Practical Fault Detection and Diagnosis in Data Centers

Candidate
Claudio Ciccotelli
ID number 1161668

Thesis Advisor
Leonardo Querzoni

Reviewers
António Casimiro
Felicità Di Giandomenico

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering in Computer Science
December 2016
Thesis defended on 21 February 2017
in front of a Board of Examiners composed by:
Prof. Alfonso Gerevini (chairman)
Prof. Miriam Di Ianni
Prof. Laura Tarantino

Practical Fault Detection and Diagnosis in Data Centers
Ph.D. thesis. Sapienza – University of Rome

© 2016 Claudio Ciccotelli. All rights reserved

Version: February 19, 2017
Author’s email: ciccotelli@dis.uniroma1.it
Acknowledgments

First of all I want to thank my advisor, Prof. Leonardo Querzoni, for his support during my Ph.D. His valuable advice and careful guidance allowed me to complete such an important path in my life. It has been a pleasure working with you.

I want to thank Dr. Leonado Aniello and Dr. Luca Montanari. This thesis is also the result of your precious advice.

Thanks to Prof. Roberto Baldoni for letting me join the CIS research group, where I could grow in a stimulating environment with smart people. A genuine “thank you” goes to all Ph.D. students, postdocs, professors and all people involved in this research group who helped me in any way.

Finally, I want to thank my family. They have believed in me, supported me and encouraged me in every important step of my life, including this one. I owe everything to you.
Contents

1 Introduction 1
  1.1 Thesis Objective ........................................... 2
  1.2 Thesis Outline ........................................... 5

2 Background and Related Work 7
  2.1 Basic Concepts of Dependability .............................. 7
  2.2 Fault Management ........................................... 9
    2.2.1 Fault Detection and Diagnosis ......................... 9
  2.3 Evaluation Metrics ........................................ 11
    2.3.1 Detection Evaluation Metrics .......................... 11
    2.3.2 Diagnosis Evaluation Metrics .......................... 12
  2.4 Related Work ............................................. 13

3 Non-Intrusive and Black-Box Fault Detection and Diagnosis 23
  3.1 Intrusiveness .............................................. 23
  3.2 Application dependence .................................... 25
  3.3 Training Methodology ..................................... 26
  3.4 Related Work ............................................. 28
  3.5 Proposed Methodologies ................................... 29

4 NIRVANA: Non-intrusive Black-box Fault Diagnosis 31
  4.1 Methodology Overview ..................................... 31
  4.2 Architecture .............................................. 32
    4.2.1 Monitoring Subsystem .................................. 33
    4.2.2 Diagnosis Subsystem .................................. 33
  4.3 Experimental Evaluation .................................. 37
    4.3.1 Experiments ............................................ 37
    4.3.2 Settings and Implementation Details ................. 37
    4.3.3 Testbed ................................................ 38
    4.3.4 Workload .............................................. 38
    4.3.5 Faultload .............................................. 38
    4.3.6 Configuration and Training ............................ 39
    4.3.7 Results ............................................... 41
    4.3.8 Intrusive vs Non-intrusive Monitoring ............... 45
4.3.9 Performance in Multi-application Environments 46

5 NiTREC: Non-intrusive Black-box Anomaly Detection 49
5.1 Methodology Overview 49
5.2 Architecture 50
5.3 Experimental Evaluation 51
  5.3.1 Testbed and Dataset 51
  5.3.2 Inferential Engine 51
  5.3.3 Results 52

6 FLOW-NIBBLER 55
6.1 Methodology Overview 55
6.2 Model Discovery and Training 57
  6.2.1 Training Data 58
  6.2.2 Topology Graph 58
  6.2.3 Dependency Graph 61
  6.2.4 Flow Processor Models 64
6.3 Architecture 65
  6.3.1 Anomaly Detection 66
  6.3.2 Fault Diagnosis 69
  6.3.3 Parameters Tuning 75
6.4 Experimental Evaluation 78
  6.4.1 Experiments 79
  6.4.2 Testbed 79
  6.4.3 Workload 79
  6.4.4 Faultload 79
  6.4.5 Configuration and Training 80
  6.4.6 Results 82

