers can be alerted of potential exceptions within a given time frame and take some action, much like in the conventional Exception Handling mechanism. This new approach is called **Preventive Exception Handling (PreX)**, and applies the methods and techniques of the Online Failure Prediction field to the lower-level concepts of programming languages, thus empowering programmers to act proactively. PreX introduces a new model for Exception Handling, with the goal of being easy to use, practical, and a successful integration of the fields of Exception Handling and Online Failure Prediction, as well as the first to act on potential exceptions before they have happened – errors are avoided, rather than handled. In contrast to Online Failure Prediction, which operates on a much higher level of abstraction, PreX allows programmers to produce code that is aware of possible predictions during its execution, so that very fine-grained solutions for exception prevention and reaction to potential exceptions can be applied. By acting on exceptions before they happen, developers get an increased range of techniques for dealing with exceptions. Thus, while traditional Exception Handling techniques can still be used, there is potential for new and hopefully more efficient strategies.

To illustrate the motivation for this new model, consider a system consisting of at least a database and several client applications. Consider also that these client applications are write-heavy, meaning they processes several thousands of write operations per second, sending them to the database. Due to the heavy load, the database may slowly become unresponsive and ultimately trigger a *ConnectionTimeout* exception on one of the applications. That application will then have to attempt to reconnect, and restart where it was previously, if such is really possible. This shows the aforementioned downfall of the conventional Exception Handling mechanism – the system only reacts to exceptions, it does not prevent them. Our motivation stems from this issue – we believe that the client applications would benefit from a prediction (i.e. a warning) that the database may trigger a *ConnectionTimeout* exception. With such a warning, they can, for instance, proactively slow their execution rate and prevent the exception from happening. Ultimately, slowing execution down could prove to be more efficient than triggering the exception and restarting the whole process.

The main contributions of this work are as follows: a) explanation of the need for a paradigm shift in Exception Handling; b) proposal of a new model for preventive handling of exceptions; c) presenting results that evidence the feasibility of the proposed model in the prediction and handling of likely exceptions.

The remainder of this paper is structured as follows. Section 2 presents background in Exception Handling models and mechanisms and Online Failure Prediction. Section 3 details the proposed model. Section 4 presents preliminary results of the model. Finally, Section 5 contains our conclusions and future work.

2 Background and Related Work

Exception Handling separates the operation domain into two distinct domains: the operation's *standard domain*, and the operation's *exceptional domain*. Normal program flow, absent of errors, is contained in the *standard domain*. If an exception or error condition is encountered, an exception is *raised*, followed by the invocation of an *exception handler*, that, in theory would deal with the abnormal condition and correct it. This invocation is done by a *signaller* (i.e. *callee*). An *Exception Handling Model* defines the interaction between the signaller and its handler. An *Exception Handling Mechanism* defines the language constructs within a programming language to express a given Exception Handling Model. [4]. In this section, we give an overview of the classical and current Exception Handling models and their limitations.

2.1 Exception Handling Models

Yemini et al. [5] identified four distinct Exception Handling Models, summarized in [6]:

- **Resumption model** – When an exception is raised, the control flow is transferred from the raise point to the handler and, after the exception has been handled, it is transferred back to the raise point. This model effectively binds the caller and the callee together and is prone to recursive resumption, thus being difficult to implement [6].
- **Termination model** – An exception is raised within a protected block, with the control flow transferred to the handler, terminating any intervening blocks. The control flow then resumes as if the protected block terminated without any errors. This is the most widespread model in use [6].
- **Retrying model** – The signaller is invoked after some operation has been made. This model is more appropriate to transient faults, where retrying the invocation might lead to no exceptions. The main disadvantages of this model are its inherent implications for non-idempotent operations, counters, etc – the programmer must be wary of how the code executes.
- **Nonlocal transfer** – The program flow can be transferred to any other location in the program. This model has the obvious drawback of being hard to maintain and much more error-prone [6].

Modern advances in Exception Handling, such as concurrent exception handling or alternative models in the context of the actor model and Functional Reactive Programming, still share a common characteristic with these classical models – an exception is raised, and only then can corrective actions happen. This, unfortunately, means that many incorrect uses of Exception Handling Mechanisms arise, with programmers focusing on hiding errors, rather than attempting to fix them [3]. PreX intends to shift the current practice and allow programmers to act before problems arise.

2.2 Failure Prediction

Recent trends in industry and academia have triggered a shift to new efforts on autonomic computing, trustworthy computing, recovery-oriented computing and other techniques for proactively handling failures. Several techniques have been proposed and used with success, to an extent, in Online Failure Prediction systems. Salfner et al. [7] present a lengthy survey and taxonomy of online failure prediction systems. In their work, these authors explain how Liang et al. [8] explored temporal and spatial correlation to successfully predict hardware component failures in IBM's BlueGene/L. Cheng et al. [9] presented an approach for failure prediction within a high availability cluster system. They showed that they could improve the availability due to accurate prediction and recovery mechanisms (backup nodes and system administrator notifications). Vilalta et al. [10] propose the eventset method, using a data mining approach (a rule-based model). Under specific conditions, they have a false negative error of only 0.16, although this value can be as extreme as 0.83 under other system conditions (false positives are always lower than 0.1).

These and other results show that Online Failure Prediction can be successfully used to predict failures. However, little work has been done for predictions at a more fine-grained level. Predictions are usually made at the system level, at most predicting a generic "crash" of some component. Thus, while promising, these techniques have no practical use for developers who wish to provide specific counter-measures when faced with the possibility of an exception. In this sense, some work has been done in the field of self-healing systems. For example, Magalhães and Silva [11] propose a general self-healing proactive framework for web-based applications. Their work introduces a general framework to create self-healing transactional web-based systems. The framework, although operating at a lower level than traditional Online Failure Prediction methods, does not support run-time notifications at code-level nor operate at the fine-grained level that might be desired for applying more efficient preventive measurements.

3 A New Model for Exception Handling – PreX

PreX is an Exception Handling model that focuses on preventing exceptions rather than catching them. The central idea was depicted with the example given in Section 1: it could be more efficient to temporarily reduce the throughput of a write-heavy application than to catch a *ConnectionTimeout* exception and have to restart the process. We believe that there are other scenarios, similar to this, where systems and developers would benefit from an easy-to-use proactive model for Exception Handling. We now present PreX. Note that this is not a formal description of the model, such work is out of the scope of this article due to the space limitation.

Preventing exceptions implies predicting them. To this end, the area of Online Failure Prediction provides valuable insight. There have been successful failure prediction systems, but these operate on a much broader level. In order to predict exceptions, the proposed model needs to adapt failure prediction techniques to a

per-exception basis. Furthermore, since no two systems are alike, the prediction models will have to be trained for specific deployments. Thus, the PreX model comprises different phases from development to successful prevention:

1. **Coding phase**. The programmer develops the application using a new set of programming language constructs introduced by the PreX model. These are similar to traditional *try-catch* blocks seen in several languages.
2. **Training phase**. In this phase, different machine learning algorithms are applied (after feature selection, data pre-processing, etc), determining which is the most applicable to the specific exception. Data is gathered for different runs of the application, using resource monitoring facilities.
3. **Detection phase**. The application is deployed with the trained model and executed. The model is used to detect potential exceptions. If the trained model becomes ineffective due to changes in environment conditions, the training phase might be required again. Alternatively, self-adapting pattern recognition algorithms can be used.
4. **Prevention phase**. If an exception is predicted, the application can apply preventive measures and try to avoid the potential exception from effectively being raised.

In the following sections, each of these phases is detailed from the perspective of the syntax and semantics of the model, followed by an overall perspective of the architecture and the necessary components of PreX. Lastly, the behavior of the system during its different phases is presented.

3.1 Syntax & Semantics

An example of the syntax of PreX is shown in Figure 1. Pseudo-code similar to Java is used. A database connection is established to send a number of pending writes (sent at line 15). PreX provides semantics similar to traditional *try-catch* blocks, although two different constructs are added to the language: *try { }-prevent(){ }-catch(){ }* and *try{ }-prevent_ async(){ }-catch(){ }*. Additionally, in PreX, predicting an exception does not necessarily terminate the execution of code within the *try* block (as opposed to the *termination* model). Instead, execution is resumed as if the exception had not been predicted, because preventing an exception should not halt normal execution of the current code.

If an exception is predicted, it can be handled in two different ways:

- **Synchronously**: execution within the *try* block is suspended and flow is transferred to the *prevent* block. In normal circumstances, the execution is then transferred back to the previous code within the *try* block.
- **Asynchronously**: execution within the *try* block continues normally, and the *prevent_ async* block is executed asynchronously.

Thus, the *try* block denotes the scope during which a program cares about predicting some exception, the protected region. This exception may then be predicted and raised synchronously. If the exception is not predicted or cannot be

prevented, it is effectively raised by the code, and the traditional (synchronous) Exception Handling model is used, with program flow being transferred from within the *try* block to the *catch* block.

When using the synchronous approach, predicted exceptions are not raised at just any point in time within the *try* block. Within this block, every program statement will be executed without any interruption from the exception prediction mechanism. The flow of execution only moves to the prevention handler between statements, thus eliminating the need for propagation in the call stack. Thus, unlike traditional Exception Handling models, PreX uses a static binding approach, meaning that exceptions are not propagated. Instead, they are tied to the handler of the *try-prevent-catch* or *try-prevent_ async-catch* to which they belong. Only the closest code, within the exact context of the particular exception, can know how to react to a specific prediction. If developers wish to prevent the delivery of predicted exceptions during the execution of a set of statements, a special *no_ interrupt* keyword can be used to denote a new scope within which interruptions are not possible. In Figure 1, lines 13-16 belong to one of these scopes. In the asynchronous approach, as in the synchronous approach, exceptions are only delivered when the program flow is outside the protected block. Thus, the *no_ interrupt* keyword can act as a synchronization primitive between the prediction asynchronous handler code and the code within the *try* block.

PreX allows programmers to periodically sample variables that they think will be useful for prediction, in addition to system variables monitored with custom probes. For instance, the remaining number of operations left might be useful in predicting connection timeouts. These variables can be supplied to the prediction system at any time using the *sample (<variable_ name>,<variable_ value>)* construct. Furthermore, since no two systems are alike, the prediction models will have to be trained for specific deployments. In particular, note that, for instance, *ConnectionTimeout* exceptions may be different depending on the workload (e.g. "write-heavy" vs "read-heavy") or the variables being provided by the program. This motivates the need to distinguish different blocks of code and assign them meaningful names. Thus, the *try* keyword requires an additional argument that uniquely identifies the block of code that it encloses: *try(<prediction context>)*. Write-heavy blocks can then use *try("write-exceptions")*, whereas read-heavy blocks can use *try("read-exceptions")* to handle two completely different models for the same kind of exception (*ConnectionTimeout*) under different contexts. The argument of the *try* keyword is the *prediction context*. Several exceptions can be predicted within the same *prediction context*, and a *prediction context* binds training data and a prediction model to a unique name. An example of this construct is in line 10 of Figure 1.

To train the model, the system administrator may specify which *prediction context*s he/she wants the program to be trained in during a training phase. When in this training phase, no prediction mechanism is used in those *prediction context*s and exceptions can only be caught. Data with the *sample* keyword is still fed to the training mechanism, and if the exception is raised and caught, this training mechanism is notified to adapt its prediction models.

```
1  // Connect to database and build list of data to write
2  DBConnection c = connectToDB();
3  List<Write> writes = fetchWrites();
4
5  // Try to predict exceptions using the "write-exceptions"
6  // prediction context  (a)
7  try("write-exceptions") {
8      for (int i = 0; i < writes.size(); i++) {
9          // Feed writes left to the prediction system
10         sample('writes_left', writes.size()-i);  (b)
11         // Write the data through the connection
12         // Don't allow prediction interruptions
13         no_interrupt{  (c)
14             print("Writing data!");
15             c.write(writes.get(i));
16         }
17     }
18 } prevent ( ConnectionTimeout e ) {  (d)
19     // Sleep for some time to prevent an exception
20     sleep(1000);
21 } catch ( ConnectionTimeout e ) {
22     // The exception could not be avoided...  (e)
23     error("Database timeout!");
24 }
```

Fig. 1. Example of PreX using synchronous *try-catch-prevent*.

3.2 Architecture

Within PreX, exceptions can be predicted using system-wide information. Several entities can share information used for prediction (more data, when appropriately filtered, implies better predictions). An entity need not be an independent machine on its own, and different entities might share the same machine. There are three main kinds of entities within a system using PreX:

- **Coordinator entities**: A (potentially replicable) coordinator entity, responsible for aggregating the data from the other two types of entities and running the prediction system.
- **Data gathering entities**: These entities feed periodic samples of data (e.g. memory and CPU usage) to the coordinator entities. Most of the prediction data comes from these entities, which don't execute any code that wants to predict exceptions. The sampling rate for each of these variables is not pre-determined and may vary according to system load and characteristics.
- **Code entities**: Whenever an application wants to predict exceptions (or train that prediction), it spawns a code entity that connects to its coordinator entity. Each code entity may want to register with the coordinator for a certain kind of exception within a *prediction context*. These entities are then notified whenever an exception of that kind has been predicted. It is the responsibility of the code entity to raise the local exception within the code.

It is then clear that each *try-predict-catch* or *try-predict_ async-catch* block spawns a new code entity. The coordinator entity is responsible for running the failure prediction methods for predicting exceptions. These can be selected *a priori* by a system administrator during the training phase.

The behaviour of the code and coordinator entities is different during the training phase. During this phase, the code entity registers that it will be sending

data and information regarding a given *prediction context* and a given exception. The coordinator then uses this data to train the model.

Since we are dealing with a distributed system, it is complex to generate samples at the exact same time in every entity. For this reason, the coordinator entity groups data within time windows. Precise information about when an event (e.g. a data sample) happened is lost in favor of a more general interval during which several events happened. This data can then be processed using Online Failure Prediction methods such as those presented in [10], [12] or [13].

3.3 Behaviour

To illustrate the behaviour of PreX, Figure 2 depicts the interaction between code, entities, and the prediction system for the code example in Figure 1 during a prediction. Notice that the code uses the synchronous prediction model. In the asynchronous prediction model, the *no_ interrupt* keyword would not be allowed. Also note that if the code was executed during the training phase, no exceptions would be predicted, so sections *c* and *d* would not be entered.

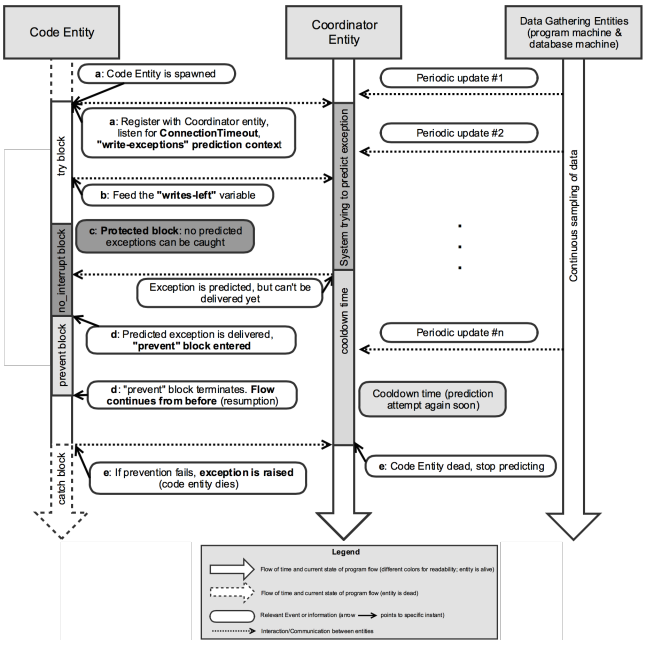

Fig. 2. Interaction between code and entities through time for the code example in Figure 1. For simplicity, in this example, the loop only runs once.

4 Preliminary Results

We have conducted experiments with the TPC-W benchmark to demonstrate that the proposed model is feasible in the prediction and handling of likely exceptions. TPC-W simulates the activities of a retail store Web site. It has a workload generator that emulates the behavior of users according to pre-specified configurations. In our experiments, the connection pool of the TPC-W server was exhausted due to client overload, resulting in different *NullPointerException*s. Using failure prediction methods, we attempted to predict these exceptions.

The setup consisted of three virtual machines running the Crunchbang Linux distribution (Irrera et al. [13] showed that virtualization did not significantly influence failure prediction results). These were allocated with 1 GB of RAM and a single virtual CPU core. They communicated through a local network bound to the host machine. One of the machines ran the TPC-W Server, the other the MySQL database, and the third the TPC-W load-generator. In this third machine, a custom-built data gathering tool was placed, sampling data at the rate of 100 samples per second. 49 variables were sampled, including CPU load, open TCP connections, and number of running processes.

After recording several *golden-runs* (executions with no failure), as well as several executions with failure, the final data was passed to a pre-processing window-based algorithm (the dataset was filtered to contain a balance between both kinds of runs). This algorithm involved two stages: (i) *time-window construction* and (ii) *window-merge*. In the first, fixed time windows of size T, starting at time $t = 0$ were created. For each of the 49 variables, samples within the same window were condensed using the mean, standard deviation, maximum, minimum, and derivative (rate of change). Thus, each window contained 245 variables (5×49). The second step, *window-merge*, involved concatenating N of the previously generated time-windows, appending them with a binary variable indicating if a crash was recorded within the next group of merged windows. In practice, each group of merged windows offers a *time-to-failure* prediction of $N \times T$. The final windows were then processed within *Weka*[3] as a classification problem. Empirically, we chose $T = 5$ s and $N = 2$, meaning a *time-to-prediction* of 10 seconds. The experiments were done for a TPC-W simulation of a "slow" and "fast" ramp up of users. This was done to assess if classifiers trained with one set of data could still be accurate on different data within similar circumstances.

The results were promising – with 10-fold cross-validation, a Naive Bayes classifier only failed to predict failures 1.52% of the time, and inaccurately predicted a failure 8.89% of the time. When this classifier was applied to data from the "faster" ramp of TPC-W clients, the false negative rate was of 23.3%, whereas the false positive rate was 7.01%. These results, although very preliminary, show that prediction of exceptions is within our reach, and that such a model can be useful for programmers – in this scenario, a time-to-prediction window of 10 seconds would allow the TPC-W clients to reduce their rate of requests, thus preventing or delaying the exception.

[3] `http://www.cs.waikato.ac.nz/ml/weka/`

5 Conclusions and Future Work

In this paper, we have proposed a new Exception Handling model that defies nowadays' Exception Handling preconceptions. Current research in exception handling and online failure prediction shows that a fine-grained system for predicting exceptions is currently missing. Instead of catching exceptions, this model proposes that the system, as a whole, actively work towards predicting and preventing exceptions. Applications can then be more resilient, robust, reliable and have increased performance. Our preliminary results also show that it is possible to predict exceptions, and that a paradigm shift towards prevention, rather then reaction, is quite within our reach. As future work, we intend to develop a proof of concept implementation of PreX in a modern programming language.

References

1. Goodenough, J.B.: Exception handling: issues and a proposed notation. Communications of the ACM **18**(12) (1975) 683–696
2. Fonseca, A., Cabral, B.: Handling exceptions in programs with hidden concurrency: New challenges for old solutions. In: Exception Handling (WEH), 2012 5th International Workshop on, IEEE (2012) 14–17
3. Cabral, B., Marques, P.: Exception handling: A field study in java and. net. In: ECOOP 2007–Object-Oriented Programming. Springer (2007) 151–175
4. Issarny, V.: An Exception Handling Mechanism for Parallel Object-oriented Programming: Towards the Design of Reusable and Robust Distributed Software. Inst. National de Recherche en Informatique et en Automatique (1992)
5. Yemini, S., Berry, D.M.: A modular verifiable exception handling mechanism. ACM Trans. on Programming Languages and Systems (TOPLAS) **7**(2) (1985) 214–243
6. Cabral, B.: A Transactional Model for Automatic Exception Handling. PhD thesis, Universidade de Coimbra (2009)
7. Salfner, F., Lenk, M., Malek, M.: A survey of online failure prediction methods. ACM Computing Surveys (CSUR) **42**(3) (2010) 10
8. Liang, Y., Zhang, Y., Jette, M., Sivasubramaniam, A., Sahoo, R.: Bluegene/l failure analysis and prediction models. In: International Conference on Dependable Systems and Networks, IEEE (2006) 425–434
9. Cheng, F.T., Wu, S.L., Tsai, P.Y., Chung, Y.T., Yang, H.C.: Application cluster service scheme for near-zero-downtime services. In: Proc. of the 2005 IEEE International Conference on Robotics and Automation, IEEE (2005) 4062–4067
10. Vilalta, R., Ma, S.: Predicting rare events in temporal domains. In: Proc. of the IEEE Int. Conf. on Data Mining (ICDM), IEEE (2002) 474–481
11. Magalhaes, J.P., Silva, L.M.: Showa: a self-healing framework for web-based applications. ACM Trans. on Autonomous and Adaptive Systems **10**(1) (2015)
12. Chen, X., Lu, C.D., Pattabiraman, K.: Failure prediction of jobs in compute clouds: A google cluster case study. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE (2014) 341–346
13. Irrera, I., Duraes, J., Madeira, H., Vieira, M.: Assessing the impact of virtualization on the generation of failure prediction data. In: Sixth Latin-American Symposium on Dependable Computing (LADC), IEEE (2013) 92–97

Appendix D

Submission: PreX: A Predictive Model to Prevent Exceptions

PreX: A Predictive Model to Prevent Exceptions

João Ricardo Lourenço[a], Bruno Cabral[b], Jorge Bernardino[c]

[a]*joaoml@student.dei.uc.pt*
[b]*bcabral@dei.uc.pt*
[c]*jorge@isec.pt*

Abstract

The exception handling mechanism has been one of the most used reliability tools in programming languages in the last decades. However, this model has not changed much with time, in spite of advances in programming languages, which include concurrent programming and a shift towards more reactive paradigms, the basic principle remains the same - an exception occurs, and the mechanism reacts. We propose a new paradigm, inspired by Online Failure Prediction (OFP), to predict exceptions and possibly avert them by triggering the execution of preventive actions. The proposed model - PreX - is, thus, proactive, operating in a much finer-grained level than any other form of online failure prediction. OFP has shown promising results in predicting failures at a higher level, but has never been available to the developer, being mainly a system level technique. Thus, PreX will offer developers a new range of revitalization strategies. In this work, we describe the model and evaluate its performance by applying it to a real e-commerce solution, demonstrating how it is capable of predicting and preventing exceptions at run-time. Furthermore, we also show that PreX increases the overall availability and performance of the system under the same conditions.