II Reducing Monitoring Overhead 87

7 Invariants Space Filtering 89
7.1 Background and Related Work 89
7.2 Methodology Overview 91
7.3 Invariant Mining 92
7.4 Filtering 93
  7.4.1 Checking 94
  7.4.2 Filtering 1 94
  7.4.3 Filtering 2 95
7.5 Detection 96
7.6 Experimental Evaluation 96
  7.6.1 Testbed and Monitoring 97
  7.6.2 Workload 97
  7.6.3 Faultload 97
  7.6.4 Plan of experiments 98
  7.6.5 Results 98
<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conclusion</td>
<td>101</td>
</tr>
<tr>
<td>Bibliography</td>
<td>105</td>
</tr>
</tbody>
</table>
Notation and Acronyms

V(\mathcal{G}) \quad \text{Set of vertexes of graph } \mathcal{G}.

E(\mathcal{G}) \quad \text{Set of edges of graph } \mathcal{G}.

\{u, v\} \quad \text{Undirected edge between node } u \text{ and } v.

(u, v) \quad \text{Directed edge from node } u \text{ to node } v.

N_\mathcal{G}(v) \quad \text{Set of nodes adjacent to } v \text{ (neighborhood of node } v) \text{ in graph } \mathcal{G}.

\text{deg}_\mathcal{G}(v) \quad \text{Degree of node } v \text{ in graph } \mathcal{G}.

\text{deg}_\mathcal{G}^+(v) \quad \text{Out-degree of node } v \text{ in graph } \mathcal{G}.

\text{deg}_\mathcal{G}^+(X) \quad \text{Out-degree of } X \subseteq V(\mathcal{G}) \text{ in graph } \mathcal{G}.

\text{deg}_\mathcal{G}^+(X) = \left| \left\{ (u, v) \in E(\mathcal{G}) \mid u \in X \land v \notin X \right\} \right|.

\left( v \right)_i \quad i\text{-th element of vector } v.

\text{DAG} \quad \text{Directed Acyclic Graph.}

\text{SCC} \quad \text{Strongly Connected Component.}

\text{ANN} \quad \text{Artificial Neural Network.}

\text{SVM} \quad \text{Support Vector Machine.}

\textsuperscript{1}\text{When there is no ambiguity } \mathcal{G} \text{ is omitted.}

\textsuperscript{2}\text{This notation is used in place of the more common notation } v_i, \text{ when the definition of } v \text{ already contains a subscript.}
Chapter 1

Introduction

With the proliferation of cloud-computing and online services, such as e-commerce applications, search engines and social networking platforms, data centers have grown at a rapid pace and have become central to our daily lives. Service providers base their business on the availability and reliability of the services they provide. A service interruption or degradation implies an economic damage, including loss of revenues due to the unavailability of the services (e.g., e-commerce applications), reputational damage and penalties due to violations of service level agreements (e.g., XaaS providers). Given the business criticality of services provided by data centers, the cost of even a few minutes of downtime can be very high. In August 2013, Amazon.com and .ca went down for approximately half an hour. Forbes estimated the revenue loss due to the downtime to be $66,240 per minute, or a total of nearly $2 million, based on Amazon’s 2012 net sales. In the same month all Google services went down for about 5 minutes causing a 40% drop in worldwide web traffic and an estimated loss of $500,000. Besides these huge online service providers, a recent study, surveying 63 U.S. data centers from 15 different industry segments, estimated the mean cost per minute of a data center outage in $8,851, with average total cost of a data center outage that increased from $505,502 in 2010 to $740,357 in 2016. Apart from service interruptions, degraded performance can have important economic impacts as well. For example, Google reported 20% revenue loss due to an experiment that increased the time to display search results by 500 ms. Amazon reported a drop in 1% sales for an additional delay of 100 ms. Given the high cost incurred from even few minutes of downtime or slight performance degradation, ensuring high availability and reliability of services provided by data centers is of primary importance.

Availability is a quantitative attribute of dependability, measuring the ratio of uptime over lifetime, and is often expressed in terms of number of “nines”. For instance, five nines availability (0.99999) means that the service must be up at least 99.999% of the time. The high availability requirement commonly imposes five or even six nines availability. This translates into a maximum average downtime of about, respectively, 5.26 minutes or 31.5 seconds per year. Even less demanding requirements, such as three or four nines availability, imply an average maximum downtime of few minutes per week. Given such short time intervals to recover from a failure, the entire process must be automated. The fault management process
involves (i) the detection of the problem (e.g., a service failure), (ii) the identification of its root cause, that is, the fault, and (iii) the recovery, that is obtained by removing the fault from the system and restoring a correct service. The detection and the root-cause identification (often referred to as diagnosis) can take as much as 75\% of the total recovery time \cite{37}. Thus, a prompt detection and diagnosis of faults is essential to improve the data center availability and reliability. With the ever-growing complexity and scale of data centers, these tasks have become increasingly complex. Modern data centers are very large-scale heterogeneous systems that may comprise thousands to millions of computing, communication and storage components and host a wide variety of applications and services. Due to the complex interdependencies between hardware and software system components, faults often exhibit unexpected and unforeseen propagation paths, that make it difficult to trace a problem back to its cause. In such environments faults are unavoidable with the majority of problems imputed to software, misconfiguration and human errors, rather than hardware or network faults \cite{31,93,109}.

The monitoring and diagnosis systems currently employed in industry \cite{5,6,8,11} are ad hoc solutions for specific systems and applications and implement simple detection mechanisms, mainly based on performance thresholds or rules. Typically, such techniques require a fine grained tuning performed by an expert operator to avoid high false alarm rates. More general and sophisticated solutions have been proposed in the research \cite{72}. The majority of such approaches, however, are too intrusive, that is, they requires to install monitoring software in the target system or to instrument the operating systems or applications. This intrusive approach has several disadvantages. First of all it imposes a high burden on system operators in terms of management overhead due to the installation, configuration and maintenance of the monitoring system. Operators are often reluctant to introduce monitoring software into a production environment, especially if it may interfere with the monitored system \cite{35}. In some environments it is not possible to install third-party software in the monitored system, e.g., due to security and privacy policies or access restrictions. Finally, since the monitoring system is installed within the monitored system, faults originating in the latter may affect the monitoring system itself, causing degraded monitoring performance that may reduce the detection and diagnosis accuracy. Despite intrusiveness, many approaches proposed in the literature assume application-specific information (e.g., the presence of a middleware layer that allows to track end-to-end communications) that limits the applicability of the solution to specific contexts and applications. Intrusiveness and reliance on application-specific information both make these approaches less practical and more difficult to deploy in real systems. Therefore, the necessity of developing more practical and applicable solutions has been identified \cite{21,35,98,110}.

1.1 Thesis Objective

The focus of the thesis is the development of practical and applicable yet effective fault detection and diagnosis techniques. As already discussed, many aspects impact

\footnote{Throughout the whole thesis we will often refer to the monitored system as the target system, as it is common in the literature.}
the practicality and applicability of a detection and diagnosis approach. In this thesis we focus on two different aspects impacting the practicality and applicability, that we associate to two kinds of overhead introduced by the detection and diagnosis: the management and the monitoring overhead. With management overhead we refer to the additional burden put on system operators mainly due to the installation, configuration and maintenance of the diagnosis system. The monitoring overhead refers to the amount of additional resources (network bandwidth and computational) required to collect and process monitoring data.

The first part of the thesis focuses on reducing management overhead. We identified the application independence and the non-intrusiveness properties as two important requirements to reduce the management overhead of the diagnosis approach. Application dependent solutions typically need to be configured with application-specific information, which sometimes are difficult to obtain. Configuring and maintaining up-to-date such a knowledge base puts a considerable burden on operators and, thus, contributes to the management overhead of the monitoring system. Application independence can be realized through a black-box approach, that is, by considering the components of the monitored system as black-boxes that can be observed only from outside, without prior knowledge of their internal functioning. The agnosticism with respect to the applications running in the system and their internal functioning forces the development of application-independent solutions. As already discussed, also intrusiveness imposes a significant management overhead due to the installation, configuration and maintenance of the monitoring software, and, in general, reduces practicality and applicability of the approach. Another aspect of a detection and diagnosis technique that impacts the management overhead is the training. Indeed, the vast majority of the techniques rely upon some kind of initial training stage in which the target system is observed with the aim of, depending on the particular technique, discovering characteristics of it, learning or building a model of its behavior, or configuring and tuning parameters. The characteristic of the training process that most impacts management overhead is its reliance on fault injection [45]. Indeed, some techniques during the training stage need to observe how the system behaves when it is subject to faults. This is achieved by purposely injecting faults in the target system. As discussed in chapter 3 this strongly impacts the practicality and applicability of the approach and, particularly, management overhead, due to the need of performing an extensive fault injection campaign. Conversely, other techniques need to observe only the correct behavior of the system during the training stage, strongly reducing the impact on management overhead. Thus, in the first part of the thesis we focus our attention on detection and diagnosis techniques based on a black-box and non-intrusive approach. We first describe a methodology that relies on fault injection for the training stage. Then we present an alternative solution that does not rely on fault injection.