Keywords:
exception handling, proactive, failure prediction, self-healing, preventive, predictive

1. Introduction

The Exception Handling (EH) mechanism [1] has stood as one of the most used reliability tools in programming languages since the decade of 1980. Most programming languages, such as Java, Python and C++, provide some form of "try-catch" model for Exception Handling. Modern advances in exception handling models, in particular due to multi-core platforms and new programming languages (e.g. Scala, Erlang and Elixir) still rely on the base assumption of classical models – an error occurs and a handling routine can only react to it after it has happened.

However, the ubiquity of the Exception Handling mechanism for error recovery does not imply its most correct or desirable usage [2]. Most of the time, developers use Exception Handling language constructs as a way of hiding problems, performing log activities or informing the user of unexpected behaviour, rather than recovering from it autonomously [3]. This kind of use of Exception Handling might be considered a symptom of a design flaw in the mechanism – the system only acts when it is too late, thus making the problem unavoidable [3].

Recent work in the field of Online Failure Prediction (OFP) [4, 5, 6] has shown that several techniques can be successfully used to predict failures. Online Failure Prediction mechanisms act in run-time, providing warnings and estimating failure probabilities according to the characteristics of the running system. Even if these systems have seen some success, they only provide a high-level approach to failure prediction, without fine-grained control at the source code level [4]. There has been some work [5, 7] on predicting failures of individual applications and components, but these are too generic and only able of predicting task completion status (success/error) [5] or generic component failures [7] (e.g. memory failures). There is no lower-level Online Failure Prediction mechanism that can notify programs and applications of relevant, specific, and potential failures, such as a database timeout, instead of merely predicting the overall failure of the system or the exit status of applications.

Online Failure Prediction mechanisms can be applied at a lower-level, if integrated with the Exception Handling mechanism, and provide tools for programmers to act proactively in the face of potential exceptions,

leading to an overall increase in software reliability and robustness. For instance, by detecting that a database timeout is imminent due to excessive workload, an application can adjust its workload, preventing the failure instead of failing and only afterwards triggering the failure handling strategy, thus reducing overall downtime and increasing robustness, reliability and performance.

To further illustrate the motivation for this new model, consider a system of at least a database and several client applications. Consider also that these client applications are write-heavy, meaning they processes several thousands of write operations per second, sending them to the database. Due to the heavy load, the database may slowly become unresponsive and ultimately trigger a *ConnectionTimeout* exception on one of the applications. That application will then have to attempt to reconnect, and restart where it was previously, if such is really possible. This shows the aforementioned downfall of the conventional Exception Handling mechanism – the system only reacts to exceptions, it does not avoid them. The motivation for this work stems from this issue – we believe that the client applications would benefit from a prediction (i.e. a warning) that the database may trigger a *ConnectionTimeout* exception. With such a warning, they can, for instance, proactively slow their execution rate and prevent the exception from happening. Ultimately, slowing execution down could prove to be more efficient than triggering the exception and restarting the whole process. The new model, then, could improve reliability at a fraction of the cost of other approaches, such as vertical or horizontal scaling techniques.

The new model reshapes the concept of "try-catch" blocks into "try-prevent-catch" blocks, allowing programmers to write handling routines which will be executed when an exception is predicted to happen. These routines can contain programmer-written preventive code, meant to prevent the exception, thus increasing the overall reliability and availability of systems. The preventive block is a shift from a reactive paradigm – where it might be too late to react to the error – to a proactive paradigm – where the program proactively prevents exceptions and consequently might increase performance.

Our experiments validate PreX on a real enterprise e-commerce solution – the open-source Shopizer application – showing that the use of this model can increase system reliability and availability at a fraction of the cost of other approaches. The results are extremely promising, showing decreases in the number of exceptions by two orders of magnitude, leading to several golden-runs where no exception happens, accompanied by statistically significant ($p < 0.01$) increases in successful operation throughput.

The remainder of this article is structured as follows. Section 2 presents the state of the art in Online Failure Prediction and the characteristics of current exception handling models. Section 3 details the new preventive model and its architecture. Section 4 presents the experimental setup for the first the validation of this model. The results are presented and discussed, as well as future work possibilities, in Section 5. Finally, Section 6 presents the conclusions.

2. State of The Art

This section presents the state of the art in two distinct areas of research – exception handling models and online failure prediction. In addition, it discusses related work in the field of self-healing systems. A brief summary is presented at the end of the section.

2.1. Exception Handling Models

Yemini et al. [8] identified four distinct Exception Handling Models, summarized in [9]:

- **Resumption model** – When an exception is raised, the control flow is transferred from the raise point to the handler and, after the exception has been handled, it is transferred back to the raise point. This model effectively binds the caller and the callee together, and is prone to recursive resumption, thus being difficult to implement [9].

- **Termination model** – An exception is raised within a protected block, with the control flow transferred to the handler, terminating any intervening blocks. The control flow then resumes as if the protected block has terminated without any errors. This is the most widespread model in use [9].

- **Retrying model** – The signaller is invoked after the completion of some operation. This model is appropriate to transient faults, where retrying the invocation might lead to no exceptions. The main disadvantages of this model are its inherent implications for non-idempotent operations, counters, etc – the programmer must be wary of how the code executes.

- **Nonlocal transfer** – The program flow can be transferred to any other location in the program. This model has the obvious drawback of being hard to maintain and much more error-prone [9].

Modern advances in Exception Handling, such as concurrent exception handling or alternative models in the context of the actor model and Functional Reactive Programming, still share a common characteristic with these classical models – an exception is raised, and only then can corrective actions happen. This, unfortunately, means that many incorrect uses of Exception Handling Mechanisms arise, with programmers focusing on hiding errors, rather than attempting to fix them [3].

To the best of our knowledge, our model is the first proposal for a preventive exception handling model which shifts the current practice and allows programmers to act before problems arise. However, Kim et al. [10] propose a proactive approach to exception handling within a business process. In their work, the authors note that business processes often involve "human exception handlers" that react to "exceptions". They conclude that there is a need for a proactive exception handling which allows for action as soon as a business process exception is "predicted", and that this action can be specified by an external agent (e.g. a system administrator) as a reaction to a prediction. Kim et al.'s work is, thus, similar to the work presented in this article, but differs significantly in the following key points:

- It concerns exceptions in a business process management context.
- It does not *learn* how to predict business process exceptions. Instead, a system administrator can build a set of rules similar to "$X > Y$". Thus, the prediction code is a set of rules determined from experience, only dependent on business conditions (i.e. this work cannot "predict" a database timeout).

Nevertheless, the core concept of PreX is the same as the concept seen in the work by Kim et al.: a shift from reactive exception handling is needed, in favour of more proactive behaviour. PreX focuses programming languages, and on systems which can use online failure prediction mechanisms to "learn" exceptions independently of human interaction.

2.2. Online Failure Prediction

In this section, an overview of current Online Failure Prediction methods is given, after some basic definitions have been presented. In this section, we start by presenting the definitions used throughout this work. Afterwards, we briefly describe the are of Online Failure Prediction and present several proposed and tested methods.

2.2.1. Definitions

Throughout this work the following definitions are used [11, 4]:

- **Failure**: A service failure, often abbreviated as **failure**, is the event that happens when a service deviates from correct operation. This may happen because it does not comply with its functional specification or because the original specification does not adequately describe system function. Salfner et al. [4] note that a failure refers to "misbehavior that can be observed by the user, which can be a human or another computer system". In that sense, although "things might go wrong" in a system, it does not constitute a failure insofar as there is no deviation from correct service.
- **Error**: An error corresponds to the part of the total system state that may lead to a subsequent service failure. Many errors never reach the system's external state, hence do not cause a failure. Avižienis et al. [11] further distinguish between **undetected errors** and **detected errors**: An error can remain unidentified until a detector identifies the incorrect state.
- **Fault**: A **fault** is the "hypothesized cause of an error". In other words, it is its root cause. Often, faults remain dormant until they are activated, leading to incorrect system state (an error) and, ultimately, might lead to a failure.
- **Symptom**: Errors might cause failures, but they might also cause "out-of-norm" behavior as a side-effect. For example, the system might operate with expected results but do so with increased CPU usage or memory consumption. This behaviour is called a **symptom** [4].

Online Failure Prediction can be summarized in Figure 1. A failure is to be predicted with some given **lead-time** Δt_l, at current time t, based on the current system state measured within a data window of length Δt_d (which we call **data validity time**). This prediction (e.g. fail/no fail, failure probability, etc) is valid during a window Δt_p, called the **prediction period**.

A failure might not always be predicted in time. For instance, if we predict an "out-of-memory" error within 2 seconds, but need 5 seconds to fully slow down our process and prevent the error, then the prediction *lead-time* will have been useless. Thus, there is a **minimum warning time**, Δt_w, needed for a system to react proactively to the failure prediction.

The prediction period, Δt_p, is critical in online prediction. If a value is too low, the prediction is prone to fail many times, so higher values always increase the amount of correct predictions. However, very large windows make the prediction useless – it is not useful to know that a system is going to fail somewhere between today and five years.

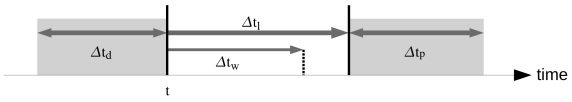

Figure 1: Time relations in online failure prediction (Salfner et al. [4]).

2.2.2. Prediction Methods

Salfner et al. [4] identified four distinct major branches of methods in Online Failure Prediction: (i) Failure Tracking, (ii) Symptom Monitoring, (iii) Detected Error Reporting, (iv) Undetected Error Auditing.

Failure Tracking methods use past failures to predict the future. Approaches to prediction can range from temporal correlation between failures to simple histograms of failure times. Csenki [12] uses a Bayesian predictive approach to improve the prediction of time-to-failure in an offline software reliability prediction context. Some methods can be used in failure prediction, although they are mostly used for root cause analysis. Such is the case with the work of Liang et al. [7], who explored temporal and spatial correlation to successfully predict hardware component failures in IBM's BlueGene/L. Fu and Xu [13] use a similar approach, coupled with a neural network, to predict the number of hardware failures with 81.6% accuracy and software failures with 72.9% accuracy for the LANL HPC system. These results are above average for online failure prediction, where, as shown in this section, most results tend towards 50% accuracy.

Symptom Monitoring methods periodically analyze samples of system state variables, such as the amount of free memory or page swaps, to estimate imminent failures. A key concept of these methods is the idea of *service degradation* – the system may start to exhibit some form of degradation before the actual failure happens (e.g. before a database time-out, successive requests might take a longer time to process). This degradation can be observed in system *side-effects*, such as longer response times, high CPU load or high memory consumption (e.g. in the case of memory leaks). These side-effects are called the *symptoms* of the failure.

Li et al. [14] present a stochastic approach where an auto-regressive model with auxiliary input is used to predict resource exhaustion times. Their method was shown to be computationally less expensive and have better results than Castelli et al.'s method [15] on their dataset. Andrzejak and Silva [16] employ a regression based approach to model the performance of an Apache Axis SOAP server. Their goal was to optimize software rejuvenation times, something which they were able to achieve. Regarding Machine Learning approaches, Neville [17] described how standard neural networks can be used for failure prediction in large scale engineering plants. Hoffmann [18] used an approach based on Universal Basis functions (UBF) to predict failures of a telecommunication system. In a followup work [19], the authors predict resource consumption of the Apache webserver using different modelling techniques. Their UBF approach yielded the best results for free physical memory prediction, although Support Vector Machines were a better choice for server response times.

Other approaches use classification methods. An example is the work of Hamerly and Elkan [20], who perform hard disk drive failure prediction with two different bayesian methods. The first uses a mixture of naive Bayes submodules and the second is a naive Bayes classifier trained with Expectation-Maximization. This second method computes conditional probabilities for SMART (Self-Monitoring, Analysis, and Reporting Technology) values belonging to the failure/non-failure classification. Using a decision threshold of 0.005, their approach achieves a true positive rate of 0.33 with false warnings having a probability of 0.0036. Of all failures, they predict 56% of them with a false positive rate of 0.0082 and a class threshold t of 0.001, outperforming industry standard methods and other classification approaches (such as [21]). Turnbull and Alldrin [22] use Radial Basis Function networks to classify data windows as failure/non-failure. These data windows contain monitoring values of hardware sensors (e.g. temperature and voltage). They achieve a 0.87 true positive rate and 0.10 false positive rate on a balanced dataset. Irrera et al. [23] presented a sliding window approach to incorporate the time dimension in a classification failure prediction problem, showing that this dimension improves results, but that it makes the problem harder as larger windows are used. There have also been some symptom monitoring methods which use time-series analysis. For example, Garg et al. [24] present a three-step approach to predict resource exhaustion. Their approach smooths the time-series and detects trends using a seasonal Kendall test (the method cannot be applied

otherwise), finally applying a non-parametric procedure for prediction.

In **Detected Error Reporting** methods, past error logs are used to predict future failures. This approach differs significantly from Symptom based approaches (which use system variables) in two ways: (i) events, unlike system variables, are not continuously spaced/-monitored, and thus require an *event-oriented* approach; (ii) system variables are usually real-valued, whereas event logs usually contain discrete information such as a timestamp or an ID. This branch of methods, then, carries a data window (coinciding with the *data validity time*) whose past events are taken into account when predicting the future.

Vilalta et al. [25] propose the eventset method, using a data mining approach (a rule-based model) based on the type of reported error (the time dimension is discarded). Under specific conditions, they have a false negative rate of only 0.16, although this value can be as extreme as 0.83 under other system conditions (the false positive rate is always lower than 0.1). Lin and Siewiorek [26] present a set of heuristic rules on the time of occurrence of consecutive error events to identify permanent failures. Their technique is mostly suited to hardware failures, and achieves a true positive rate of 0.9375 and precision of 0.75. However, the method has little value if failures do not occur frequently. Several authors [27, 28, 29] use grouping methods to make the observation that temporal co-occurence should be used to predict upcoming failures in short timespans.

Salfner et al., who have proposed the classification used in this article, also present the only identified methods of Detected Error Reporting with Pattern Recognition. Their central idea is that of merging the time-based characteristics of Lin and Siewiorek's [26] method with the type-of-error characteristic of Vilalta et al.'s [25] method. In this sense, in [30], the authors present Similar Events Prediction (SEP), a semi-Markov chain model. Their method achieves a precision of 0.800, recall of 0.923 and F-Measure of 0.8571 on data from an industrial telecommunication system. In Salfner and Malek [31], the authors propose a hidden semi-Markov model to "add an additional level of flexibility". This method achieves precision of 0.852, recall of 0.657 and F-Measure of 0.7419. Both of these methods seem to perform better than other failure prediction methods, whose precision and recall tend towards 50-60%.

Lastly, in **Undetected Error Auditing**, methods look for errors and error states in data not currently used – as opposed to Detected Error Reporing. For example, a consistency check on the filesystem might help predict failures in the future – if some files are found to be inconsistent, we can predict that accesses to those files will trigger a failure. Salfner et al. did not find any examples of such systems in their literature review, and this still seems to be the case.

For a full survey regarding Online Failure Prediction methods, interested readers should look at the aforementioned survey [4].

2.3. Self-Healing Systems

The previous sections have shown how Online Failure Prediction can be successfully used to predict failures, in spite of challenges regarding, for instance, data sampling and feature selection [32, 33, 34]. However, little work has been done for predictions at a more fine-grained level. Predictions are usually made at the system level, at most predicting a generic "crash" of some component. Thus, while promising, these techniques have no practical use for developers who wish to provide specific counter-measures when faced with the possibility of an exception.

There has been some related work done in the field of self-healing systems. Magalhães and Silva [35] propose a general self-healing proactive framework for web-based applications. Their work introduces a general framework to create self-healing transactional web-based systems. The framework, although operating at a lower level than traditional Online Failure Prediction methods, does not support run-time notifications at code-level nor operate at the fine-grained level that might be desired for applying more efficient preventive measures. Psaier et al. [36] propose a similar framework for mixed interactions between humans and Software-Based Services (SBS). Schneider et al. [37] present a recent and thorough survey of self-healing systems and frameworks. Their survey shows that self-healing systems are becoming more autonomous, although this is partly attributed to more specialization. For example, different approaches are used for mobile and centralised computing environments. The most successful systems are also those that use heavily supervised methods, thus relying strongly on human interaction. This interaction can be in the form of failure detection, as well as in manual insertion of recovery and healing strategies. Some methods employ evolutionary programming techniques to dynamically build new recovery solutions, such as the Plato framework proposed by Ramirez et al. [38]. Self-healing systems, thus, are still heavily dependent on predefined assumptions. For example, in the case of the Plato framework, crossover and mutation operators must be defined (the authors give an example of dynamic remote data mirror reconfiguration). Furthermore, these systems do not

allow application developers to leverage information regarding system failure. It is usually the job of a system administrator to define recovery actions, and no preventive actions are taken.

2.4. Summary

There is a broad class of Online Failure Prediction methods which has been shown to produce good results in practical scenarios. Many of these methods are based on machine learning algorithms. Thus, some of the major challenges in failure prediction are associated with machine learning challenges – for each problem, there is a particular set of features deemed ideal, but it is not a trivial task to find this ideal set. Furthermore, failure prediction in the context of dynamic environments requires alternative methods which support re-training or can act on-demand with new data.

The Exception Handling mechanism has not seen much change throughout the years. Although there have been several alternative models proposed, in particular due to an increase in popularity of alternative programming paradigms, such as functional reactive programming, the base principle of modern exception handling is still the same – the mechanism reacts to an exception, it cannot prevent it. Online Failure Prediction methods show potential as the base for a paradigm shift in Exception Handling mechanisms where exceptions are no longer caught, but can be predicted on a fine-grained level. This article presents PreX, a preventive exception model that proposes that the system, as a whole, should actively work towards predicting and preventing exceptions. Applications can then be more resilient, robust, reliable and have increased performance.

3. PreX – A predictive model for exception handling

PreX is an Exception Prevention model that focuses on preventing exceptions rather than catching them. The central idea was depicted with the example given in Section 1: it could be more efficient to temporarily reduce the throughput of a write-heavy application than to catch a *ConnectionTimeout* exception and have to restart the process. It makes sense that there are other scenarios, similar to this, where systems and developers would benefit from an easy-to-use proactive model for Exception Handling.

Preventing exceptions implies predicting them. To this end, the area of Online Failure Prediction provides valuable insight. There have been successful failure prediction systems, but these operate on a broader level. In order to predict exceptions, the proposed model needs to adapt failure prediction techniques to a per-exception basis. Furthermore, since no two systems are alike, the prediction models will have to be trained for specific deployments. Thus, the PreX model comprises different phases, including training and prevention. In the following sections, each of these phases is detailed from the perspective of the syntax and semantics of the model, followed by an overall perspective of the architecture and the necessary components of PreX. Lastly, the behavior of the system is presented.

3.1. Basic Definitions

PreX introduces several new concepts in the context of exception handling. In current exception handling models, exceptions are raised, caught and handled. In PreX, exceptions can also be **predicted** with some degree of confidence within certain blocks of code. These blocks are called **prediction blocks**. Whenever an exception is predicted to occur within a prediction block, the application is notified of this prediction through an **alarm**, which is said to be **triggered**. Program flow is then interrupted and transferred to a **prevention block**, where some action can be taken, after which program execution continues as normal (similar to the way the resumption model works). Note that this alarm is not an exception, nor does it propagate like exceptions do in traditional systems, either static or dynamic. It is, instead, an event that alerts for a potential exception.

At times, it might be useful to temporarily delay the exception prevention mechanism. For example, a set of atomic operations should not be interrupted. Programmers can explicitly set these blocks, which are called **no-alarm blocks**.

In summary, PreX introduces the following new concepts:

- **Prediction Block** – The region of code where programmers wish to be notified of exception predictions (to which they want to react). This matches the *try* block seen in current exception handling mechanisms, which is why it is also called the **try block**.
- **Alarm** – The notification that an exception *might* happen (i.e. it has been **predicted**) within a prediction block. Alarms are **triggered** by the exception prediction mechanism.
- **Prevention Block** – The code block to which control is transferred after an alarm is triggered. These blocks can contain any code, but will usually try to prevent the exception from happening.

- **No-alarm Block** – A block of code (within a prediction block) where no alarm can be triggered, and where they are instead postponed. If an exception has been predicted during the execution of a no-alarm block, the corresponding alarm is only triggered once flow is outside of the aforementioned block. There can be any number of no-alarm blocks.

There are other concepts relating to PreX that will be introduced in the next sections. However, these basic definitions are essential to start understanding the system.