Many solutions proposed in the literature require to collect and process a huge amount of monitoring data. This incurs a non-negligible monitoring overhead in terms of computational and communication resources required to collect and process such large volume of data, which impacts the practicality and applicability of the approach. The second part of this thesis focuses on reducing monitoring overhead. In particular, we aim at reducing both computational and communication overhead by filtering the set of monitoring metrics, without reducing detection accuracy. To
this aim, in the context of invariant-based fault detection \cite{85,101,104}, we developed a filtering methodology of the invariant space which allows to considerably reduce the volume of monitoring data as well as the computational overhead to process it. Moreover, we show that this methodology also allows to improve fault detection accuracy.

**Contribution**

In this thesis we propose some methodologies to improve the practicality and applicability of detection and diagnosis solutions with respect to multiple aspects. Among the various aspects we focus particularly on alleviating and mitigating the management and monitoring overhead.

Summarizing the main contributions of the thesis are:

- **Reducing Management Overhead:**
  - The development of a black-box and non-intrusive detection and diagnosis methodology \cite{41}. Non-intrusiveness and application independence are the combination of properties that, in our opinion, allows to improve practicality and applicability the most, particularly by reducing management overhead. We compare our solution with an intrusive version of it, by letting the deployment of the monitoring system be intrusive, and we show that the intrusive solution does not provide significantly better accuracy. This methodology requires fault injection during the training stage.
  - The development of a black-box and non-intrusive detection and diagnosis methodology that does not rely on a fault injection based training. As already mentioned, a training that requires fault injection impacts the management overhead. To remove the reliance on fault injection, we propose a methodology that builds a model of only the correct behavior of the system. Such a model can be learned without observing the faulty behavior of the system, thus removing the need to perform fault injection during training.

- **Reducing Monitoring Overhead:**
  - The development of a general filtering strategy of the invariant space \cite{22}, applicable to invariant-based fault detection techniques, that reduces the monitoring overhead while improving the detection accuracy. Invariant-based techniques exploit stable relationships among monitoring metrics (invariants) that are expected to hold while the system is behaving correctly, to detect anomalous behaviors. Monitoring and processing the large number of invariants commonly found in large-scale complex systems incurs a significant monitoring overhead. We propose a methodology to filter the set of invariants with the aim of reducing the monitoring overhead, while improving detection accuracy by discarding useless, redundant and unstable invariants.
1.2 Thesis Outline

Chapter 2 introduces the basic concepts and the related work referred to both parts of the thesis. In chapter 3 we discuss three aspects that impact practicality and applicability, particularly in terms of management overhead: intrusiveness, application dependence and training methodology. We describe each aspect and how they impact practicality and applicability. Then, we report the related work specific to these topics and we clarify the position of our proposed methodologies with respect to the related literature along the two dimensions of intrusiveness and application dependence. In chapter 4 we present our first contribution: a methodology for fault detection and diagnosis based on a non-intrusive and black-box approach. The solution that we propose relies on fault injection for the training phase. In chapter 5 we present a simple black-box non-intrusive methodology for anomaly detection that does not require fault injection for the training phase. In chapter 6 we present the second contribution of this thesis. We further elaborate the methodology presented in the previous chapter, and we integrate it in a black-box non-intrusive fault detection and diagnosis methodology that does not require fault injection for the training. In chapter 7 we discuss our third contribution. In the context of invariant-based fault detection, we propose a methodology to filter the invariant space with the aim of reducing monitoring overhead, while improving detection accuracy. We first introduce a background on invariant analysis and the related work. Then, we present our methodology and, finally, we discuss the evaluation results.

This thesis is mainly based on [22, 25, 26, 11].
Chapter 2

Background and Related Work

This chapter presents the background and related work common to both parts of the thesis. Background and the literature specifically related to each part of the thesis will be presented in chapters 3 and 7 respectively.

2.1 Basic Concepts of Dependability

One of the first efforts to provide a precise definition of dependability and its basic concepts is due to the seminal work of Laprie [77], which dates back to 1985.

Computer system dependability is the quality of the delivered service such that reliance can justifiably be placed on this service.

Since then, the proposed taxonomy has been refined, expanded and simplified over the years [23,78], and an alternate definition of dependability has been formulated:

The dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable.

This definition involves terms and concepts that are rigorously defined [23]. This taxonomy provides an excellent unified way of expressing concepts in the context of dependability and, as of today, has been almost universally adopted. In the following we report the basic concepts and terminology that are referred to throughout the whole thesis.

A system is an entity which interacts with other entities (i.e., other systems). The function of a system is what the system is intended to do, while the behavior is what the system does to implement its function. The service delivered by a system is its behavior as it is perceived by its users, that is, the other systems that receive the service. When the service delivered by the system implements the intended function the service is said correct. A service failure occurs when the delivered service deviates from the correct service.

Dependability is an integrating concept encompassing the following attributes [23]:

- **Availability**: readiness for correct service.
- **Reliability**: continuity of correct service.
2. Background and Related Work

- **Safety**: absence of catastrophic consequences on the system’s users and the environment.
- **Integrity**: absence of improper system alterations.
- **Maintainability**: ability to undergo modifications and repairs.

The threats to dependability are failures, errors and faults. As already mentioned, a **failure**, or service failure, is an event that occurs when the service delivered by the system becomes incorrect. According to the definition of service, a deviation of the system’s behavior from the correct behavior corresponds to a failure only if it is perceived by the user (either a human or another system receiving the service). Thus, a misbehavior which is not perceivable by the user is not considered a service failure. A failure is caused by a deviation of the system’s state from a correct to an incorrect state, which is called an **error**. Note that an error may or may not lead to a failure. The adjudged or hypothesized cause of an error is called a **fault**. A fault is said active if it produces an error, otherwise it is dormant. An error can propagate in the system and result in other errors. When an error propagates to the system interface and causes the service provided by the system to deviate from correct service, it causes a failure. Finally, a service failure can cause a fault on the systems receiving the service. Figure 2.1 summarizes the relationship between faults, errors and failures. A fault may or may not become active and cause an error. The error may propagate (possibly causing other errors) until the system interface and produce a service failure. A failure, in turn, may cause a fault in the system receiving the failed service, and so on. In the example shown in figure 2.1 the fault in system B is **caused** by the failure at the interface between system A and B, and thus, indirectly, by the fault in system A. By following the arrows in the figure backward we would eventually reach an initial fault, which is called the **root cause**. Multiple faults may be concurrent, that is, they may overlap in time. Concurrent faults can be **related**, when they have a common cause, or **independent**, when they are attributed to different causes. For a thorough classification of faults, errors and failures see [23].

The means to support dependability can be categorized in:

- **Fault Prevention**: regards how to prevent the introduction or the occurrence of faults in the system.
- **Fault Tolerance**: regards how to ensure the correct service (i.e., avoid failures) despite the presence of faults.

![Figure 2.1. Relationship between faults, errors and failures.](image-url)
2.2 Fault Management

Fault management is the process of handling faults from their detection to their removal from the system, and is fundamental to support fault tolerance. It consists of three main steps:

- **Detection**: this step involves the determination of the presence of a fault in the system, typically by observing symptoms of an anomalous system behavior.

- **Diagnosis**: this step consists in the determination of the root-cause of the anomalous system behavior detected in the previous step.

- **Recovery**: this step concerns the removal of the fault or the faulty component, the system repair and the consequent restoration of the correct service.

The recovery step is out of the scope of this thesis, and will not be discussed in further details.