3.2. Syntax & Semantics

An example of the syntax of the model is shown in Figure 2. In this example, we use pseudo-code very similar to Java. In the following sections, we will describe the detailed syntax and semantics of PreX used in Figure 2. The code is described in Section 3.3, as it is used in Figure 2.

```
1  // Connect to database and build list of data to write
2  DBConnection c = connectToDB();
3  List<Write> writes = fetchWrites();
4
5  // Try to predict exceptions using the "write-exceptions"
6  // prediction context
7  try("write-exceptions") {                                  (a)
8      for (int i = 0; i < writes.size(); i++) {
9          // Feed writes left to the prediction system
10         sample("writes_left", writes.size()-i);            (b)
11
12         // Don't allow alarms to be triggered
13         no_alarm {                                         (c)
14             print("Writing data!");
15             // Write the data through the connection
16             c.write(writes.get(i));
17         }
18     }
19 } prevent ( ConnectionTimeout, info ) {                     (d)
20     // info has information about the prediction
21     // (e.g. lead time)
22
23     // Sleep for some time to try to prevent an exception
24     sleep(1000);
25 } catch ( ConnectionTimeout e ) {
26     // The exception could not be avoided...               (e)
27     error("Database timeout!");
28 }
```

Figure 2: Example of PreX using synchronous *try-catch-prevent*.

3.2.1. The try-prevent-catch and try-prevent_async-catch constructs

PreX provides semantics similar to traditional *try-catch* blocks, although two different constructs are added to the language. The first is the *try-prevent-catch* construct, as seen below:

Listing 1: The *try-prevent-catch* construct

```
1  try(<prediction_context>) {
2      // ... Prediction Block.
3      // Exceptions can be caught
4      // and alarms can be triggered
5  } prevent ( <exception_name>,
6              <information_object> ) {
7      // ... Prevention Block.
8      // Execution follows the resumption
9      // model.
10 } catch ( ... ) {
11     // ... Exception Handling code
12 }
```

The *try* block denotes the scope during which a program cares about alarms regarding some particular exception (i.e. it is the prediction block). If an alarm is not raised, and if the exception cannot be prevented, it is raised and caught by the code, and the traditional termination Exception Handling model is used, with program flow being transferred from within the *try* block to the *catch* block.

The second construct added by the model is similar, but uses the *prevent_async* keyword instead of the *prevent* keyword. When an alarm is triggered, it can transfer to the prevention block in two different ways, depending on the construct used:

- **Synchronously**: execution within the *try* block is suspended and flow is transferred to the *prevent* block. In normal circumstances, the execution is then transferred back to the previous code within the *try* block (resumption model).

- **Asynchronously**: execution within the *try* block continues normally, and the *prevent_async* block is executed asynchronously.

Thus, in PreX, triggering an alarm does not necessarily terminate the execution of code within the *try* block (as opposed to the *termination* model when an exception is raised). Instead, execution is resumed as if there had not been any interruption, because preventing an exception should not halt normal execution of the current code.

When using a synchronous approach, alarms are not triggered at just any point in time within the *try* block. Within this block, every program statement will be executed without any interruption from the exception prediction mechanism (unless exceptions are raised). The flow of execution only moves to the prevention block between statements, without the need for propagation

in the call stack. In this sense, PreX uses an approach similar to static binding, meaning that alarms are not "propagated" as if they were exceptions. Instead, they are tied to the prevention block to which they belong. This approach makes sense if we consider that only the closest code, within the exact context of the particular exception, can know how to react to a specific prediction. Listing 2 shows an example of where alarms can be triggered (A, B, C, D, E and F) .

Listing 2: Alarms can only be triggered in between statements (A, B, C, D, E and F)

```
1 try(<prediction context>) {
2     function1(); A
3     c = a + b; B
4     if ( c > 0 && d < a ) {
5         C
6         function2(); D
7         function3(); E
8     } F
9 }
```

3.2.2. The no_alarm keyword

If developers wish to prevent the triggering of alarms during the execution of a set of statements, a special *no_alarm* keyword can be used to denote a new scope within which alarms are not possible – the no-alarm block. In Figure 2, lines 13-16 belong to one of these scopes. In the asynchronous approach, as in the synchronous approach, alarms are only triggered when the program flow is outside the no alarm block. Thus, the *no_alarm* keyword can act as a synchronization primitive between code in the prevention block and code within the *try* block.

Listing 3: The *no_alarm* keyword starts a no-alarm block where alarms can't be triggered

```
1 no_alarm {
2     // No-alarm block! No alarm can be
3     // triggered here.
4 }
```

3.2.3. The prediction context

The prediction system must be informed of which variables/features are relevant to each prediction block. It is not practical to use every available feature for prediction, because there might be too many features. Thus, there is a need to uniquely identify the different prediction blocks. In addition, note that the same kind of exception can happen in different circumstances, and that models trained for one circumstance might be of little use in other circumstances due to different system configurations. To solve both of these problems, a (string) argument to the *try* keyword can be supplied, called the *prediction context*.

Listing 4: The *try* block accepts a *prediction context* as its argument

```
1 try(<prediction_context>) {
2     ...
3 }
```

Blocks can then be uniquely identified by their *prediction context*. For example, a *ConnectionTimeout* exception in a write-heavy block can be distinguished from a *ConnectionTimeout* exception in a read-heavy block by using different arguments, such as *try("write-exceptions")* and *try("read-exceptions")*. Two completely different models for the same kind of exception (*ConnectionTimeout*) under different contexts and possibly using different features (later defined by the user) can then be trained. Several exceptions can be predicted within the same *prediction context* (and corresponding alarms triggered), and a *prediction context* binds training data and a prediction model to a unique name. To train the model, the system administrator may specify which *prediction context*s he/she wants the program to be trained in during a training phase. During this training phase, no prediction mechanism is used for those *prediction context*s (i.e. no alarms can be triggered), although exceptions can still be caught. If an exception is raised and caught, the training mechanism is notified to adapt its prediction models.

3.2.4. The sample keyword

PreX allows programmers to periodically sample variables that they think will be useful for prediction, in addition to system variables monitored with custom probes. For instance, the remaining number of operations left might be useful in predicting connection timeouts. These variables can be supplied to the prediction system at any time using the *sample (<variable_name>, <variable_value>)* construct.

Listing 5: The *sample* keyword can feed data into the prediction system within any prediction block

```
1 sample(<variable name>, <sample value>);
```

Note that if the *sample* keyword is used when in the training phase, it still feeds data to the prediction system.

3.2.5. The prevent block and the prediction information object

The prevent block is introduced with the *prevent* keyword. It requires two arguments:

- The first is the name of the exception to prevent, so that different prevent blocks be assigned to the same *try* block, similarly to how there can be different *catch* blocks in most programming languages.

- The second is the name to assign to the **prediction information object** within that prevent block. This object is created by the exception prediction mechanism and contains information regarding the alarm that has been triggered, such as the *lead time*, the *data validity time* or specific information regarding the prediction method chosen by the mechanism. This information might be used by application code to apply different prevention techniques.

In Listing 6, an example prevent block is shown for *TimeoutException* exceptions. The prediction information object can be accessed within the prevent code with the name *predInfo*.

Listing 6: Example prevent block, which requires the name/type of exception and a name to assign to the prediction information object

```
1  prevent ( TimeoutException, predInfo ) {
2      // Block for preventing
3      // TimeoutExceptions, when triggered
4      // by an alarm.
5      // predInfo contains prediction
6      // information such a lead-time.
7  }
```

3.2.6. Architecture

The PreX model's architecture can be applied to three different scenarios:

- **Training Scenario**: in which no prediction happens, only data is collected, and the models are trained for future usage (coinciding with the training phase).

- **Prediction Scenario**: in which a set of models has been trained and are used at run-time to predict failures.

- **Combined/On-line Scenario**: in which the previous scenarios might happen at the same time, providing a suitable system for dynamic environments where models must be retrained as conditions change.

Some components of PreX are shared among these scenarios, while others are only active in a particular scenario. The model's architecture relies on several different entities which cooperate to train models, make predictions and activate the *prevent block*. These entities can be distributed among the system, and do not need to exist in the same machine as the code with the *try block*. There are three main kinds of entities within a system using PreX:

- **Data gathering entities**: These entities feed periodic samples of data (e.g. memory and CPU usage) to the coordinator entities. Most of the prediction data comes from these entities, which don't execute any code that wants to be alerted of possible exceptions. The sampling rate for each of these variables is not pre-determined and may vary according to system load and characteristics.

- **Code entities**: Whenever a *try* block is entered, the exception prevention mechanism spawns a code entity that connects to the coordinator entity. Each code entity may want to register with the coordinator to be notified of predictions of certain kinds of exceptions within a *prediction context*. It is the responsibility of the code entity to trigger the alarm and transfer execution to the prevention block.

- **Coordinator entities**: A (potentially replicable) coordinator entity, responsible for aggregating the data from the other two types of entities and running the prediction system.

Each *try-prevent-catch* or *try-prevent_async-catch* block spawns a new code entity. The coordinator entity is responsible for running the failure prediction methods for predicting exceptions. This behavior is described in Section 3.3 in more depth.

The behaviour of the code and coordinator entities is different during the training phase. During this phase, the code entity registers that it will be sending data and information regarding a given *prediction context* and a given exception. The coordinator then uses this data to train the model.

The different entities can be freely distributed among machines, or might all run on a single machine, although this might have a performance impact on the overall system. A typical deployment scenario is depicted in Figure 3, where the Coordinator Entity is allocated to an individual machine.

Lastly, PreX is developed with the idea of extensibility. As such, an open protocol for interaction with the coordinator entity is specified so that other developers can build their own probes to feed data to the prediction system. Furthermore, this modularity means that if a new feature is ever needed, the data gathering entity does not need to be rebuilt and redeployed – instead, a

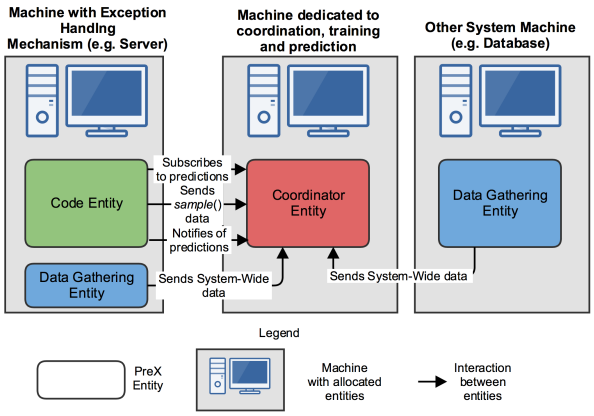

Figure 3: Example deployment of entities in a system.

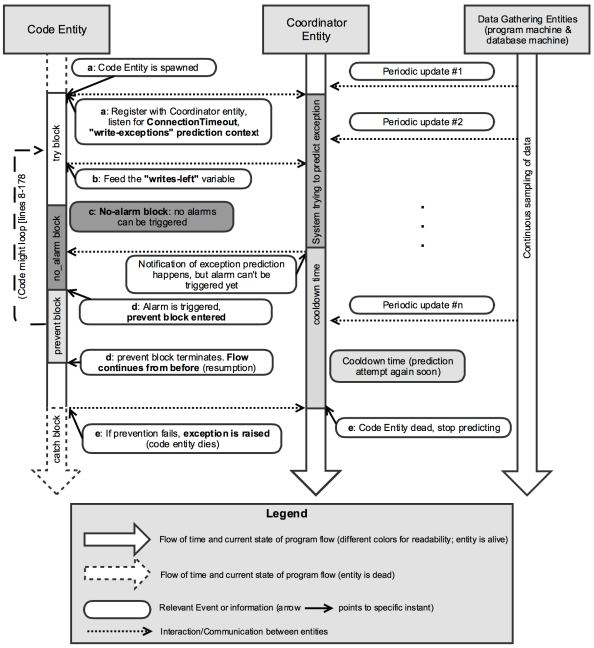

Figure 4: Interaction between code and entities through time for the code example in Figure 2. For simplicity, in this example, the loop only runs once.

new probe can be written, and data sent according to the open protocol.

3.3. Model Behaviour

To illustrate the behaviour of PreX, Figure 4 depicts the interaction between code, entities, and the prediction system for the code example in Figure 2 shown in Section 3.2. Notice that the code uses the synchronous prediction model. Consider also that if the code is executed during the training phase, no alarms will be triggered, so section *d* will not be entered.

The example Listing 2 allows understanding of how the overall system behaves when deployed. In the example, several data gathering entities are placed: one at the database machine and one more for each of the client machines (executing the code). Furthermore, a coordinator entity might be placed somewhere within the system.

When line 7 is reached, a code entity is spawned by the exception prediction mechanism, registering with the coordinator that it wants to be warned of *ConnectionTimeout* exceptions within the "write-exceptions" *prediction context*. The coordinator node begins predicting exceptions. At line 10, the *sample* keyword is used, making the code entity send a sample of data to the coordinator entity (which now uses this data together with the data from the data gathering entities). Normal operation continues at line 13.

Line 13 starts a *no_alarm* block, indicating that even if notifications of exception predictions are received, alarms cannot be triggered (thus guaranteeing that lines 14-16 are not interrupted). In Figure 4, an exception is predicted during execution of the *no_alarm* block. This information is sent to the *Code Entity*, which delays *triggering* the prediction alarm. Once at line 17, outside of the *no_alarm* block, the code is interrupted and execution continues in line 19, the prevent block (an alarm is *triggered*). For the sake of brevity in the example, the preventive handler executes a very simple prevention strategy. A sleep instruction is executed to momentarily reduce the workload on the server (thus preventing the exception) and execution continues exactly where it stopped (resumption model), at line 18. This sequence of steps can happen many times during execution.

In the case that the exception cannot be prevented, it may be raised at line 16 (the *write* method is the one that throws this exception), transferring program flow to line 25. The code entity terminates its execution at the end of the *try-prevent-catch* block, notifying the coordinator that it is no longer going to be interested in notifications.

Consider now that the prediction model has not been trained yet. To perform training, no changes to the code are required. A system administrator selects the "write-exceptions" *prediction context* for training and runs the application. In this case, reaching line 7 still spawns a code entity, but it is only to feed data regarding the "write-exceptions" *prediction context* and the *ConnectionTimeout* exception. At line 10, data is sent to the coordinator as normal. Line 13 has no effect, since exceptions are not being predicted, and the only way of

exiting the *try* block scope is to either successfully terminate the operation (i.e. a golden run) or by catching an exception. In both cases, the code entity notifies the coordinator and terminates itself afterwards.

3.4. Training and Prediction Methodology

PreX supports several prediction methods. Individual methods can be trained on the gathered data, and the most appropriate one can be selected. Nevertheless, PreX prediction methods have to operate on a basic set of features and follow a certain methodology. For instance, there are several differences in the way regression and classification methods are used. In this section we present the approach taken by PreX for prediction.

3.4.1. Features

As noted previously, Data Gathering Entities, as well as Code Entities, supply data to the prediction system. Each data point fed to the Coordinator Entity is characterized by a system-local timestamp and a value. The Coordinator Entity distinguishes features sent from different machines (the CPU Usage in machine A and the CPU Usage at machine B are different features, even if measured at the exact same time). Each Data Gathering entity will sample variables regarding CPU Usage, Network statistics, Disk usage/errors and Memory/Swap Usage. In total, this can amount to dozens of variables being sampled regularly (for example, 100ms). All data is numeric or binary. Since the PreX protocol will be open, any (authorized) probe will be able to provide additional features used for prediction.

In summary, the proposed model works in a distributed way, using any kind of features suitable for the specific exception and context of prediction. The default probes in PreX gather system variables regarding CPU, I/O, Network and other services. These are then used within the prediction method to make predictions.

3.4.2. Prediction Method

In Section 2.2.2, we presented the work of Salfner et al. [4], a classification of Online Failure Methods and survey of published work in OFP. PreX's prediction method will heavily influence the kind of algorithms and their performance, as well as the data preprocessing steps to be taken. As can be gathered from our chapter on the state of the art, there is no definitive method for online failure prediction – no approach has shown to be generally better than other approaches, and the application scenario heavily influences the methodology used. Prediction algorithms can be considered as a kind of optimization problems where a number of input varibles (i.e. "features") are used to model the state (failure/no-failure or a probability of failure) of the system. Depending on the type of failure and system, these input variables and their correlation with the external failure state of the program can vary greatly. In this sense, the "No Free Lunch Theorem" of Wolpert and Macready [39] shows that, indeed, no algorithm will be better than every other algorithm for all different kinds of predictions. Thus, if PreX intends to be broad, it must support several different prediction methods.

Different methods require different data. Some methods, such as the Eventset method proposed by Vilalta et al. [25], discard most temporal information regarding events. These methods also use Detected Error Reporting techniques. In the case of PreX, the only kind of error reporting is the exception itself, so these approaches are not appropriate. Additionally, we need to include the time dimension within our prediction system, since it is a critical part of the prediction process (Duetterich and Irrera et al. have shown the importance of the time dimension [40, 23]). Other methods (e.g. [41], [42] and [43]) use statistical tests to compare feature distributions with error-free states or error states. These statistical methods could be applied to the kind of data gathered by PreX, but they also make it harder to include the time-dimension for prediction. Additionally, these statistical methods often require more *a priori* knowledge of the system so that the statistical models can be built. Other approaches (e.g. [44], [14], and [16]) use regression and function approximation techniques. PreX could implement this approach, although it is more adequate to resource exhaustion scenarios and might not be appropriate for all kinds of exceptions that PreX intends to predict.

A failure prediction problem can also be modeled as a classification problem. These problems can be solved by state-of-the-art classifiers, and many authors have used classification methods for Online Failure Prediction (e.g. [20], [22], [21], [42] and [45]). Irrera et al. [23] presented a sliding window approach to incorporate the time dimension in a classification failure prediction problem. Their work, which applies techniques presented by Duetterich [40], is the inspiration for our approach – PreX treats failure prediction as a **classification problem**.

3.4.3. Feature Set Construction (prediction algorithm)

In this section we present the prediction approach used by PreX, which is a classification approach.

Data Gathering Entities gather data at arbitrary times (usually with some desired frequency, such as 100 ms or 1000 ms). This data is condensed/summarized in windows of size T, called **augmented feature windows**,

which are represented by **augmented feature vectors** $x_1, ..., x_n$. Window x_1 contains condensed/augmented feature information (in the current version of PreX, the *mean*, *standard deviation*, *maximum*, *minimum* and *derivative* of the data are taken) for the interval $[0; T[$. Window x_2 contains condensed/augmented feature information for the interval $[T; 2T[$ and so forth. If a failure/exception is recorded within the timespan of a window, that window is considered as a failure window. If there are m variables (e.g. CPU Usage, Memory Usage, etc), the augmented feature vector is of size $5 \times m$, since each variable produces 5 *augmented features*. This step, the **time-window construction**, is shown in Figure 5.

The augmented feature windows are then merged in the **window merging process** to incorporate the time dimension. A configurable parameter, $k \in \mathbb{N}$, specifies how many windows should me merged. The input vector for classification, w_i is formed by taking

$$w_i = (\underbrace{x_{i-(k-1),1}, ..., x_{i-(k-1),n}}_{x_{i-(k-1)}}, \underbrace{x_{i-(k-2),1}, ..., x_{i-(k-2),n}}_{x_{i-(k-2)}}, \underbrace{...}_{...}, \underbrace{x_{i,1}, ..., x_{i,n}}_{x_i})$$

and using a sliding window process for each of the original augmented feature windows. Each window w_i is then assigned a label, l_i, defined as

$$l_i = \begin{cases} 1, & \text{if there was an exception in the} \\ & t\text{-th next augmented feature window} \\ 0, & \text{otherwise} \end{cases}$$

where $t \in \mathbb{N}$ is another configurable parameter which controls the lead-time. This step is shown in Figure 6.

A classifier can then be trained to predict the labels l_i based on the windows w_i, ensuring a lead time $\Delta t_l = t \times T$ with a data validity time of $\Delta t_d = T \times k$ and a prediction period of $\Delta t_p = T$. PreX trains several different classifiers using this input data and chooses the best for each prediction context. Thus, the configurable data for feature set construction is given by the tuple

$$(T, k, t)$$

where T is the window-size, k is the number of windows to merge/concatenate and t is a "look-ahead" parameter to determine how much in advance a failure is to be predicted. Obvious trade-offs exist among these parameters, the classification performance and accuracy. In Section 5 part of these trade-offs are studied. Also note that for the same dataset, higher values for k reduce the number of instances available for classifier training at the cost of making each instance much larger in terms of features, potentially impacting accuracy.

The example shown in Figures 5 and 6 presents a representation of the feature set construction process with $k = 2$, $t = 1$ and 2 features.

It should be noted that data will not always be available (some windows might have no data at all regarding some features). Thus, a missing value imputation method should be used.

Finally, a form of feature selection should be used. As we have seen, after the time-window construction and window-merge steps, the number of features increases dramatically (by a factor of $k \times t$), and the classification problem might become unfeasible. Several strategies can be used for feature selection, such as the three-step feature selection process seen in [23]: (i) removal of null/constant features; (ii) using a classical linear correlation metric (such as the Pearson correlation coefficient); and (iii) a classical approach involving wrapper or filter methods. Alternatively, a strategy similar to the one proposed by Irrera et al. [34] can be used, where changes in features are compared between failure and failure-free executions of a program, based on the concept of a symptom.

4. Experimental Evaluation

In this section, we present the evaluation strategy. Unlike experiments in [46], these involved a prototype of the full model, using offline training, online monitoring, and methods to prevent exceptions. In addition, this work focuses on a real-world scenario, instead of an artificial benchmark. PreX can provide new revitalization strategies for developers, possibly enabling them to increase reliability at a lower cost when compared with other reliability techniques, such as replication. The aim of our experiments was to validate this hypothesis in a real world scenario

A modified instance of the open-source e-commerce Java software Shopizer [47] was configured, using a MariaDB database as the backend. A workload was configured using jMeter [48] to simulate Web users navigating through the virtual shop, eventually overloading the database. As a consequence, since queries were preconfigured to timeout in the database after a period of time, during the execution of the workload, several exceptions are raised. PreX was then used to predict and prevent as many of these exceptions as possible.