2.2.1 Fault Detection and Diagnosis

We stress the fact that in this thesis we are interested in online techniques for fault detection and diagnosis. Online means that the detection and diagnosis are performed on the basis of the current observation of the system obtained through monitoring. Offline techniques, such as “post-mortem” analysis are out of the scope of the thesis and will not be discussed. With respect to detection we should further distinguish between fault and failure detection. Even though the two terms, fault detection and failure detection, are sometimes used as synonyms in the literature, they have actually different meanings. Clearly, since a failure is always caused by a fault, the detection of a failure obviously implies the detection of the presence of a fault. The contrary, however, is not true. A fault not necessarily causes a failure, as already discussed in section \[2.1\]. The presence of a fault is not always detected by observing a service failure. The detection may be consequent to the observation of symptoms of an anomalous, yet correct, system behavior. Another term which is sometimes used in the literature is *anomaly detection* \[36\]. This has a broader meaning than failure and fault detection and concerns a wide variety of applications including fraud detection, intrusion detection and of course fault and failure detection. An anomaly represents a pattern that does not conform to the expected behavior. It may correspond to a wide range of misbehaviors, including a failure, or a symptom of the presence of a fault. For this reason, we will sometimes refer to anomaly detection in the thesis.
Fault diagnosis (sometimes referred to as root-cause analysis) is a process that is typically triggered after the detection process. The task of the diagnosis process is to find the root cause of the problem, that is, the fault that caused the anomaly (e.g., a misbehavior, or a failure) revealed by the detection process. Depending on the level of detail of the diagnosis this may comprise the localization of the fault, that is, the component of the system that is affected by the fault, and the identification of the type of the fault.

Fault detection and diagnosis are reactive approaches. They are triggered after the problem (e.g., a fault or a failure) has already occurred. Failure prediction represents a proactive approach towards the management of failures (but not faults). The task of failure prediction is to hypothesize the occurrence of a failure in the near future on the basis of the current observation of the system. Figure 2.2 highlights the differences between fault/failure detection, fault diagnosis and failure prediction. In particular it should be noted that failure prediction represents a proactive approach with respect to failure detection, but is an orthogonal process with respect to fault diagnosis. The difference between the two processes is pointed out in figure 2.2. They not only look to different directions of time, but, more importantly, they look...
2.3 Evaluation Metrics

In this section we provide a definition of the metrics commonly used to evaluate the quality of the detection and diagnosis of a particular technique.

2.3.1 Detection Evaluation Metrics

In order to assess the quality of the detection provided by a particular technique, a number of well-established evaluation metrics are used by the community \[100\]. These are the same metrics commonly used in the evaluation of classification techniques, namely accuracy, precision, recall, F1-score and false positive rate. Indeed, the problem of fault detection, is a binary classification problem. The system state can be either correct (i.e., the system is currently fault-free) or faulty (i.e., the system is affected by a fault). The absence of a fault in the system is regarded as a negative, while the presence of a fault corresponds to a positive. The detection task involves raising an alert whenever the presence of a fault is discovered. This leads to four possible cases:

- **True Positive**: the detector raises an alert and the system is actually in a faulty state. Thus, a true positive corresponds to a correct detection of a faulty state.

- **False Positive**: the detector raises an alert but the system is in a correct state. Thus, a false positive corresponds to a false alarm.

- **False Negative**: the detector does not raise an alert but the system is in a faulty state. Thus, a false negative represents an undetected faulty state.

- **True Negative**: the detector does not raise an alert and the system is in a correct state. Thus, a true negative corresponds to a proper classification of the system state as correct.

The above-mentioned evaluation metrics can be defined in terms of the number of true positives $TP$, false positives $FP$, false negatives $FN$ and true negatives $TN$. The sum of true positives and false negatives $P = TP + FN$ is the total number of positives, while $N = TN + FP$ is the total number of negatives. The sum of false positives and false negatives $FP + FN$ gives the total number of misclassified instances, while $TP + TN$ gives the number of correctly classified instances. Finally, $n = TP + TN + FP + FN$ is the total number of instances.

Accuracy measures the fraction of correctly classified instances over the whole set of instances:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{n}$$

Precision (also known as positive predictive value) measures the fraction of true positives over the instances classified as positive:

$$\text{precision} = \frac{TP}{TP + FP}$$
In the context of fault detection it is the ratio of correctly detected faults to raised alerts.