The goal of the experiments was to answer the following **research hypotheses**:

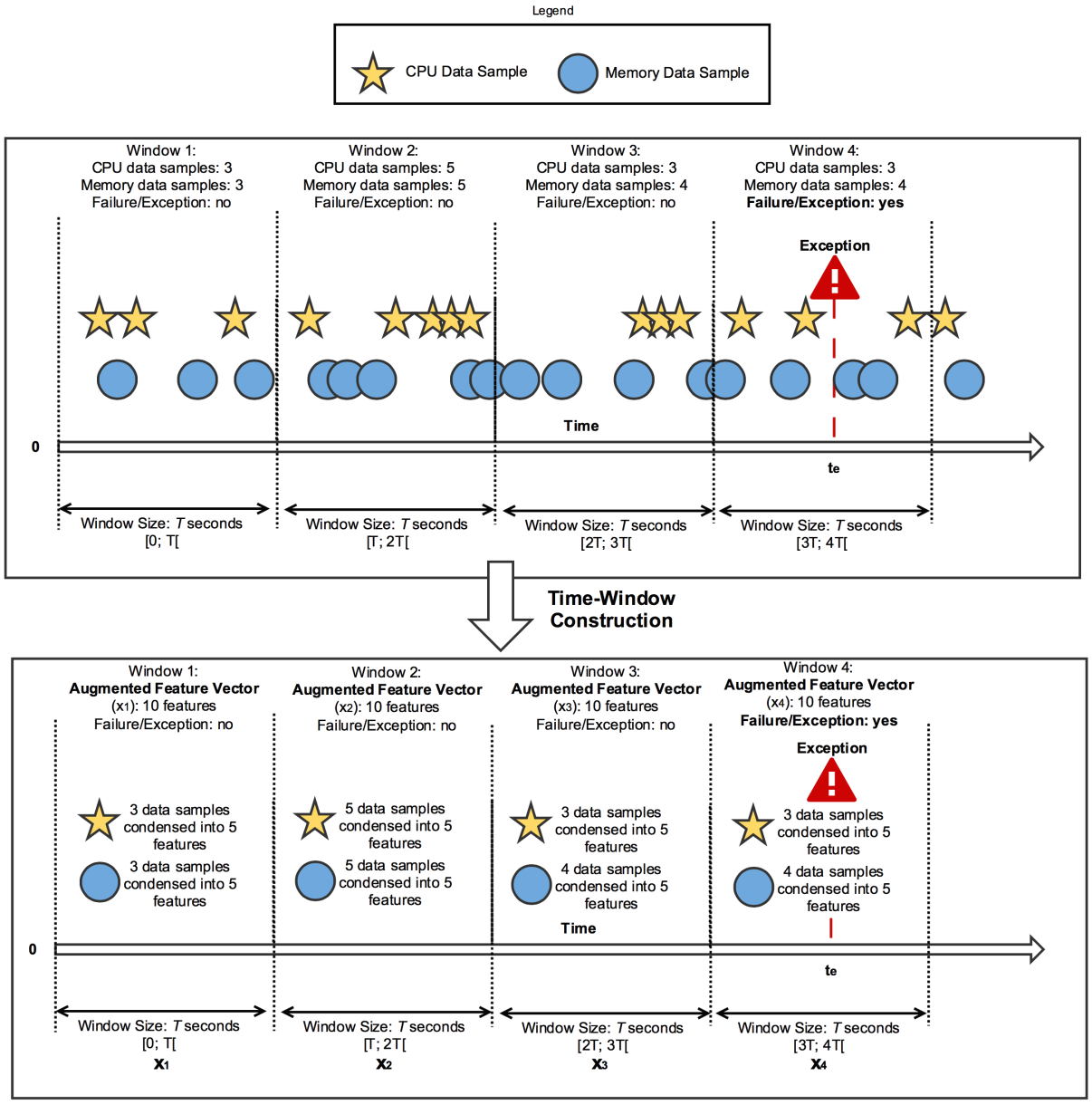

Figure 5: Time-Window Construction groups and summarizes variables according to “time windows” of size T.

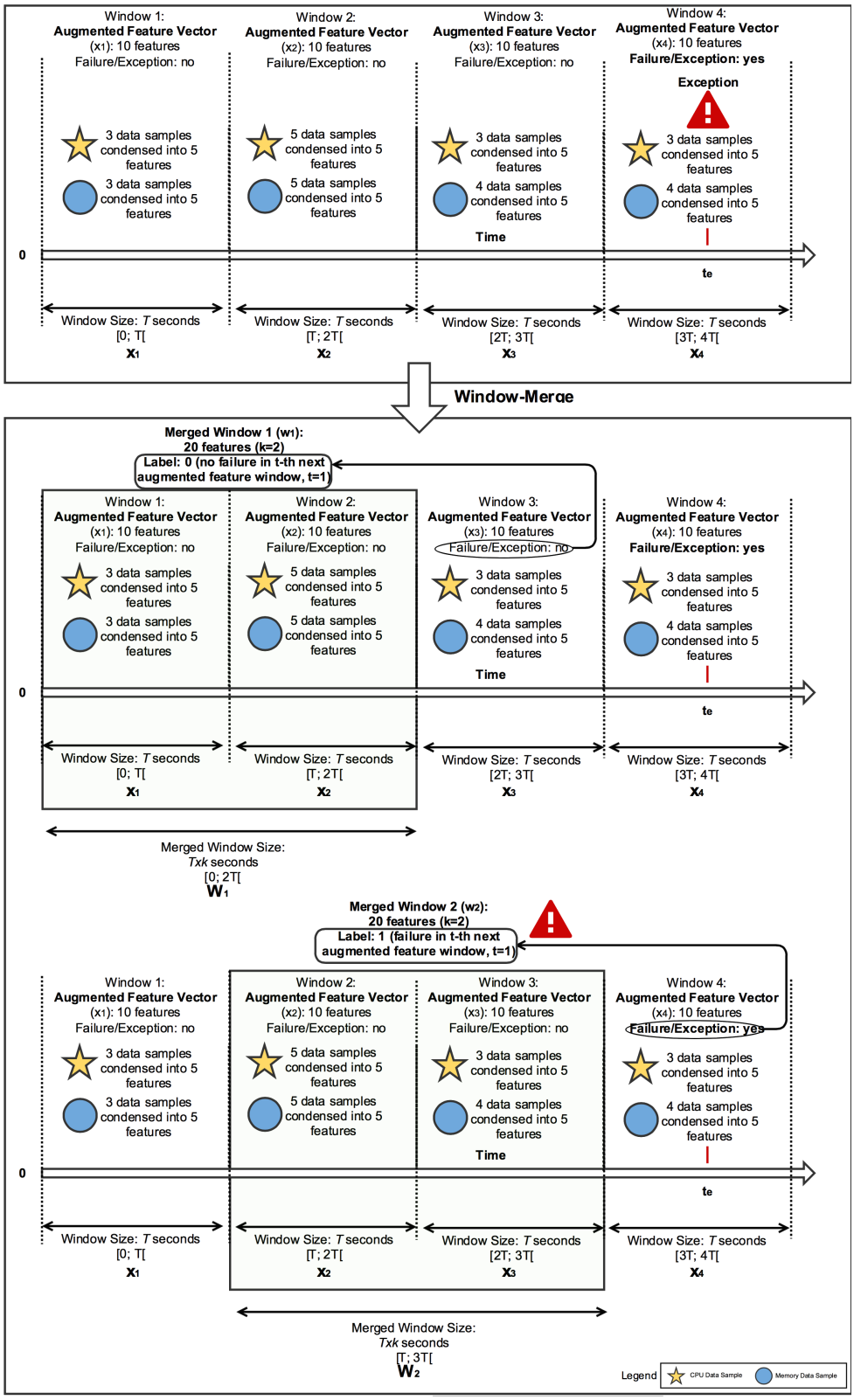

Figure 6: Window Merge merges augmented feature windows using a sliding window process. Each of the sliding windows, composed of merged augmented feature windows, is then passed to a classifier. In this example, $k = 2$ and $t = 1$.

1. **H1**: *PreX is able to predict exceptions with False Positive Rates and False Negative Rates below 20%.*
2. **H2**: *PreX is able to prevent exceptions in an effective way.*
3. **H3**: *PreX enables the usage of new and differentiated recovery strategies, not available to traditional EH models.*
4. **H4**: *It is possible to define a (T, k, t) parameter configuration that will deliver good performance.*
5. **H5**: *The exception prediction algorithms are sufficiently robust to withstand slight differences in environment conditions without affecting accuracy*

To answer question 1, an offline training of classifiers is sufficient. These classifiers can then be used at runtime to answer question 2. At the same time, traditional recovery strategies should be employed to be compared with the preventive mechanism and answer question 3. Finally, question 4 can be partially answered by exploring different parameters in the offline training. Lastly, to answer question 5, we can train models in different scenarios, with different workloads, and validate them on one another.

4.1. Experimental Setup

The setup consisted of 4 machines, simulating a 3-tiered architecture. The machines all had the same hardware and software, shown in Table 1, and were named *prex1*, *prex2*, *prex3* and *prex4*.

An instance of MariaDB 10 was configured in the *prex1* machine, to act as the data source for Shopizer. Shopizer itself was downloaded from the official repository and installed in *prex2*. By default, the application comes with a limited set of mock data (6 products). Since we wanted to create high workloads at the database layer, we modified Shopizer so that each product was inserted, with slightly different versions, 1000 times. Hence, in total, there were 60.000 products.

An instance of jMeter was installed in the prex3 and prex4 machines, so that they could be used to induce heavy loads on the other two machines, simulating the client layer. Figure 7 shows the overal experimental setup. Two Code Entities (probes), as defined by the architecture of PreX, were placed in the system: one in the database machine and another one in the server/shopizer machine. These entities sampled several system variables, including CPU, memory and disk usage, network and TCP related statistics.

Table 1: Summary of the machine specifications for the validation experiments

CPU + OS	RAM	HDD	Name
Intel Core i3 3.10GHz CentOS 7 (64 bit)	4GB	WD5000AAKX 7200 RPM 16MB Cache SATA 6.0GB/s	*prex1*
Intel Core i3 3.10GHz CentOS 7 (64 bit)	4GB	WD5000AAKX 7200 RPM 16MB Cache SATA 6.0GB/s	*prex2*
Intel Core i3 3.10GHz CentOS 7 (64 bit)	4GB	WD5000AAKX 7200 RPM 16MB Cache SATA 6.0GB/s	*prex3*
Intel Core i3 3.10GHz CentOS 7 (64 bit)	4GB	WD5000AAKX 7200 RPM 16MB Cache SATA 6.0GB/s	*prex4*

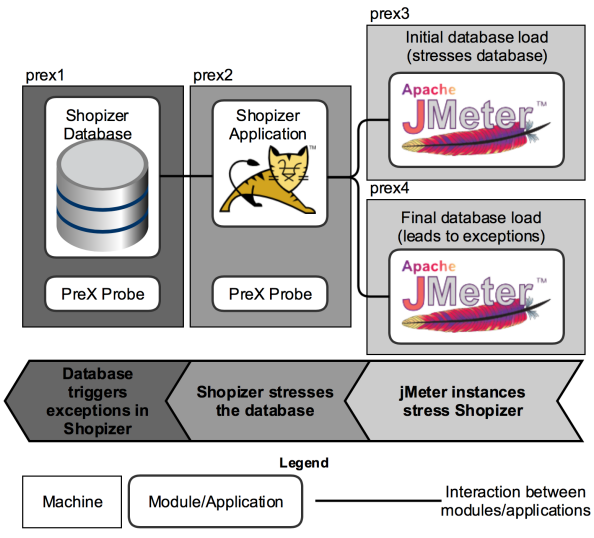

Figure 7: Experimental Setup.

4.2. Experimental Methodology

The experimental procedure consisted of two stages: training and preventing. The first stage involved running a series of tests where exceptions were triggered, using a simulation of several users navigating in Shopizer (with jMeter). After each test was completed, the machines were reset to its original state and the experiments repeated, so that enough data was available to accurately train several validation models. After these models had been trained, they were used in the second stage, repeating the previous steps, but using a method to try to prevent exceptions, detailed further below. Thus, the full model was tested, from prediction to prevention.

The default configurations for Shopizer allowed queries to take a very long time, making tests unpractical and unrealistic. To counter this situation, we configured MariaDB to timeout unfinished queries after 15 seconds. Under heavy workloads, this triggers exceptions in the Shopizer application, which were used for training.

The workload was generated by jMeter instances running in the remaining two machines (*prex3* and *prex4*). When each test started, the prex3 machine introduced a heavy workload simulating 128 clients clicking through the website at a random time after the test start (between 35 - 50 seconds). This number was just enough to strain resources on the database machine, making its four cores reach 99-100% usage, but still without raising exceptions. At a random time (ranging between 1 to 2 minutes to let the original workload stabilize), the *prex4* machine introduced an additional workload, using a ramp-up time, with the intent of pushing the database beyond its limits and raising exceptions. The random time makes the scenario more realistic and less predictable, simulating a sudden burst of interest in the website, as is usually observable in Christmas and Black Friday.

The aforementioned workload introduced by the *prex4* machine was not always the same. Three different workloads were selected so that different scenarios could be simulated. The goal of the experiment was to validate the accuracy of models trained on one workload and being used to predict exceptions in another. The three different workloads are shown in Table 2. After 3-5 minutes, the test was stopped and then the procedure was redone until all tests had finished.

For each stage (training and preventing), each test was run 15 times, which totals 45 full tests (15 for each of the three workloads presented), each containing data for about 10 minutes of execution, with dozens of exceptions recorded.

Table 2: Characteristics of the three different workloads

Workload/Scenario	Number of Clients	Ramp-Up time (seconds)	Notes
Heavy Load	256	100	Simulates quick high influx of clients
Medium Load	128	100	Simulates quick moderate influx of clients
Heavy Load with long ramp-up	256	400	Simulates slow high influx of clients

Besides a training phase, the predictive exception mechanism also involves a prevention phase wherein the developer might specify code to run in case of an alarm. In our scenario, the e-commerce website was flooded with too many requests (see Table 2), but not all of them are of the same value for the company. For example, updates and inserts are usually associated with transactions and important business operations, whereas read operations are more geared towards navigation of the online catalog (where most of the time is spent). Therefore, it makes sense that, upon detecting a possible exception, indicating a heavy workload, a mechanism should prioritize inserts and updates over reads. This way, the important business operations can take precedence, increasing the monetary gain of the company during an overload of clients. This was the mechanism implemented in our experiments.

Upon detecting that an exception was imminent, the modified Shopizer application waited a random interval (depending on the lead-time) before executing the lengthy read operations that lead to exceptions. This has several consequences:

- The insert and update queries are executed without any delay, thus being prioritized and increasing the company's monetary gain.
- The heavy read operations were scattered across a random uniform interval, effectively reducing the load on the database (throttling the rate of requests).
- The shopizer application, which uses a threadpool, would not be stuck waiting for the lengthy read operations to finish. Instead, it could process other, smaller operations, effectively increasing the overall throughput of the application.
- Due to the reduced load on the database, exceptions might be prevented.

As described in Section 3.4.3, the prediction algorithm involves three parameters: T, k and t. For this experiment, we trained algorithms only for $t = 1$, since it is likely that windows closer to the exception contain more valuable information (it might be interesting to experiment with other values for t in future work). With

regards to T and k, the values are presented in Table 3. These first five pairs of values were selected to achieve a lead-time of 10 seconds, which are enough for a corrective action to take place. The last pairs were selected to try and achieve a lead-time of 15 seconds, which is the time it takes for the database to timeout the queries and trigger an exception, thus making it an interesting value to explore.

Table 3: All combinations of variables used in the experiments

T **(Time-Window size)**	k **(Number of merged windows)**	t **(Number of windows to look-ahead)**
1000 ms	10	1
2000 ms	5	1
2500 ms	4	1
5000 ms	2	1
7500 ms	2	1
15000 ms	1	1

After an initial offline analysis, detailed in Section 5, the best combination of values was used for real-time prediction and preventive action. In addition to that, since we are performing an initial analysis of the model, we chose to use a Decision Tree classifier (using the C4.5 algorithm, as provided by the J48 classifier in Weka), so as to be able to interpret the prediction rules.

4.3. Comparison Metrics

To compare the preventive exception model with the prior exception handling model, we modified Shopizer to retry the failed operations to a maximum of three times. This means that even if exceptions were caught, the operation itself might be successful on a following retry attempt. Thus, this recovery mechanism can be compared with the aforementioned prevention mechanism. This is summarized in Table 4

Table 4: Comparison of recovery and prevention strategies

Scenario	**Strategy**	**Notes**
Without PreX	Retry operation until a maximum of three attempts is reached.	Common recovery strategy
With PreX	Sleep for a random amount of time between 5-15 seconds when alarm (of exception) is triggered (for read queries).	Simple prevention strategy offered by PreX. Distributes load and prioritizes updates.

We introduced profiling code into Shopizer. As such, it was possible to register when heavy operations were started, when they ended, when an exception was caught and, at a later stage, weather the prevention technique was being used. This allowed us to collect the following metrics on each test:

- The count of successfully completed operations (even if they had to be retried)
- The count of exceptions
- The count of unsuccessful operations (i.e. operations which never finished, exceeding the limit of retries)
- Average count of successfully completed operations per second (successful operation throughput)

With these metrics, it is possible to assess the effectiveness of the proposed model. For example, it is possible to know if the model prevents exceptions at the cost of performance (i.e. throughput), or if it is inefficient (i.e. prevents few exceptions).

5. Evaluation

In this section we present the results of our experiments. We start by discussing the offline analysis of the several (T, k, t) parameter combinations. Afterwards, we present the results of the prevention mechanism, once the best of these combinations had been selected.

5.1. Classifier Performance (Offline Analysis)

Table 5 presents the results of the different (T, k, t) parameter combinations. FPR stands for False Positive Rate, indicating the percentage of "false warnings". FNR indicates the False Negative Rate, or the amount of "missed warnings". It is worse to have a high FNR than to have a high FPR, since "missed warnings" mean an exception was not predicted, possibly having consequences for business. In addition to performing 10-fold cross-validation for each of the workloads (heavy workload, medium workload, heavy workload with long ramp-up), some of the datasets were used for training and validating on other datasets. This allows for an assessment of the generality of the predictor (i.e. is a good predictor for the heavy workload an equally good predictor for the medium workload?)

Note that, as previously noted, only a decision tree algorithm was chosen because it allows for an interpretation of the prediction method. If this classfication approach showed poor results, we would have moved on to others.

In general, the best results were achieved with the $(7500, 2)$ and $(15000, 1)$ pairs. In other words, using windows of 7500 ms, grouped in pairs of two, or windows of 15000, without grouping, achieves the best prediction accuracy. This remains the case even when datasets are validated on one another (i.e. training with

Table 5: Offline Analysis classifier performance (DNF means Did Not Finish)

		Dataset													
		heavy load (10-fold CV)		**heavy load with long ramp-up (10-fold CV)**		**medium load (10-fold CV)**		**train heavy load validate heavy load with long ramp-up**		**train high-load validate medium load**		**train medium load validate heavy load**		**train heavy load with long ramp-up validate medium load**	
T	k	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR
1000	10	0.37	0.20	0.33	0.10	0.38	0.20	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF
2000	5	0.15	0.13	0.16	0.14	0.16	0.14	0.17	0.06	0.20	0.16	0.33	0.10	0.99	0.00
2500	4	0.10	0.11	0.20	0.02	0.10	0.11	0.09	0.06	0.11	0.13	0.01	0.01	1.00	0.01
5000	2	0.03	0.04	0.19	0.02	0.04	0.05	0.20	0.04	0.08	0.06	0.12	0.03	1.00	0.00
7500	2	0.01	0.02	0.16	0.01	0.01	0.03	0.01	0.04	0.01	0.03	0.09	0.02	0.16	0.00
10000	1	0.02	0.03	0.12	0.02	0.02	0.03	0.01	0.04	0.01	0.04	0.08	0.02	0.99	0.00
15000	1	0.03	0.05	0.02	0.03	0.02	0.03	0.02	0.03	0.03	0.03	0.06	0.06	0.00	0.00

one and validating on another). It is also clear that an increase in windows quickly makes the data too noisy (i.e. adds too many features and reduces the amount of information about each one), rendering poor results (these results are in line with those of Ivano et al. [23]).

It should be noted that when the heavy workload with long ramp-up dataset is used for training (and even simply using 10-fold cross validation), the results are bad (exceptions are almost never predicted). This is a consequence of less data on exceptions for this dataset. Since the ramp-up time is high, very few exceptions were recorded on the dataset, making it unbalanced and affecting the classifier performance. To counter this scenario, the dataset could be balanced, or an increased penalty could be added for a missed exception.

These results show that it is possible to predict these exceptions, and that predictors for different scenarios are accurate on each other's datasets. Since the best results were reached with the $(7500, 2)$ pair (excluding an outlier for the heavy load with long ramp-up time), the next phase of the experiments – prevention – was performed using this predictor.

The predictor itself, a Decision Tree classifier (using the C4.5 algorithm, as provided by the J48 classifier in Weka [49]) used a combination of variables from both machines, namely:

- TCP Passive Opens (from *prex2*)
- CPU Combined Usage (from *prex1* and *prex2*)
- CPU User Time (from *prex1*)
- Used Memory (from *prex1*)

The number of passive opens variable makes sense, as it increases when more clients connect. Similarly, the CPU usage easily allows one to understand in which situation the system is in. If the CPU usage in the database machine increases, consistently reaching 100%, then it is at its maximum load (and in particular, it might be spending much time doing user-mode processing in the database), possibly leading to an exception. In much the same way, if the CPU usage in the server machine decreases, it might indicate that there is a bottleneck in the database (the server would be waiting for the database to finish). When all of these variables are combined with the decision tree, then they allow for the results shown in Table 5. Keep in kind that no human had to create rules for triggering alert events, everything was machine generated from the data collected at run-time.

5.2. Prevention Mechanism Results (Online Analysis)

Table 6 shows the results of the prevention mechanism. Using the exception mechanism significantly decreased the number of exceptions (by factors between 40 and 100), with some increase (2.2%) in successful operation throughput for the Heavy Load and Medium Load scenario. In the last scenario, throughput decreased by 1%, although this scenario has far less exceptions and, thus, the mechanism is less useful and might add some overhead. These results show that the prediction mechanism worked, detecting exceptions as soon as the database machine started being overloaded with connections. Once these exceptions were being predicted, the prevention action was used with success.

Table 6: Comparison of scenarios with and without PreX

Scenario	**Successful operations (mean)**	**Unsuccessful operations (mean)**	**Exceptions (mean)**	**Successful operations per second (mean)**
Heavy Load	4294.6	1.134	207.34	12.65
Heavy Load with PreX	4832.27	0.0	3.8	12.93
Medium Load	4462.33	1.067	200.93	12.64
Medium Load with PreX	4507.4	0.0	2.93	12.91
Heavy Load with long ramp-up	4286.0	0.47	132.2	12.81
Heavy Load with long ramp-up with PreX	4566.73	0.0	1.2	12.62

The effects of the prevention mechanism can be seen in Figure 8, where two plots of different runs of the

experiment (for the heavy workload) are shown: one with the mechanism and one without it. Throughput starts by increasing, until it peaks (maximum capacity is reached). Afterwards, without the prevention mechanism, several exceptions are raised, although the throughput is roughly the same. When the prevention mechanism is used, there is a slight increase in throughput, and a significant decrease in the number of exceptions. For the Heavy Load scenario, exceptions are decreased by a factor of 54, a change from 207.34 to 3.8 average exceptions per test. For the Medium Load scenario, this factor is 69, a change from 200.93 to 2.93 average exceptions per test. Finally, in the last scenario, there is a decrease in exceptions by a factor of 110, a change from 132.2 to 1.2 average exceptions per test. Note that the prediction mechanism sometimes stopped predicting exceptions (possible false negatives), leading to an increase in operations completed successfully without the preventive action, but also leading to the few exceptions that still happen. This is an indication that if a better prediction model had been found, more exceptions might be prevented. Also note that there are no false negatives before the *prex4* workload, which leads to exceptions, begins.

The increase in successful operation throughput can be attributed to a more efficient resource usage. Since the workload is being distributed, and since less time is being spent retrying failed queries, then the overall throughput increases. In addition, note that these results do not reflect the throughput of other operations. For example, since read operations are effectively being "deprioritized", an increase in the throughput of write operations is expected. Thus, for this scenario, the use of the proposed exception prevention mechanism is a valid and valuable technique, with very little additional effort for the programmer (he/she only needs to code the recovery mechanism, train the model by running the system, and then enable the trained model).

5.3. Answer to Research Hypotheses

If we recall the research hypotheses presented previously (see Section 4), we can conclude that:

1. **H1**: the hypothesis is accepted. PreX is able to predict exceptions for our scenario with satisfactory accuracy, displaying False Positive Rates of no less than 15% and False Negative Rates of no less than 19%. The best results showed FPR and FNR in the 3-5% range.
2. **H2**: the hypothesis is accepted. PreX can be used to prevent exceptions in an effective way. The number of exceptions dramatically decreased in all three scenarios, by factors of 54, 68 and 110.
3. **H3**: the hypothesis is accepted. PreX offers new techniques that can compete with current revitalization strategies. Using the new model to prevent exceptions, even with a simplistic approach, the successful operation throughput increased about 2% in two of the three scenarios, when compared with a retry strategy. The increase in throughput is significant, with $p < .01$ using a dependent T-Test.
4. **H4**: the hypothesis is accepted. The values for (T, k, t) have a high impact on prediction performance. High values for k, coupled with low values for T lead to worse results. The $(7500, 2, 1)$ and $(15000, 1, 1)$ configurations provided the best performance.
5. **H5**: the hypothesis is accepted. The prediction models showed generality, since when applied to different workloads, performance was still adequate. More different scenarios should also be the focus of future work.

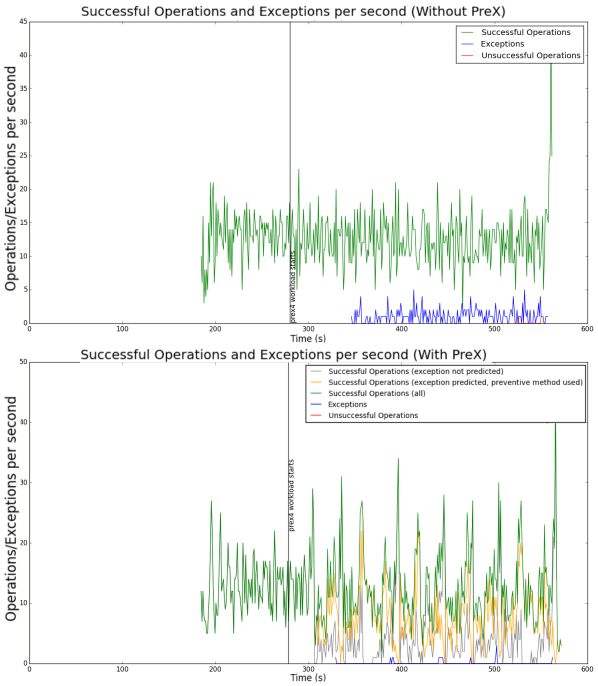

Figure 8: Operation and exception throughput with and without PreX.

5.3.1. Future Work

This model can be applied to other real-world scenarios to assert how effective it is outside of the experiments shown in this paper. In addition, a more thorough study can be conducted, evaluating the effort of

development with and without PreX. The experiments also focused heavily on validating the model, but not finding ideal parameters for our scenario or more general results. For example, is there a general pattern that can be used for all exceptions of a certain kind? Are some combinations of parameters (T, k, l) generally better than others? Are these parameters specific to certain scenarios? All of these questions raise future work possibilities. Lastly, some aspects of the model have not yet been tested, namely the *sample* keyword, the asynchronous variant of the model, and the online variant where new models are trained at run-time.

6. Conclusions

In this work we introduced and validated PreX, a new preventive exception model which changes the exception handling paradigm from reactive to proactive. Experiments show that PreX is a viable alternative to other revitalization strategies. Implementations of the model can be used to prevent exceptions, effectively increasing the reliability and availability of systems at a fraction of the cost of other, and, perhaps, more powerful alternatives, such as vertical or horizontal scaling. The recovery strategy used in this work is simple, but effective, leaving potential for exploration of other strategies.

The new model is focused on developers, enabling them to program the recovery mechanisms as they write application code, exactly as they would write exception handling code. Some exception handling strategies can be used (e.g. attempting a reconnection, cleaning caches, etc), although in preventive fashion. More importantly, PreX adds new strategies, such as dynamic adjustment of the workload or a prioritization of different queries, focusing on the business interest of the application.

The experiments show that PreX can offer very significant advantages to developers. In our scenario, there were decreases in the number of exceptions by two orders of magnitude, accompanied by statistically significant ($p < 0.01$) increases in successful operation throughput.

7. References

[1] J. B. Goodenough, Exception handling: issues and a proposed notation, Communications of the ACM 18 (12) (1975) 683–696.

[2] H. Shah, C. Görg, M. J. Harrold, Why do developers neglect exception handling?, in: Proceedings of the 4th international workshop on Exception handling, ACM, 2008, pp. 62–68.

[3] B. Cabral, P. Marques, Exception handling: A field study in java and. net, in: ECOOP 2007–Object-Oriented Programming, Springer, 2007, pp. 151–175.

[4] F. Salfner, M. Lenk, M. Malek, A survey of online failure prediction methods, ACM Computing Surveys (CSUR) 42 (3) (2010) 10.

[5] X. Chen, C.-D. Lu, K. Pattabiraman, Failure prediction of jobs in compute clouds: A google cluster case study, in: Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on, IEEE, 2014, pp. 341–346.

[6] I. Irrera, M. Vieira, J. Duraes, Adaptive failure prediction for computer systems: A framework and a case study, in: High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on, IEEE, 2015, pp. 142–149.

[7] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, R. Sahoo, Bluegene/l failure analysis and prediction models, in: International Conference on Dependable Systems and Networks, IEEE, 2006, pp. 425–434.

[8] S. Yemini, D. M. Berry, A modular verifiable exception handling mechanism, ACM Transactions on Programming Languages and Systems (TOPLAS) 7 (2) (1985) 214–243.

[9] B. Cabral, A transactional model for automatic exception handling, Ph.D. thesis, Universidade de Coimbra (2009).

[10] K. Kim, I. Choi, C. Park, A rule-based approach to proactive exception handling in business processes, Expert Systems with Applications 38 (1) (2011) 394–409.

[11] A. Avižienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (1) (2004) 11–33.

[12] A. Csenki, Bayes predictive analysis of a fundamental software reliability model, IEEE Transactions on Reliability 39 (2) (1990) 177–183.

[13] S. Fu, C.-Z. Xu, Exploring event correlation for failure prediction in coalitions of clusters, in: Proc. of the 2007 ACM/IEEE Conference on Supercomputing, IEEE, 2007, pp. 1–12.

[14] L. Li, K. Vaidyanathan, K. S. Trivedi, An approach for estimation of software aging in a web server, in: Proceedings of the International Symposium on Empirical Software Engineering, IEEE, 2002, pp. 91–100.

[15] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, W. P. Zeggert, Proactive management of software aging, IBM Journal of Research and Development 45 (2) (2001) 311–332.

[16] A. Andrzejak, L. Silva, Deterministic models of software aging and optimal rejuvenation schedules, in: 10th IFIP/IEEE International Symposium on Integrated Network Management, IEEE, 2007, pp. 159–168.

[17] S. W. Neville, B. Eng, Approaches for early fault detection in large scale engineering plants.

[18] G. A. Hoffmann, Failure prediction in complex computer systems: A probabilistic approach, Shaker, 2006.

[19] G. Hoffmann, K. S. Trivedi, M. Malek, et al., A best practice guide to resource forecasting for computing systems, IEEE Transactions on Reliability 56 (4) (2007) 615–628.

[20] G. Hamerly, C. Elkan, et al., Bayesian approaches to failure prediction for disk drives, in: ICML, Citeseer, 2001, pp. 202–209.

[21] J. F. Murray, G. F. Hughes, K. Kreutz-Delgado, Hard drive failure prediction using non-parametric statistical methods, in: Proceedings of ICANN/ICONIP, Citeseer, 2003.

[22] D. Turnbull, N. Alldrin, Failure prediction in hardware systems, Tech. rep., Tech. rep., University of California, San Diego. available at http://www. cs. ucsd. edu/˜ dturnbul/Papers/ServerPrediction. pdf (2003).

[23] I. Irrera, C. Pereira, M. Vieira, The time dimension in predicting failures: a case study, in: Sixth Latin-American Symposium on Dependable Computing (LADC), IEEE, 2013, pp. 86–91.

[24] S. Garg, A. Van Moorsel, K. Vaidyanathan, K. S. Trivedi, A

methodology for detection and estimation of software aging, in: Proceedings of The Ninth International Symposium on Software Reliability Engineering, IEEE, 1998, pp. 283–292.

[25] R. Vilalta, S. Ma, Predicting rare events in temporal domains, in: IEEE International Conference on Data Mining (ICDM), IEEE, 2002, pp. 474–481.

[26] T.-T. Y. Lin, D. P. Siewiorek, Error log analysis: statistical modeling and heuristic trend analysis, IEEE Transactions on Reliability 39 (4) (1990) 419–432.

[27] F. A. Nassar, D. M. Andrews, A methodology for analysis of failure prediction data, Center for Reliable Computing, Computer Systems Laboratory, Depts. of Electrical Engineering and Computer Science, Stanford University, 1985.

[28] R. Lal, G. Choi, Error and failure analysis of a unix server, in: High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International, IEEE, 1998, pp. 232–239.

[29] C. Leangsuksun, T. Liu, T. Rao, S. Scott, R. Libby, A failure predictive and policy-based high availability strategy for linux high performance computing cluster, in: The 5th LCI International Conference on Linux Clusters: The HPC Revolution, Citeseer, 2004, pp. 18–20.

[30] F. Salfner, M. Schieschke, M. Malek, Predicting failures of computer systems: A case study for a telecommunication system, in: 20th International Symposium on Parallel and Distributed Processing, IEEE, 2006, pp. 8–pp.

[31] F. Salfner, M. Malek, Using hidden semi-markov models for effective online failure prediction, in: Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on, IEEE, 2007, pp. 161–174.

[32] G. A. Hoffmann, F. Salfner, M. Malek, Advanced failure prediction in complex software systems.

[33] G. Hoffmann, M. Malek, Call availability prediction in a telecommunication system: A data driven empirical approach, in: 25th IEEE Symposium on Reliable Distributed Systems, IEEE, 2006, pp. 83–95.

[34] I. Irrera, J. Duraes, M. Vieira, H. Madeira, Towards identifying the best variables for failure prediction using injection of realistic software faults, in: Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on, IEEE, 2010, pp. 3–10.

[35] J. P. Magalhaes, L. M. Silva, Sho wa: a self-healing framework for web-based applications, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 10 (1) (2015) 4.

[36] H. Psaier, F. Skopik, D. Schall, S. Dustdar, Behavior monitoring in self-healing service-oriented systems, Springer, 2011.

[37] C. Schneider, A. Barker, S. Dobson, A survey of self-healing systems frameworks, Software: Practice and Experience.

[38] A. J. Ramirez, D. B. Knoester, B. H. Cheng, P. K. McKinley, Plato: a genetic algorithm approach to run-time reconfiguration in autonomic computing systems, Cluster Computing 14 (3) (2011) 229–244.

[39] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 67–82.

[40] T. G. Dietterich, Machine learning for sequential data: A review, in: Structural, syntactic, and statistical pattern recognition, Springer, 2002, pp. 15–30.

[41] G. F. Hughes, J. F. Murray, K. Kreutz-Delgado, C. Elkan, Improved disk-drive failure warnings, IEEE Transactions on Reliability 51 (3) (2002) 350–357.

[42] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. Jordan, et al., Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization, in: Proc. Second International Conference on Autonomic Computing, IEEE, 2005, pp. 89–100.

[43] A. Ward, P. Glynn, K. Richardson, Internet service performance failure detection, ACM SIGMETRICS Performance Evaluation Review 26 (3) (1998) 38–43.

[44] K. Vaidyanathan, K. S. Trivedi, A measurement-based model for estimation of resource exhaustion in operational software systems, in: Proceedings of the 10th International Symposium on Software Reliability Engineering, IEEE, 1999, pp. 84–93.

[45] C. Domeniconi, C.-S. Perng, R. Vilalta, S. Ma, A classification approach for prediction of target events in temporal sequences, in: Principles of Data Mining and Knowledge Discovery, Springer, 2002, pp. 125–137.

[46] J. R. Lourenço, B. Cabral, J. Bernardino, A predictive model for exception handling, in: New Advances in Information Systems and Technologies, Springer, 2016, pp. 767–776.

[47] Shopizer, (Last accessed 15th April, 2016).
URL `http://www.shopizer.com/`

[48] Apache jmeter, (Last accessed 15th April, 2016).
URL `http://jmeter.apache.org`

[49] Weka project, (Last accessed 15th April, 2016).
URL `http://www.cs.waikato.ac.nz/ml/weka/`

Appendix E

Submission: Predicting and preventing exceptions for increasing reliability

Predicting and preventing exceptions for increasing reliability

João Ricardo Lourenço[1], Bruno Cabral[1], Jorge Bernardino[1,2]

[1]CISUC – Centre of Informatics and Systems of the University of Coimbra, University of Coimbra, 3030-290 Coimbra, Portugal
[2]ISEC – Superior Institute of Engineering of Coimbra, Polytechnic Institute of Coimbra, 3030-190 Coimbra, Portugal

Abstract: The exception handling mechanism has been used as a reliability tool for over four decades. It is present in most modern languages, often with similar constructs, such as "try-catch" or "try-except". However, this model has not changed, in spite of new challenges, namely concurrent and reactive programming. This model is reactive, instead of proactive – exceptions are raised, caught and handled. We propose an alternative exception handling model – PreX – where exceptions are no longer caught, but, instead, are predicted and prevented. This model uses Online Failure Prediction techniques at a low-level, allowing the programmer to be notified of possible exceptions before they happen, ultimately offering new reliability tools. In this work, we present additional validation of the preventive exception handling model with a simulation experiment, concluding that operation throughput can be increased under heavy loads, and that even when prediction accuracy is low, the proposed model offers significant throughput improvements.

1. Introduction

The Exception Handling (EH) mechanism was proposed by Goodenough [1] and has been one of the most used reliability tools in programming languages for more than four decades. This model has gone unchanged, even in the face of concurrent software and programming languages for multi-core platforms (e.g. Scala, Erlang and Elixir). Thus, the sequential Exception Handling model remains the preferred Exception Handling model [2].

However, the ubiquity of the Exception Handling mechanism for error recovery does not imply its most correct or desirable usage. Developers often use EH language constructs as a way of hiding problems, performing log activities or informing the user of unexpected behavior, rather than recovering from it autonomously [3]. This kind of use of EH might be considered a symptom of a design flaw in the mechanism – the system only acts when it is too late, thus making the problem unavoidable.

In a previous work [4], we proposed a new approach to Exception Handling, by providing the means for developers to act on an exception before it happens, thus broadening the range of their revitalization strategies. The approach reshapes the concept of "try-catch" blocks, so that programmers can be alerted of potential exceptions within a given time frame and take some action, much like in the conventional Exception Handling mechanism. This new approach is called **Preventive Exception Handling (PreX)**, and applies the methods and techniques of the Online Failure Prediction field to the lower-level concepts of programming languages, thus empowering programmers to act proactively. PreX introduces a new model for Exception Handling, with the

goal of being easy to use, practical, and a successful integration of the fields of Exception Handling and Online Failure Prediction, as well as the first to act on potential exceptions before they have happened – errors are avoided, rather than handled. In contrast to Online Failure Prediction, which operates on a much higher level of abstraction, PreX allows programmers to produce code that is aware of possible predictions during its execution, so that very fine-grained solutions for exception prevention and reaction to potential exceptions can be applied. By acting on exceptions before they happen, developers get an increased range of techniques for dealing with exceptions. Thus, while traditional Exception Handling techniques can still be used, there is potential for new and hopefully more efficient strategies.

To illustrate the motivation for this new model, consider a system consisting of at least a database and several client applications. Consider also that these client applications are write-heavy, meaning they processes several thousands of write operations per second, sending them to the database. Due to the heavy load, the database may become unresponsive and ultimately trigger a *ConnectionTimeout* exception on one of the applications. That application will then have to attempt to reconnect, and restart where it was previously, if such is really possible. This shows the aforementioned downfall of the conventional Exception Handling mechanism – the system only reacts to exceptions, it does not prevent them. Our motivation stems from this issue – client applications benefit from a prediction (i.e. a warning or alert) that the database may trigger a *ConnectionTimeout* exception. With such a warning, they can, for instance, proactively slow their execution rate and prevent the exception from happening. Ultimately, slowing execution down should prove to be more efficient than triggering the exception and restarting the whole process.

This article is a revised and extended version of our WorldCIST 2016 paper [4]. It improves and complements the former by validating the preventive capabilities of the proposed model using a simulation. By using simulation tools, we can quickly experiment with model parameters and study how PreX is affected by prediction accuracy and different workloads.

The remainder of this paper is structured as follows. Section 2 presents background in Exception Handling models and mechanisms and Online Failure Prediction. Section 3 details the proposed model. Section 4 presents the preliminary experiments used to validate the model's predictive accuracy. Section 5 presents the simulation used to validate the model's preventive capabilities. Finally, Section 6 contains our conclusions and future work.

2. Background and Related Work

Exception Handling separates the operation domain (the execution domain of a particular segment of code) into two distinct domains: the operation's *standard domain*, and the operation's *exceptional domain*. Normal program flow, absent of errors, is contained in the *standard domain*. If an operation is invoked within its exceptional domain, it leads to an exception being *raised*, followed by the invocation of an *exception handler*, that, in theory would deal with the abnormal condition (e.g. by logging it, correcting it, or using a different approach altogether). This invocation is done by a *signaller* (i.e. *callee*). An *Exception Handling Model* defines the interaction between the signaller and its handler. An *Exception Handling Mechanism* defines the language constructs within a programming language to express a given Exception Handling Model. [5]. In this section, we give

an overview of the classical and current Exception Handling models and their limitations.

2.1. Exception Handling Models

Yemini et al. [6] identified four distinct Exception Handling Models, summarized in [7]:

- **Resumption model** – When an exception is raised, the control flow is transferred from the raise point to the handler and, after the exception has been handled, it is transferred back to the raise point. This model effectively binds the caller and the callee together, and is prone to recursive resumption, thus being difficult to implement [7].
- **Termination model** – An exception is raised within a protected block, with the control flow transferred to the handler, terminating any intervening blocks. The control flow then resumes as if the protected block has terminated without any errors. This is the most widespread model in use [7].
- **Retrying model** – The signaller is invoked after the completion of some operation. This model is appropriate to transient faults, where retrying the invocation might lead to no exceptions. The main disadvantages of this model are its inherent implications for non-idempotent operations, counters, etc – the programmer must be wary of how the code executes.
- **Nonlocal transfer** – The program flow can be transferred to any other location in the program. This model has the obvious drawback of being hard to maintain and much more error-prone [7].

Modern advances in Exception Handling, such as concurrent exception handling or alternative models in the context of the actor model and Functional Reactive Programming, still share a common characteristic with these classical models – an exception is raised, and only then can corrective actions happen. This, unfortunately, means that many incorrect uses of Exception Handling Mechanisms arise, with programmers focusing on hiding errors, rather than attempting to fix them [3].

To the best of our knowledge, our model is the first proposal for a preventive exception handling model which shifts the current practice and allows programmers to act before problems arise. However, Kim et al. [8] propose a proactive approach to exception handling within a business process. In their work, the authors note that business processes often involve "human exception handlers" that react to "exceptions". They conclude that there is a need for a proactive exception handling which allows for action as soon as a business process exception is "predicted", and that this action can be specified by an external agent (e.g. a system administrator) as a reaction to a prediction. Kim et al.'s work is, thus, similar to the work presented in this article, but differs significantly in the following key points:

- It concerns exceptions in a business process management context.
- It does not *learn* how to predict business process exceptions. Instead, a system administrator can build a set of rules similar to "$X > Y$". Thus, the prediction code is a set of rules determined from experience, only dependent on business conditions (i.e. this work cannot "predict" a database timeout).

Nevertheless, the core concept of PreX is the same as the concept seen in the work by Kim et al.: a shift from reactive exception handling is needed, in favour of more proactive behaviour.

PreX focuses programming languages, and on systems which can use online failure prediction mechanisms to "learn" exceptions independently of human interaction.

Modern advances in Exception Handling, such as concurrent exception handling or alternative models in the context of the actor model and Functional Reactive Programming, still share a common characteristic with these classical models – an exception is raised, and only then can corrective actions happen. This, unfortunately, means that many incorrect uses of Exception Handling Mechanisms arise, with programmers focusing on hiding errors, rather than attempting to fix them [3]. PreX intends to shift the current practice and allow programmers to act before problems arise.

2.2. Failure Prediction

Recent trends in industry and academia have triggered a shift to new efforts on autonomic computing, trustworthy computing, recovery-oriented computing and other techniques for proactively handling failures. Several techniques have been proposed and used with success, to an extent, in Online Failure Prediction systems. Salfner et al. [9] present a lengthy survey and taxonomy of online failure prediction systems. In their work, these authors explain how Liang et al. [10] explored temporal and spatial correlation to successfully predict hardware component failures in IBM's BlueGene/L. Cheng et al. [11] presented an approach for failure prediction within a high availability cluster system. They showed that they could improve the availability due to accurate prediction and recovery mechanisms (backup nodes and system administrator notifications). Vilalta et al. [12] propose the eventset method, using a data mining approach (a rule-based model). Under specific conditions, they have a false negative error of only 0.16, although this value can be as extreme as 0.83 under other system conditions (false positives are always lower than 0.1).

These and other results show that Online Failure Prediction can be successfully used to predict failures. However, little work has been done for predictions at a more fine-grained level. Predictions are usually made at the system level, at most predicting a generic "crash" of some component. Thus, while promising, these techniques have no practical use for developers who wish to provide specific counter-measures when faced with the possibility of an exception. In this sense, some work has been done in the field of self-healing systems. For example, Magalhães and Silva [13] propose a general self-healing proactive framework for web-based applications. Their work introduces a general framework to create self-healing transactional web-based systems. The framework, although operating at a lower level than traditional Online Failure Prediction methods, does not support run-time notifications at code-level nor operate at the fine-grained level that might be desired for applying more efficient preventive measurements.

3. PreX – A predictive model for exception handling

PreX is an Exception Handling model that focuses on preventing exceptions rather than catching them. The central idea was depicted with the example given in the introduction: it could be more efficient to temporarily reduce the throughput of a write-heavy application than to catch a *ConnectionTimeout* exception and have to restart the process. We now present PreX.

Preventing exceptions implies predicting them. To this end, the area of Online Failure Prediction provides valuable insight. There have been successful failure prediction systems, but these operate on a much broader level. In order to predict exceptions, the proposed model needs to adapt failure prediction techniques to a per-exception basis. Furthermore, since no two systems are alike, the prediction models will have to be trained for specific deployments. Thus, the PreX

model comprises different phases from development to successful prevention:

1. **Coding phase**. The programmer develops the application using a new set of programming language constructs introduced by the PreX model. These are similar to traditional *try-catch* blocks seen in several languages.

2. **Training phase**. In this phase, different machine learning algorithms are applied (after feature selection, data pre-processing, etc), determining which is the most applicable to the specific exception. Data is gathered for different runs of the application, using resource monitoring facilities.

3. **Detection phase**. The application is deployed with the trained model and executed. The model is used to detect potential exceptions, triggering *alarm*s of potential exceptions. If the trained model becomes ineffective due to changes in environment conditions, the training phase might be required again. Alternatively, self-adapting pattern recognition algorithms can be used.

4. **Prevention phase**. If an exception is predicted (in which case an alarm is triggered), the application can apply preventive measures and try to avoid the potential exception from effectively being raised.

In the following sections, each of these phases is detailed from the perspective of the syntax and semantics of the model, followed by an overall perspective of the architecture and the necessary components of PreX. Lastly, the behavior of the system during its different phases is presented.

3.1. Syntax & Semantics

An example of the syntax of PreX is shown in Figure 1. Pseudo-code similar to Java is used. A database connection is established to send a number of pending writes (sent at line 16). PreX provides semantics similar to traditional *try-catch* blocks, although two different constructs are added to the language: *try { }-prevent(){ }-catch(){ }* and *try{ }-prevent_async(){ }-catch(){ }*. Additionally, in PreX, triggering an alarm does not necessarily terminate the execution of code within the *try* block (as opposed to the *termination* model). Instead, execution is resumed as if the alarm had not been triggered, because preventing an exception should not halt normal execution of the current code.

If an exception is predicted, triggering an alarm, such alarm can be handled in two different ways:

- **Synchronously**: execution within the *try* block is suspended and flow is transferred to the *prevent* block. In normal circumstances, the execution is then transferred back to the previous code within the *try* block.

- **Asynchronously**: execution within the *try* block continues normally, and the *prevent_async* block is executed asynchronously.

Thus, the *try* block denotes the scope during which a program cares about predicting some exception, the protected region. This exception may then be predicted, triggering an alarm. If the exception is not predicted or cannot be prevented, the traditional (synchronous) Exception Handling model is used, with program flow being transferred from within the *try* block to the *catch* block.

```
1  // Connect to database and build list of data to write
2  DBConnection c = connectToDB();
3  List<Write> writes = fetchWrites();
4
5  // Try to predict exceptions using the "write-exceptions"
6  // prediction context
7  try("write-exceptions") {                                    (a)
8      for (int i = 0; i < writes.size(); i++) {
9          // Feed writes left to the prediction system
10         sample("writes_left", writes.size()-i);              (b)
11
12         // Don't allow alarms to be triggered
13         no_alarm {                                           (c)
14             print("Writing data!");
15             // Write the data through the connection
16             c.write(writes.get(i));
17         }
18     }
19 } prevent ( ConnectionTimeout, info ) {                       (d)
20     // info has information about the prediction
21     // (e.g. lead time)
22
23     // Sleep for some time to try to prevent an exception
24     sleep(1000);
25 } catch ( ConnectionTimeout e ) {
26     // The exception could not be avoided...                 (e)
27     error("Database timeout!");
28 }
```

Fig. 1. *Example of PreX using synchronous try-catch-prevent.*

When using the synchronous approach, alarms are not triggered at just any point in time within the *try* block. Within this block, every program statement will be executed without any interruption from the exception prediction mechanism. The flow of execution only moves to the prevention handler between statements, thus eliminating the need for propagation in the call stack. Thus, unlike traditional Exception Handling models, PreX uses a static binding approach, meaning that exceptions are not propagated. Instead, they are tied to the handler of the *try-prevent-catch* or *try-prevent_async-catch* to which they belong. Only the closest code, within the exact context of the particular exception, can know how to react to a specific alarm. If developers wish to prevent the triggering of alarms during the execution of a set of statements, a special *no_alarm* keyword can be used to denote a new scope within which alarms are not possible. In Figure 1, lines 13-17 belong to one of these scopes. In the asynchronous approach, as in the synchronous approach, exceptions are only delivered when the program flow is outside the protected block. Thus, the *no_interrupt* keyword can act as a synchronization primitive between the prediction asynchronous handler code and the code within the *try* block.

PreX allows programmers to periodically sample variables that they think will be useful for prediction, in addition to system variables monitored with custom probes. For instance, the remaining number of operations left might be useful in predicting connection timeouts. These variables can be supplied to the prediction system at any time using the *sample (<variable_name>,<variable_value>)* construct. Furthermore, since no two systems are alike, the prediction models will have to be trained for specific deployments. In particular, note that, for instance, *ConnectionTimeout* exceptions may be different depending on the workload (e.g. "write-heavy" vs "read-heavy") or the variables being provided by the program. This motivates the need to distinguish different blocks of code and assign them meaningful names. Thus, the *try* keyword requires an additional argument that uniquely identifies the block of code that it encloses: *try(<prediction context>)*. Write-heavy blocks can then use *try("write-exceptions")*, whereas read-heavy blocks can use *try("read-exceptions")* to handle two completely different models for the same kind of exception (*ConnectionTimeout*) under different contexts. The argument of the *try* keyword is the *prediction context*. Several alarms can be triggered within the same *prediction context*, and a *prediction context* binds training data and a prediction model to a unique name. An example of this construct is in line 7 of Figure 1.

To train the model, the system administrator may specify which *prediction context*s he/she wants the program to be trained in during a training phase. When in this training phase, no alarm can be triggered in those *prediction context*s and exceptions can only be caught. Data with the *sample* keyword is still fed to the training mechanism, and if the exception is raised and caught, this training mechanism is notified to adapt its prediction models.

3.2. Architecture

Within PreX, exceptions can be predicted using system-wide information. Several entities can share information used for prediction (more data, when appropriately filtered, implies better predictions). An entity need not be running on an independent machine on its own, and different entities might share the same machine. There are three main kinds of entities within a system using PreX:

- **Coordinator entities**: A (potentially replicable) coordinator entity, responsible for aggregating the data from the other two types of entities and running the prediction system.
- **Data gathering entities**: These entities feed periodic samples of data (e.g. memory and CPU

usage) to the coordinator entities. Most of the prediction data comes from these entities, which don't execute any code that wants to be alarted of possible exceptions. The sampling rate for each of these variables is not pre-determined and may vary according to system load and characteristics.

- **Code entities**: Whenever a *try* block is entered, the exception handling mechanism spawns a code entity that connects to the coordinator entity. Each code entity may want to register with the coordinator to be notified of predictions of certain kinds of exceptions within a *prediction context*.

It is then clear that each *try-predict-catch* or *try-predict_async-catch* block spawns a new code entity. The coordinator entity is responsible for running the failure prediction methods for predicting exceptions, which lead to alarms triggered by the code entity. These can be selected *a priori* by a system administrator during the training phase. The behaviour of the code and coordinator entities is different during the training phase. During this phase, the code entity registers that it will be sending data and information regarding a given *prediction context* and a given exception. The coordinator then uses this data to train the model.

Since we are dealing with a distributed system, it is complex to generate samples at the exact same time in every entity. For this reason, the coordinator entity groups data within time windows. Precise information about when an event (e.g. a data sample) happened is lost in favor of a more general interval during which several events happened. This data can then be processed using Online Failure Prediction methods such as those presented in [12], [14] or [15].

3.3. Behaviour

To illustrate the behaviour of PreX, Figure 2 depicts the interaction between code, entities, and the prediction system for the code example in Figure 1 during a prediction. Notice that the code uses the synchronous prediction model. Also note that if the code was executed during the training phase, no alarm would be triggered, so section *d* would not be entered.

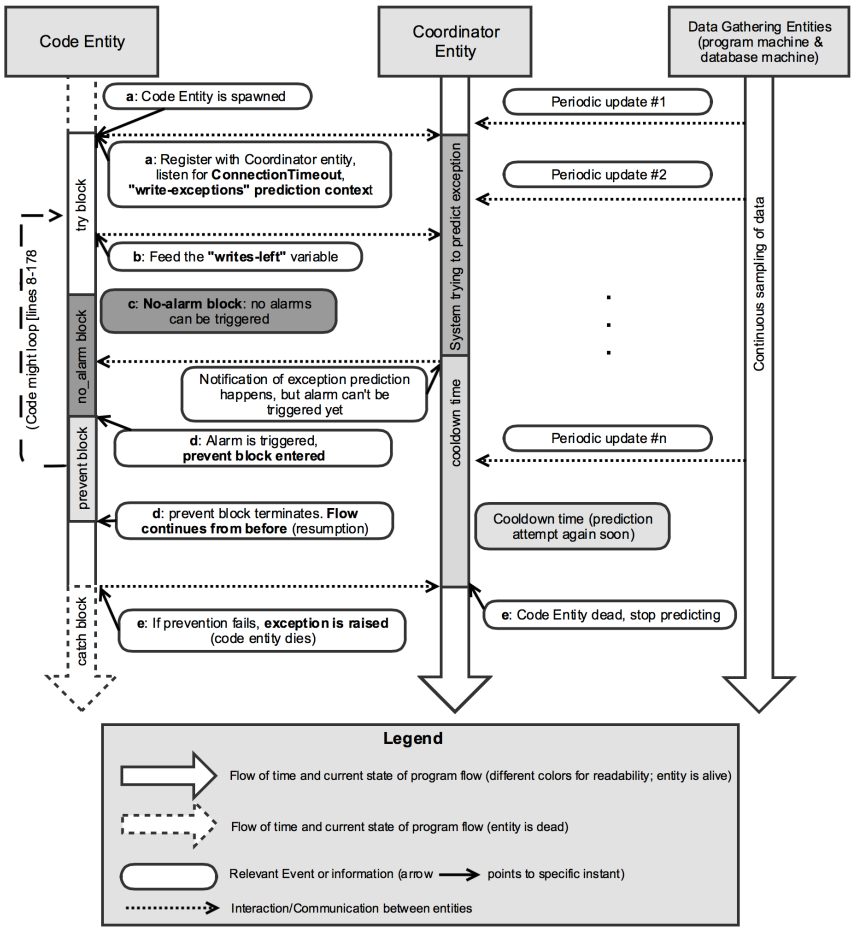

Fig. 2. *Interaction between code and entities through time for the code example in Figure 1. For simplicity, in this example, the loop only runs once.*

4. Predicting Exceptions

We have conducted experiments with the TPC-W benchmark to demonstrate that the proposed model is feasible in the **prediction** of likely exceptions. In the following Section (Section 5), we use a simulation to demonstrate that the model is useful in the **prevention** of exceptions and that it can increase the overall reliability of a system. TPC-W simulates the activities of a retail store Web site. It has a workload generator that emulates the behavior of users according to pre-specified configurations. In our experiments, the connection pool of the TPC-W server was exhausted due to client overload, resulting in different *NullPointerException*s. Using failure prediction methods, we attempted to predict these exceptions.

4.1. Experimental Setup

The setup consisted of three virtual machines running the Crunchbang Linux distribution (Irrera et al. [15] showed that virtualization did not significantly influence failure prediction results). These were allocated with 1 GB of RAM and a single virtual CPU core. They communicated through a local network bound to the host machine. One of the machines ran the TPC-W Server, the other the MySQL database, and the third the TPC-W load-generator. In this third machine, a custom-built data gathering tool was placed, sampling data at the rate of 100 samples per second. 49 variables were sampled, including CPU load, open TCP connections, and number of running processes. This setup can be seen in Figure 3.

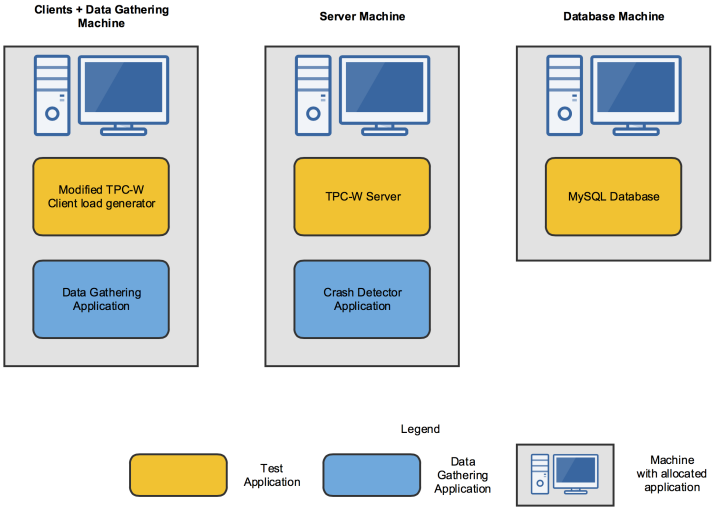

Fig. 3. *Experimental setup of the preliminary experiments.*

After recording several *golden-runs* (executions with no failure), as well as several executions with failure, the final data was passed to a pre-processing window-based algorithm (the dataset was filtered to contain a balance between both kinds of runs). This algorithm involved two stages: (i) *time-window construction* and (ii) *window-merge*. In the first, fixed time windows of size T, starting at time $t = 0$ were created. For each of the 49 variables, samples within the same window were condensed using the mean, standard deviation, maximum, minimum, and derivative (rate of change). Thus, each window contained 245 variables (5×49). The second step, *window-*

merge, involved concatenating N of the previously generated time-windows, appending them with a binary variable indicating if a crash was recorded within the next group of merged windows. In practice, each group of merged windows offers a *time-to-failure* prediction of $N \times T$. The final windows were then processed within *Weka*[1] as a classification problem. Empirically, we chose $T = 5$ s and $N = 2$, meaning a *time-to-prediction* of 10 seconds. The experiments were done for a TPC-W simulation of a "slow" and "fast" ramp up of users. This was done to assess if classifiers trained with one set of data could still be accurate on different data within similar circumstances.

4.2. Results

The experimental results were promising. Using 10-fold cross-validation, of the several classifiers tested, which included decision trees, support vector machines and belief networks, a Naive Bayes classifier only failed to predict failures 1.52% of the time (false negative rate), and inaccurately predicted a failure 8.89% of the time (false positive rate). To further check that this result was not due to overfitting, the same classifiers were trained with data from the "faster" ramp of TPC-W clients and then validated on the "slower" ramp of TPC-W clients. The aforementioned Naive Bayes classifier remained the best, with a false negative rate was of 23.3%, and a false positive rate of 7.01%.

These results, although very preliminary, show that prediction of exceptions is within our reach, and that such a model can be useful for programmers – in this scenario, a time-to-prediction window of 10 seconds would allow the TPC-W clients to reduce their rate of requests, thus preventing or delaying the exception.

5. Validation with a simulation

In the previous section, we presented the preliminary experiments designed to show that exceptions can be predicted. However, the PreX model involves two distinct stages: prediction and prevention. The latter stage is heavily dependent on the application domain, meant to be written by the programmer (when an alarm is triggered). Nevertheless, it is important to assess the usefulness of the proposed model with regards to prevention. In this section, we present the simulation experiment used to validate the prevention capabilities of PreX.

Using simulation tools, we can quickly change model parameters and assess their overall impact on the system. For example, it is possible to assert the impact of too many false negatives on the system. Thus, without the need for a full experiment, it becomes feasible to evaluate the model's prevention capabilities, as long as we are aware of the simulation limitations. In the next sections, we present the simulated scenario, the simulation parameters, its implementation and limitations.

5.1. Simulated Scenario

The simulated scenario closely resembles the one presented in Section 4. A number of simulated clients are running concurrently, overloading a simulated server that accesses a database (which, in our preliminary experiments, was the TPC-W Server). Figure 4 shows the simulated scenario.

In the case of TPC-W, exceptions happen due to connection pool exhaustion. However, real-world applications are often protected against this kind of situation, so we opted by modeling the exceptions differently and more realistically. As the server gets overloaded with clients, the time to run queries increases, eventually becoming impractical, with some database management systems

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

triggering an exception in server code when the query exceeds a fixed time. This is the scenario which we model in our simulation: if a query reaches a certain time limit (the query *max_time*), an exception is raised in the server code. In addition, the query time should depend on the number of active clients.

If we include PreX in this scenario, it now becomes possible for alarms to be triggered, signifying the prediction of a potential exception. When this occurs, a preventive action can be taken. Note that due to existence of false negatives, it is possible that this action is taken even if the query time is far from the *max_time*.

The preventive action is written by the application developer, and might involve complex procedures. However, we intended to assess the impact of a simple action first, so that complicated preventive actions were not unnecessarily coded. To this end, we modeled our clients to sleep for a random amount of time, uniformly distributed in a fixed interval. By sleeping, the clients essentially delay their execution and re-schedule it to a later time, according to a uniform distribution. The rationale behind this is that heavy/lengthy queries are scattered across a random uniform interval, effectively reducing the load on the database and throttling the rate of requests. Consequently, faster queries are prioritized, allowing these clients to disconnect and freeing more CPU room for the execution of the heavy queries, possibly reducing the number of exceptions and increasing the throughput of successful operations.

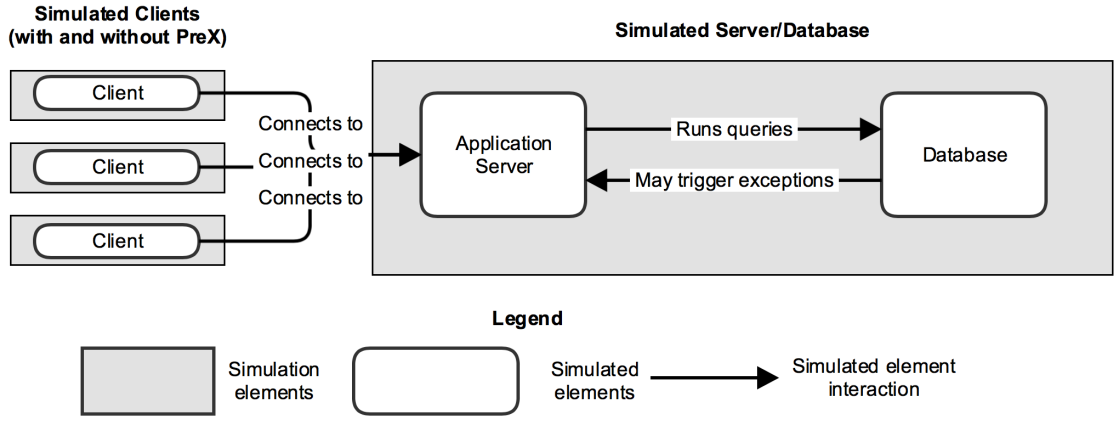

Fig. 4. *Simulation scenario. Each client is an independent agent.*

5.2. Simulation Parameters

In the simulation, a number of clients are simulated using agents. These agents's behaviour is different, depending on whether we are using the PreX model or not. If we do not use this model, their behaviour can be modeled by the flowchart seen in Figure 5. A simulated client sleeps during a connection time, connects to the server (incrementing the number of active clients) and, based on the number of active clients determines a query time. By comparing this time with the *max_time*, it determines if an exception should be triggered or not. If an exception is triggered, the client should sleep for *max_time*, whereas if it is not triggered, it should sleep for the determined query time. When PreX is used, their behaviour is extended to the one seen in the flowchart in Figure 6. This

flowchart adds a set of steps that simlate the prevention mechanism: a client simulates a prediction (based on the FNR and FPR) and, if this prediction is true, invokes the preventive action (a sleep). When a certain maximum simulation time is reached, every client is terminated.

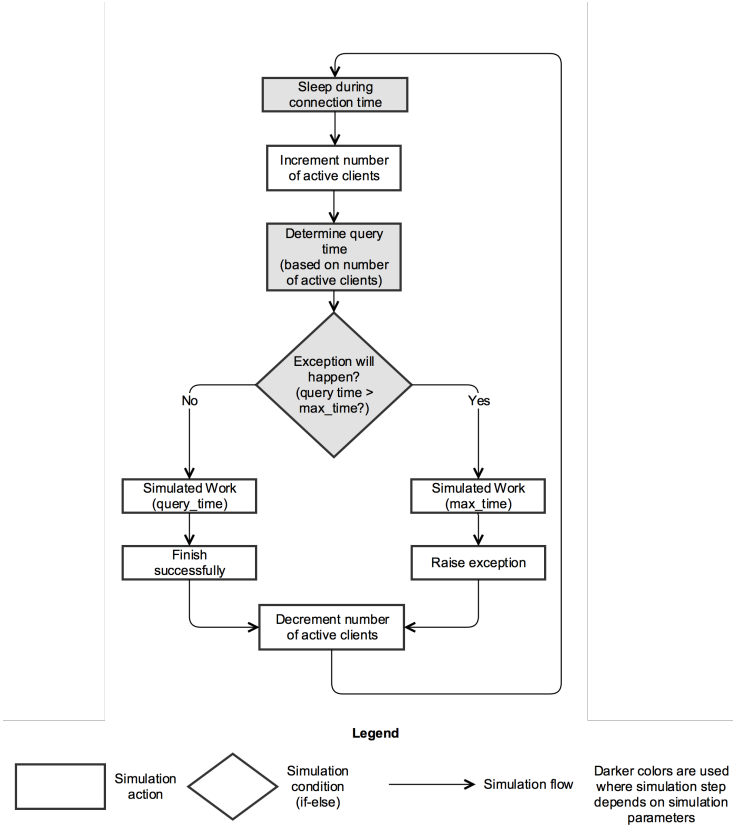

Fig. 5. *Flowchart of the behavior of each individual simulated client when PreX is not used.*

Figures 5 and 6 show that there are a number of steps in the simulation of each client that depend on simulation parameters. Some of these parameters, such as the time to run each query, depend on the application domain, and should be selected to closely resemble real-world behaviour. Other parameters, such as the time to sleep in a recovery action, or the model accuracy, are configurable parts of our model. It is, then, useful to observe how the simulation outcome changes depending on these latter parameters.

The simulation parameters are, then:

- **Number of clients**. They represent the number of active clients connecting and overloading the server. This is similar to the number of virtual clients used by the TPC-W load generator. It is mostly through this parameter that different workloads can be simulated.
- **Connection time**. This parameter represents the time that it takes for a client to start a connection to the server, before it runs any queries. In practice, lower values mean that more clients can connect in a smaller interval, leading to higher loads on the server. We determined this parameter from the experimental data gathered in Section 4.

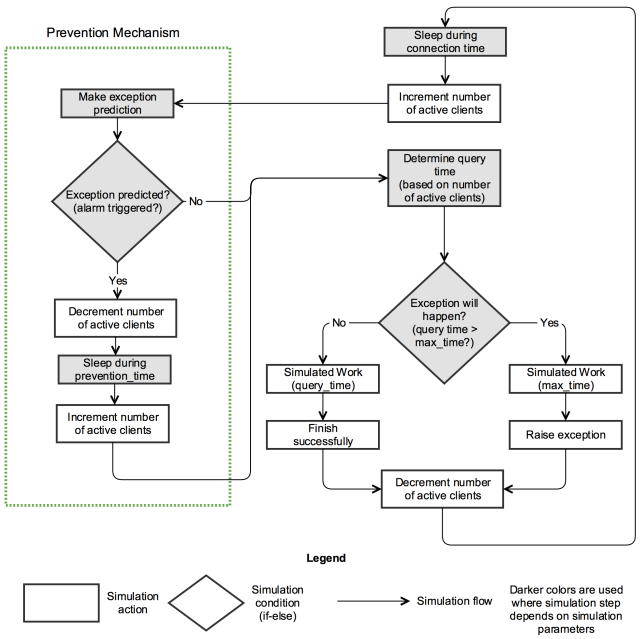

Fig. 6. *Flowchart of the behavior of each individual simulated client when PreX is used.*

- **Query time**. The query time, that is, the time to run a query, depends on the number of currently connected clients. More clients should lead to a longer query time. This parameter is essential to the simulation, and must closely resemble realistic environments. We adjusted our experimental scenario to measure the query time according to the number of clients. These experiments revealed a linear relationship between the query time and the number of clients. Other articles where TPC-W is used have shown that this linearity exists [16–18]. As such, we modeled the query time according to a linear expression of the form:

$$query\_time = base\_time + k \times number\_of\_clients + \mathcal{N}(m, \sigma)$$

  where the $base\_time$ and k are deployment-specific attributes determined empirically and where m and σ are the mean and standard deviation of a normal distribution used to add some noise and randomness to the *query time*. m and σ were chosen to add very little noise, since the query times did not exhibit a high standard deviation.

- **Maximum query time (max_time)**. This value represents the maximum time that a query can take before the database raises an exception in the server code. Lower values lead to more exceptions. This parameter was configured to have "sensible" values for an operation in an e-commerce web-site.

- **Exception prediction accuracy**. This parameter models the accuracy of the PreX model. In our simulation, it is represented by a tuple $(FNR,\ FPR)$, representing the false negative rate and the false positive rate, respectively. This way, depending on whether the query time exceeds the max_time (i.e. an exception will happen), we can accurately simulate the outcome

of PreX's predictions. It is interesting to experiment with different values and determine their effect on the simulation metrics.

- **Prevention sleep time (prevention_time)**. This parameter regards the prevention mechanism presented in the previous section. When a client receives an alarm of a potential exception, it sleeps for a given *prevention_time* to try to prevent the exception and reduce the load on the server. The prevention_time is, thus another interesting parameter to test for different values. In this work we defined it as a value sampled form a uniform distribution between 33.3% and 100% of the max_time (the rationale is that we should wait some time, at most max_time until we try again, but preferably less).
- **Simulation time**. The simulation time should not have a significant effect in the simulation, as long as it is sufficient for simulation stabilization. Indeed, it was selected so as to allow for this stabilization.

5.3. Implementation and metrics

The simulation was implemented in Python 3 using the SimPy[2] library. The clients were implemented as a simulation object which yielded execution whenever a simulation step should wait for a given time (i.e. a "timeout"). The server was simulated using a shared resource where the number of active clients was kept, thus influencing the query time.

SimPy is a unitless simulation framework. However, considering that each simulation step/unit is a millisecond, we can simulate with millisecond precision. To reach this goal, all the empirical parameters (e.g. the query base time) were measured with millisecond precision. As noted in the previous section, the simulation was run until after it had stabilized and long enough to gather enough data, totalling 5 simulated minutes.

To analyze the simulation and evaluate the proposed model, several performance metrics were gathered. These allow analysis of the number of exceptions, number of successful operations and successful operation throughput.

5.4. Limitations

Although the use of simulation software offers development advantages, as well as a fast way to study different model parameters, it is not without its limitations. The simulation is only as good as its underlying model. In our model of the TPC-W scenario, we put aside the ramp-up period and scheduled all clients to start executing concurrently. This limitation might impact the generality of our results. Similarly, we have assumed that there is independence of connection time and query time. This makes sense when a load balancer is used, forcing the bottleneck to be at the database layer, but it was not necessarily the case with our TPC-W setup.

This simulation does not distinguish between query types. This means that read queries are treated exactly the same as write queries. There are scenarios where this is the case, but the TPC-W workload used in the previous experiments contained a mix of both queries. It may nevertheless still be possible that the linear relationship between the number of clients and the query time holds. However, more advanced prevention actions, such as prioritizing write queries (usually shorter and more business-critical in e-commerce websites) over read queries, are not allowed with this simulation model.

[2] `https://simpy.readthedocs.io`

Finally, this simulation does not deal with dynamic environments, where the performance accuracy might change at run-time. However, we shall see, the results allow us to understand what would happen to the system if the performance accuracy suddenly changed.

5.5. Results

To analyse the effectiveness of the proposed prevention action and, hence, validate the usefulness of the proposed model, several measurements were made.

Figure 7 shows the operation throughput as a function of the number of clients (n), for different scenarios. The baseline scenario corresponds to normal operation, when PreX is not used. The remaining scenarios show the use of PreX and the prevention action with varying degrees of prediction accuracy. The accuracy of the preliminary results is used, in its 10-fold cross-validation variant (FNR=0.0152, FPR=0.0889) and its "test on one, validate on another" variant (FNR=0.233, FPR=0.0701). Additionally, there are three scenarios with varying accuracy which reveal how the system behaves: FPR=FNR=0.5; FPR=0.1,FNR=0.9 and FPR=0.9,FNR=0.1. In total, the 7 different scenarios allow an evaluation of PreX. Figure 8 shows the total number of recorded exceptions for the same scenarios.

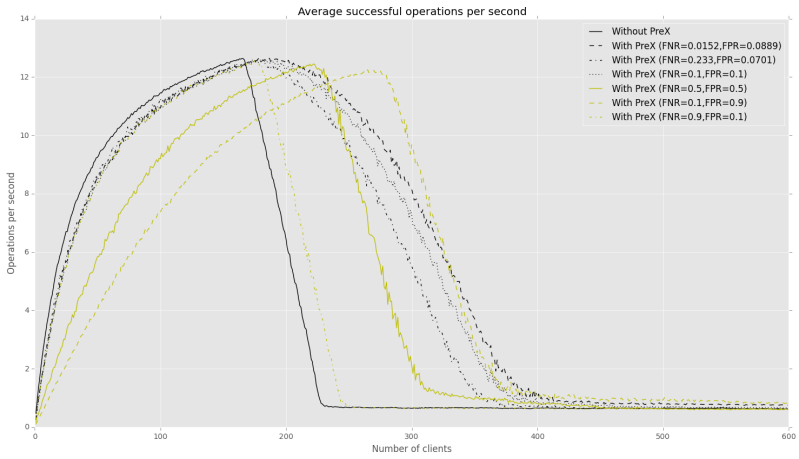

Fig. 7. *Successful operation throughput with and without PreX, based on number of clients, for different prediction accuracy models.*

It is clear from Figures 7 and 8 and that PreX offers increased throughput when the server is overloaded. Due to false positives, the prevention mechanism leads to a lower throughput in the absence of exceptions ($n < 180$). However, when exceptions start happening ($n = 180$), the throughput abruptly falls in the absence of PreX. In contrast, PreX delays the appearance of exceptions and offers increased throughput, offering advantages to application developers which intend to increase the reliability of their systems.

An increase in the false positive rate naturally leads to a decrease in operation throughput, delaying the appearance of exceptions. When this value is very high, the throughput can decrease by 50%, but exceptions are also delayed. This implies that the preventive action is also useful even when an exception is not bound to happen. However, it is clear that it is most useful when exceptions are accurately predicted (e.g. FNR=FPR=0.1). The area under curve seems larger when

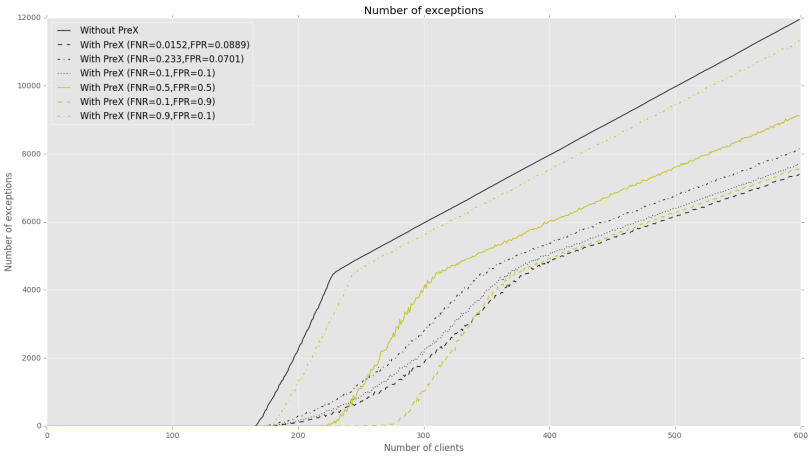

Fig. 8. *Total number of exceptions with and without PreX, based on number of clients, for different prediction accuracy models.*

there is an adequate balance of FNR and FPR (higher prediction accuracy). This is the case with the empirical data used (FNR=0.233, FPR=0.0701). Thus, the simulation shows that the TPC-W scenario, whose prediction model we already know, is an ideal use-case for PreX and the proposed prevention action.

Figures 9 and 10 show the throughput and number of exceptions when using PreX with varying degrees of false positive rate. The false negative rate is fixed at the empirically determined value for TPC-W (0.233). Three different scenarios (with and without PreX) are represented: for 128, 192 and 256 clients. If we look at Figures 7 and 8 we see that these three workloads have clearly different characteristics. The first represents the absence of exceptions. The last represents a full server overload. The second one is a balance between both, in the transition phase where the server can still accommodate some clients, but already catches exceptions. It makes sense, then, to look at these three different scenarios.

As we have seen before, a higher false positive rate has a negative effect in the operation throughput when exceptions are rare. However, in the scenario where the server is overloaded (256 clients), higher values actually have a positive effect, once again strengthening the idea that the preventive action is useful even before the exception is predicted. In general, there are less exceptions when the preventive action is used more often, but this often means that the overall throughput decreases (since operations are being adjourned for a later time).

Figures 9 and 10 show similar plots, regarding the throughput and number of exceptions when using PreX with varying degrees of false negative rate.

The analysis of the model's behaviour with regards to false negative rate is more straightforward. An increase in false negative rate leads to more exceptions, because the prevention mechanism is less used. Nevertheless, it is interesting to note that even when there are many false negatives (i.e. exceptions are often not predicted), the proposed model offers advantages. For example, a false negative rate of 0.8 still represents a three-fold increase in operation throughput for the heaviest workload (256 clients). This shows that PreX offers new reliability options which are useful to developers.

Lastly, Figures 13, 14 and 15 show the throughput during each second of three simulated sce-

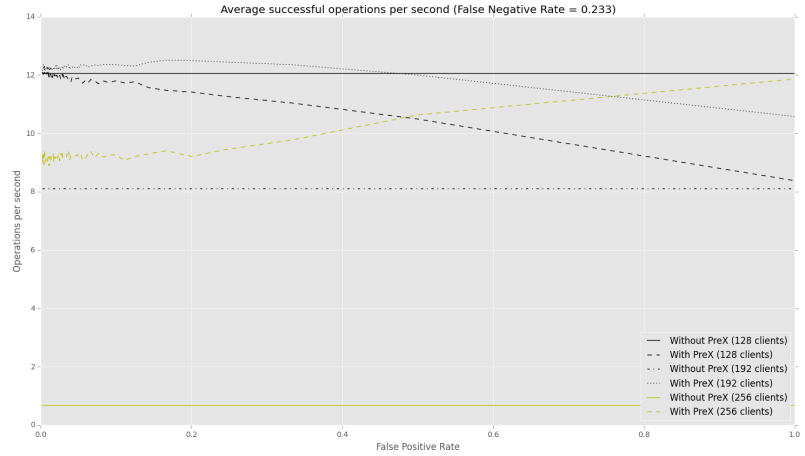

Fig. 9. *Successful operation throughput with and without PreX, based on the false positive rate, for different workloads.*

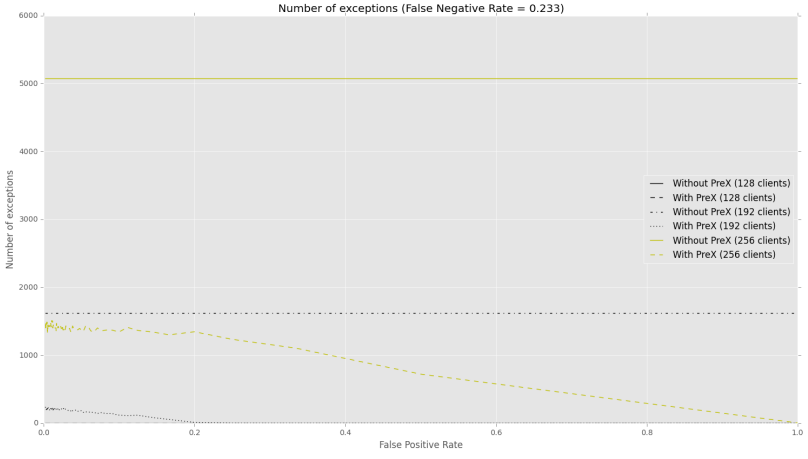

Fig. 10. *Total number of exceptions with and without PreX, based on the false positive rate, for different workloads.*

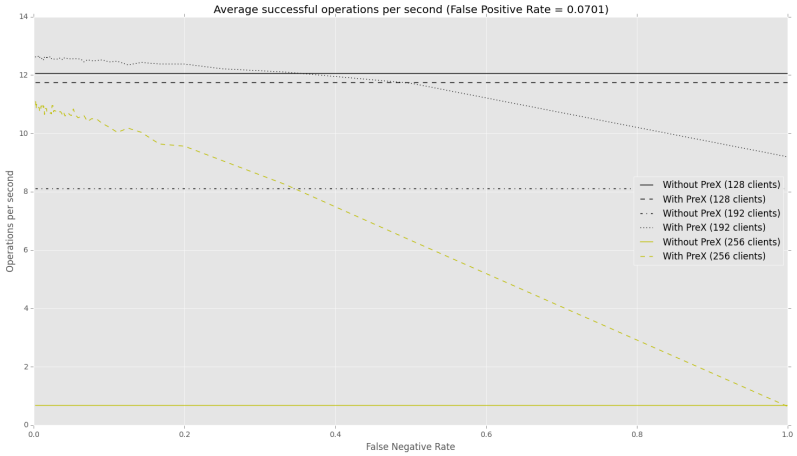

Fig. 11. *Successful operation throughput with and without PreX, based on the false negative rate, for different workloads.*

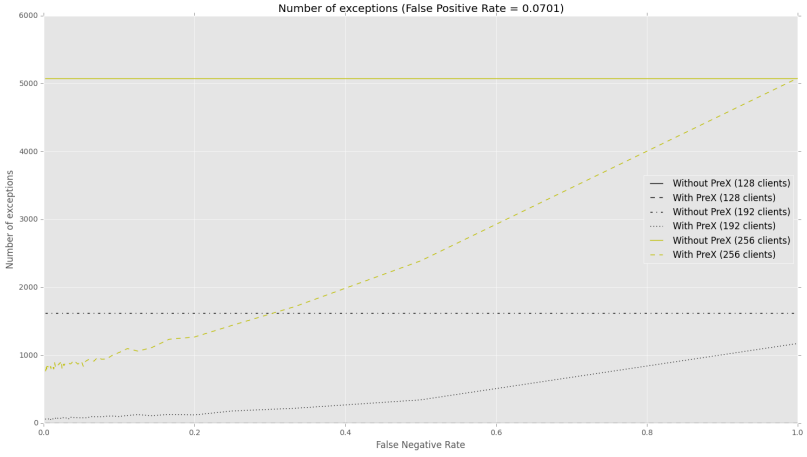

Fig. 12. *Total number of exceptions with and without PreX, based on the false negative rate, for different workloads.*

narios. Each of these represents a different workload, once again using 128, 192 and 256 clients. The prediction accuracy used the one determined empirically in the previous section (FNR=0.233, FPR=0.0701), since these are the most realistic values currently available.

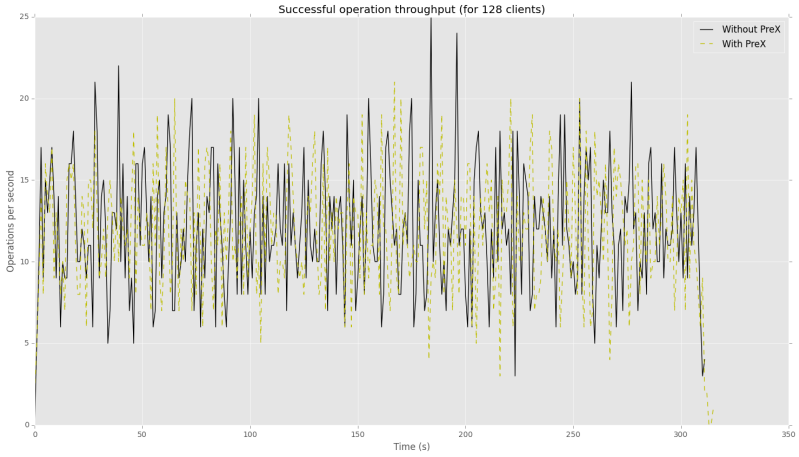

Fig. 13. *Successful operation throughput in one simulation with and without PreX, for a workload with no exceptions, using the empirically determined false negative and false positive rates.*

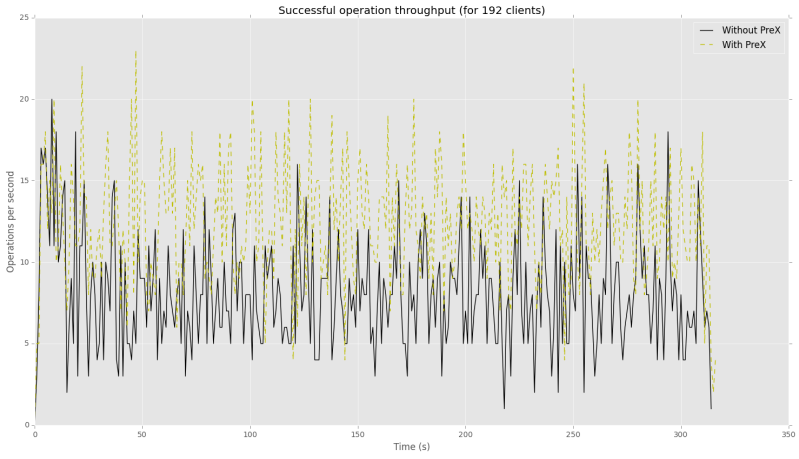

Fig. 14. *Successful operation throughput in one simulation with and without PreX, for a workload with some exceptions, using the empirically determined false negative and false positive rates.*

It can be concluded, from Figures 13, 14 and 15, that, whenever there are exceptions, PreX offers an increase in operation throughput during the simulation. For example, in the case of 192 clients (Figure 14), this increase is marginal, but guarantees that, in the end, more operations will have completed successfully. In the case of 256 clients (Figure 15), the original simulation flatlines its throughput due to the constant overload of clients. By using PreX, it is possible to continue operation, even if some exceptions are triggered. Lastly, in the workload with no exceptions (Figure 13), we see the effect of false positives marginally decreasing the operation throughput when

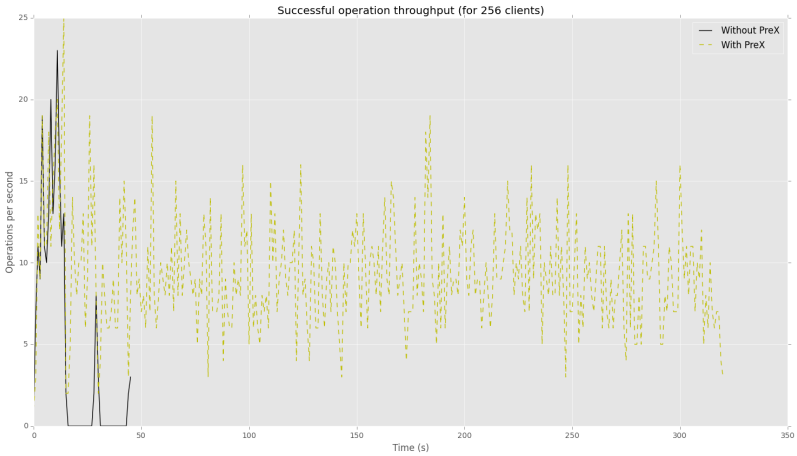

Fig. 15. *Successful operation throughput in one simulation with and without PreX, for a workload with many exceptions, using the empirically determined false negative and false positive rates.*

PreX is used. The increase in throughput is a direct consequence of the uniform redistribution of load which reduces the amount of concurrent clients and, ultimately, increases individual query performance (if there are less simultaneous clients, the overall query performance increases).

6. Conclusions and Future Work

In this paper, we have extended on previous work, where we proposed a new Preventive Exception model that defies nowadays' Exception Handling preconceptions. Current research in exception handling and online failure prediction shows that a fine-grained system for predicting exceptions is currently missing. Instead of catching exceptions, this model proposes that the system, as a whole, actively work towards predicting and preventing exceptions. Applications can then be more resilient, robust, reliable and have increased performance.

Our preliminary results show that it is possible to predict exceptions, and that a paradigm shift towards prevention, rather then reaction, is quite within our reach. Furthermore, the simulation experiments shown in this work also demonstrate the model's usefulness as a reliability tool, allowing exceptions to be prevented and an overall increase in system throughput and availability. We have shown that even when the accuracy of the prediction engine is low, there is an increase in throughput and system stability.

As future work, we intend to develop a proof of concept implementation of PreX in a modern programming language, extending the simulation validation with a real implementation.

7. References

[1] Goodenough, J.B.: Exception handling: issues and a proposed notation. Communications of the ACM **18**(12) (1975) 683–696

[2] Fonseca, A., Cabral, B.: Handling exceptions in programs with hidden concurrency: New

challenges for old solutions. In: Exception Handling (WEH), 2012 5th International Workshop on, IEEE (2012) 14–17

[3] Cabral, B., Marques, P.: Exception handling: A field study in java and. net. In: ECOOP 2007–Object-Oriented Programming. Springer (2007) 151–175

[4] Lourenço, J.R., Cabral, B., Bernardino, J.: A predictive model for exception handling. In: New Advances in Information Systems and Technologies. Springer (2016) 767–776

[5] Issarny, V.: An Exception Handling Mechanism for Parallel Object-oriented Programming: Towards the Design of Reusable and Robust Distributed Software. Inst. National de Recherche en Informatique et en Automatique (1992)

[6] Yemini, S., Berry, D.M.: A modular verifiable exception handling mechanism. ACM Trans. on Programming Languages and Systems (TOPLAS) **7**(2) (1985) 214–243

[7] Cabral, B.: A Transactional Model for Automatic Exception Handling. PhD thesis, Universidade de Coimbra (2009)

[8] Kim, K., Choi, I., Park, C.: A rule-based approach to proactive exception handling in business processes. Expert Systems with Applications **38**(1) (2011) 394–409

[9] Salfner, F., Lenk, M., Malek, M.: A survey of online failure prediction methods. ACM Computing Surveys (CSUR) **42**(3) (2010) 10

[10] Liang, Y., Zhang, Y., Jette, M., Sivasubramaniam, A., Sahoo, R.: Bluegene/l failure analysis and prediction models. In: International Conference on Dependable Systems and Networks, IEEE (2006) 425–434

[11] Cheng, F.T., Wu, S.L., Tsai, P.Y., Chung, Y.T., Yang, H.C.: Application cluster service scheme for near-zero-downtime services. In: Proceedings of the 2005 IEEE International Conference on Robotics and Automation, IEEE (2005) 4062–4067

[12] Vilalta, R., Ma, S.: Predicting rare events in temporal domains. In: Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on, IEEE (2002) 474–481

[13] Magalhaes, J.P., Silva, L.M.: Showa: a self-healing framework for web-based applications. ACM Trans. on Autonomous and Adaptive Systems **10**(1) (2015)

[14] Chen, X., Lu, C.D., Pattabiraman, K.: Failure prediction of jobs in compute clouds: A google cluster case study. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE (2014) 341–346

[15] Irrera, I., Duraes, J., Madeira, H., Vieira, M.: Assessing the impact of virtualization on the generation of failure prediction data. In: Sixth Latin-American Symposium on Dependable Computing (LADC), IEEE (2013) 92–97

[16] Amza, C., Chanda, A., Cox, A.L., Elnikety, S., Gil, R., Rajamani, K., Zwaenepoel, W., Cecchet, E., Marguerite, J.: Specification and implementation of dynamic web site benchmarks. In: Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on, IEEE (2002) 3–13

[17] Elnikety, S., Nahum, E., Tracey, J., Zwaenepoel, W.: A method for transparent admission control and request scheduling in e-commerce web sites. In: Proceedings of the 13th international conference on World Wide Web, ACM (2004) 276–286

[18] Sivasubramanian, S., Pierre, G., van Steen, M., Alonso, G.: Analysis of caching and replication strategies for web applications. Internet Computing, IEEE **11**(1) (2007) 60–66

References

[1] John B Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[2] Alcides Fonseca and Bruno Cabral. Handling exceptions in programs with hidden concurrency: New challenges for old solutions. In *Exception Handling (WEH), 2012 5th International Workshop on*, pages 14–17. IEEE, 2012.

[3] Hina Shah, Carsten Görg, and Mary Jean Harrold. Why do developers neglect exception handling? In *Proceedings of the 4th international workshop on Exception handling*, pages 62–68. ACM, 2008.

[4] Bruno Cabral and Paulo Marques. Exception handling: A field study in java and. net. In *ECOOP 2007–Object-Oriented Programming*, pages 151–175. Springer, 2007.

[5] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010.

[6] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Failure prediction of jobs in compute clouds: A google cluster case study. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 341–346. IEEE, 2014.

[7] Ivano Irrera, Marco Vieira, and Joao Duraes. Adaptive failure prediction for computer systems: A framework and a case study. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 142–149. IEEE, 2015.

[8] Yinglung Liang, Yanyong Zhang, Morris Jette, Anand Sivasubramaniam, and Ramendra Sahoo. Bluegene/l failure analysis and prediction models. In *International Conference on Dependable Systems and Networks*, pages 425–434. IEEE, 2006.

[9] Peter M Melliar-Smith and Brian Randell. Software reliability: The role of programmed exception handling. In *ACM SIGSOFT Software Engineering Notes*, volume 2, pages 95–100. ACM, 1977.

[10] Jean-Claude Laprie, Karama Kanoun, et al. Software reliability and system reliability. *Handbook of software reliability engineering*, pages 27–69, 1996.

[11] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[12] Attila Csenki. Bayes predictive analysis of a fundamental software reliability model. *IEEE Transactions on Reliability*, 39(2):177–183, 1990.

[13] Jonas D Pfefferman and Bruno Cernuschi-Frías. A nonparametric nonstationary procedure for failure prediction. *IEEE Transactions on Reliability*, 51(4):434–442, 2002.

[14] Song Fu and Cheng-Zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[15] Kalyanaraman Vaidyanathan and Kishor S Trivedi. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 84–93. IEEE, 1999.

[16] Lei Li, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. An approach for estimation of software aging in a web server. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 91–100. IEEE, 2002.

[17] Vittorio Castelli, Richard E Harper, Philip Heidelberger, Steven W Hunter, Kishor S Trivedi, Kalyanaraman Vaidyanathan, and William P Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, 2001.

[18] Artur Andrzejak and Luis Silva. Deterministic models of software aging and optimal rejuvenation schedules. In *10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 159–168. IEEE, 2007.

[19] Stephen William Neville and B Eng. Approaches for early fault detection in large scale engineering plants. 1998.

[20] Günther A Hoffmann. *Failure prediction in complex computer systems: A probabilistic approach*. Shaker, 2006.

[21] Guenther Hoffmann, Kishor S Trivedi, Miroslaw Malek, et al. A best practice guide to resource forecasting for computing systems. *IEEE Transactions on Reliability*, 56(4):615–628, 2007.

[22] Greg Hamerly, Charles Elkan, et al. Bayesian approaches to failure prediction for disk drives. In *ICML*, pages 202–209. Citeseer, 2001.

[23] Michele Pizza, Lorenzo Strigini, Andrea Bondavalli, and Felicita Di Giandomenico. Optimal discrimination between transient and permanent faults. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 214–223. IEEE, 1998.

[24] Doug Turnbull and Neil Alldrin. Failure prediction in hardware systems. Technical report, Tech. rep., University of California, San Diego. available at http://www. cs. ucsd. edu/˜ dturnbul/Papers/ServerPrediction. pdf, 2003.

[25] Joseph F Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado. Hard drive failure prediction using non-parametric statistical methods. In *Proceedings of ICANN/ICONIP*. Citeseer, 2003.

[26] Peter Bodik, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael Jordan, et al. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proc. Second International Conference on Autonomic Computing*, pages 89–100. IEEE, 2005.

[27] Ivano Irrera, Clever Pereira, and Marco Vieira. The time dimension in predicting failures: a case study. In *Sixth Latin-American Symposium on Dependable Computing (LADC)*, pages 86–91. IEEE, 2013.

[28] Sebastian Elbaum, Satya Kanduri, and A Amschler. Anomalies as precursors of field failures. In *14th International Symposium on Software Reliability Engineering*, pages 108–118. IEEE, 2003.

[29] Gordon F Hughes, Joseph F Murray, Kenneth Kreutz-Delgado, and Charles Elkan. Improved disk-drive failure warnings. *IEEE Transactions on Reliability*, 51(3):350–357, 2002.

[30] Amy Ward, Peter Glynn, and Kathy Richardson. Internet service performance failure detection. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):38–43, 1998.

[31] Sachin Garg, Aad Van Moorsel, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. A methodology for detection and estimation of software aging. In *Proceedings of The Ninth International Symposium on Software Reliability Engineering*, pages 283–292. IEEE, 1998.

[32] Fan-Tien Cheng, Shang-Lun Wu, Ping-Yen Tsai, Yun-Ta Chung, and Haw-Ching Yang. Application cluster service scheme for near-zero-downtime services. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 4062–4067. IEEE, 2005.

[33] Mark Shereshevsky, Jonathan Crowell, Bojan Cukic, Vijai Gandikota, and Yan Liu. Software aging and multifractality of memory resources. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003)*, page 721. IEEE, 2003.

[34] G Weiss. Timeweaver: A genetic algorithm for identifying predictive patterns in sequences of events. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 718–725. Citeseer, 1999.

[35] Gary Weiss. Predicting telecommunication equipment failures from sequences of network alarms. *Handbook of Knowledge Discovery and Data Mining*, pages 891–896, 2002.

[36] Ricardo Vilalta and Sheng Ma. Predicting rare events in temporal domains. In *IEEE International Conference on Data Mining (ICDM)*, pages 474–481. IEEE, 2002.

[37] Ting-Ting Y Lin and Daniel P Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, 39(4):419–432, 1990.

[38] Fares A Nassar and Dorothy M Andrews. *A methodology for analysis of failure prediction data*. Center for Reliable Computing, Computer Systems Laboratory, Depts. of Electrical Engineering and Computer Science, Stanford University, 1985.

[39] Ronjeet Lal and Gwan Choi. Error and failure analysis of a unix server. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 232–239. IEEE, 1998.

[40] Chokchai Leangsuksun, Tong Liu, Tirumala Rao, S Scott, and Richard Libby. A failure predictive and policy-based high availability strategy for linux high performance computing cluster. In *The 5th LCI International Conference on Linux Clusters: The HPC Revolution*, pages 18–20. Citeseer, 2004.

[41] Felix Salfner, Michael Schieschke, and Miroslaw Malek. Predicting failures of computer systems: A case study for a telecommunication system. In *20th International Symposium on Parallel and Distributed Processing*, pages 8–pp. IEEE, 2006.

[42] Felix Salfner and Miroslaw Malek. Using hidden semi-markov models for effective online failure prediction. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 161–174. IEEE, 2007.

[43] Carlotta Domeniconi, Chang-Shing Perng, Ricardo Vilalta, and Sheng Ma. A classification approach for prediction of target events in temporal sequences. In *Principles of Data Mining and Knowledge Discovery*, pages 125–137. Springer, 2002.

[44] Günther A Hoffmann, Felix Salfner, and Miroslaw Malek. Advanced failure prediction in complex software systems. 2004.

[45] Guenther Hoffmann and Miroslaw Malek. Call availability prediction in a telecommunication system: A data driven empirical approach. In *25th IEEE Symposium on Reliable Distributed Systems*, pages 83–95. IEEE, 2006.

[46] Ivano Irrera, Joao Duraes, Marco Vieira, and Henrique Madeira. Towards identifying the best variables for failure prediction using injection of realistic software faults. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 3–10. IEEE, 2010.

[47] Ivano Irrera, Joao Duraes, Henrique Madeira, and Marco Vieira. Assessing the impact of virtualization on the generation of failure prediction data. In *Sixth Latin-American Symposium on Dependable Computing (LADC)*, pages 92–97. IEEE, 2013.

[48] Ivano Irrera and Marco Vieira. Towards assessing representativeness of fault injection-generated failure data for online failure prediction. In *IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 75–80. IEEE, 2015.

[49] Ivano Irrera, Joao Duraes, and Marco Vieira. On the need for training failure prediction algorithms in evolving software systems. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 216–223. IEEE, 2014.

[50] Maria Rita Di Berardini, Henry Muccini, Andrea Polini, and Pengcheng Zhang. Online failure prediction of dynamically evolving systems. 2011.

[51] Valérie Issarny. *An Exception Handling Mechanism for Parallel Object-oriented Programming: Towards the Design of Reusable and Robust Distributed Software.* Inst. National de Recherche en Informatique et en Automatique, 1992.

[52] Shaula Yemini and Daniel M Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(2):214–243, 1985.

[53] Bruno Cabral. *A Transactional Model for Automatic Exception Handling.* PhD thesis, Universidade de Coimbra, 2009.

[54] Daniel Fernández Lanvin, Raúl Izquierdo Castanedo, Aquilino Adolfo Juan Fuente, and Alberto Manuel Fernández Álvarez. Extending object-oriented languages with backward error recovery integrated support. *Comput. Lang. Syst. Struct.*, 36(2):123–141, July 2010.

[55] Paris Buttfield-Addison, Jon Manning, and Tim Nugent. Learning swift. 2016.

[56] Lingli Zhang and Chandra Krintz. As-if-serial exception handling semantics for java futures. *Science of Computer Programming*, 74(5):314–332, 2009.

[57] Louis Gesbert, Frédéric Gava, Frédéric Loulergue, and Frédéric Dabrowski. Bulk synchronous parallel ml with exceptions. *Future Generation Computer Systems*, 26(3):486–490, 2010.

[58] Bruno Cabral and Paulo Marques. Implementing retry-featuring aop. In *Dependable Computing, 2009. LADC'09. Fourth Latin-American Symposium on*, pages 73–80. IEEE, 2009.

[59] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Notices*, volume 38, pages 388–402. ACM, 2003.

[60] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

[61] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP 2013–Object-Oriented Programming*, pages 302–326. Springer, 2013.

[62] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in action*. Manning Publications Co., 2015.

[63] Paul Hudak. Functional reactive programming. In *Programming Languages and Systems*, pages 1–1. Springer, 1999.

[64] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[65] Joao Paulo Magalhaes and Luis Moura Silva. Sho wa: a self-healing framework for web-based applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(1):4, 2015.

[66] Harald Psaier, Florian Skopik, Daniel Schall, and Schahram Dustdar. *Behavior monitoring in self-healing service-oriented systems*. Springer, 2011.

[67] Chris Schneider, Adam Barker, and Simon Dobson. A survey of self-healing systems frameworks. *Software: Practice and Experience*, 2014.

[68] Andres J Ramirez, David B Knoester, Betty HC Cheng, and Philip K McKinley. Plato: a genetic algorithm approach to run-time reconfiguration in autonomic computing systems. *Cluster Computing*, 14(3):229–244, 2011.

[69] Kwangmyeong Kim, Injun Choi, and Chulsoon Park. A rule-based approach to proactive exception handling in business processes. *Expert Systems with Applications*, 38(1):394–409, 2011.

[70] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[71] Thomas G Dietterich. Machine learning for sequential data: A review. In *Structural, syntactic, and statistical pattern recognition*, pages 15–30. Springer, 2002.

[72] Alan C Acock. Working with missing values. *Journal of Marriage and Family*, 67(4):1012–1028, 2005.

[73] Patrick Royston. Multiple imputation of missing values. *Stata Journal*, 4(3):227–41, 2004.

[74] Terry Speed. *Statistical analysis of gene expression microarray data*. CRC Press, 2004.

[75] Cristiana Amza, Anupam Chanda, Alan L Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Willy Zwaenepoel, Emmanuel Cecchet, and Julie Marguerite. Specification and implementation of dynamic web site benchmarks. In *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*, pages 3–13. IEEE, 2002.

[76] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web*, pages 276–286. ACM, 2004.

[77] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. *Internet Computing, IEEE*, 11(1):60–66, 2007.

[78] H2 database engine. (Last accessed 29h June, 2016).

[79] Shopizer. (Last accessed 15th April, 2016).

[80] Apache jmeter. (Last accessed 15th April, 2016).

[81] Weka project. (Last accessed 15th April, 2016).

[82] João Ricardo Lourenço, Bruno Cabral, and Jorge Bernardino. A predictive model for exception handling. In *Proc. of the 16th World Conference on Information Systems (WorldCIST)*, pages 741–750. Springer, 2016.

[83] João Ricardo Lourenço, Bruno Cabral, and Jorge Bernardino. Prex: A predictive model to prevent exceptions. In *Journal of Systems and Software*. (submitted), 2016.

[84] João Ricardo Lourenço, Bruno Cabral, and Jorge Bernardino. Predicting and preventing exceptions for increasing reliability. In *Advances in Knowledge and Information Software Management*. (submitted), 2016.

[85] Daniel Menasce et al. Tpc-w: A benchmark for e-commerce. *Internet Computing, IEEE*, 6(3):83–87, 2002.

[86] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[87] Saskia Le Cessie and Johannes C Van Houwelingen. Ridge estimators in logistic regression. *Applied statistics*, pages 191–201, 1992.

[88] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.