Recall (also known as true positive rate or sensitivity) measures the fraction of true positives over the set of all positives.

\[
\text{recall} = \frac{TP}{TP + FN} = \frac{TP}{P}
\]

In the context of fault detection it corresponds to the ratio of correctly detected faults to the total number of actual faulty states.

When tuning the parameters of a classification algorithm with the aim of improving accuracy, there is, typically, a trade-off between precision and recall. Indeed, improving precision, that is, reducing the number of false positives, typically, results in increasing the number of false negatives, which reduces recall, and vice versa. The F1-score (sometimes referred to as F-score or F-measure) is a good measure of the trade-off between precision and recall:

\[
F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}
\]

Precision and recall do not account for true negatives. False positive rate measures the fraction of false positives over the set of all negatives:

\[
\text{FPR} = \frac{FP}{FP + TN} = \frac{FP}{N}
\]

In the context of fault detection, this corresponds to the fraction of false alarms over the set of non faulty instances.

### 2.3.2 Diagnosis Evaluation Metrics

When a faulty state is detected, the task of the diagnosis is to identify the root cause, or, in general, the set of root causes, of all the anomalies observed in the system. The diagnosis, usually, consists in a set \(D\) of faulty components. Given a diagnosis \(D\) and the actual set of root causes \(R\) (i.e., the actual set of faulty components), we can define the number of true/false negatives and positives as follows:

- \(TP = |D \cap R|\)
- \(FP = |D \setminus R|\)
- \(FN = |R \setminus D|\)
- \(TN = |\overline{D} \cup \overline{R}|\)

where the notation \(\overline{A}\) means the complement of set \(A\). Note that in this context there is typically a very unbalanced proportion of positives \(R\) and negatives \(\overline{R}\), with \(|\overline{R}| \gg |R|\). Indeed, the number of simultaneously faulty components is usually much smaller than the total number of components. For this reason, it does not make much sense to compute the accuracy and the false positive rate metrics. Indeed, even a naive diagnoser that always outputs the empty set \(D = \emptyset\) would achieve a
false positive rate of 0 and a quite good accuracy, as long as the number of positives is much smaller than the number of negatives. Therefore, the diagnosis is usually evaluated only in terms of precision, recall and F1-score. The precision of a single diagnosis \( D \) can be computed as \( |D \cap R| / |D| \), and corresponds to the fraction of correctly identified root causes in the issued diagnosis. The recall of a single diagnosis \( D \) can be computed as \( |D \cap R| / |R| \). It corresponds to the fraction of correctly identified root causes in the set of actual root causes, or equivalently, 1 minus the fraction of unidentified root causes: \( 1 - |R \setminus D| / |R| \). To compute this metrics over a set of experiments, we consider the set of all diagnosis \( \{ D_1, \ldots, D_n \} \) issued during the experiments, and the corresponding actual root causes \( \{ R_1, \ldots, R_n \} \), so that \( R_i \) is the actual set of root causes at the time the diagnosis \( D_i \) was issued. Then, precision and recall can be computed as \[ \text{precision} = \frac{1}{n} \sum_{i=1}^{n} \frac{|D_i \cap R_i|}{|D_i|} \]
\[ \text{recall} = \frac{1}{n} \sum_{i=1}^{n} \frac{|D_i \cap R_i|}{|R_i|} \]
That is, they are computed as the mean value of the precision and recall of the single diagnosis. The \( F1\text{-score} \) can be computed from precision and recall as defined in section 2.3.1.

2.4 Related Work

Fault detection and diagnosis are very broad topics and the related literature is vast. A variety of techniques have been developed over the years drawing from many different areas such as statistics, graph theory, artificial intelligence and machine learning. In this section we discuss the main techniques and we review some of the most representative works. Table 2.1, at the end of this section, summarizes the classification of the related works according to the employed techniques, the context in which they are used, and whether they are used for fault diagnosis, failure prediction or performance problems and bottlenecks identification. More complete surveys can be found in [36, 61, 72, 100, 108]. We recall that in this section we review the related work common to both parts of the thesis, while a discussion of the works more specifically related to each part of the thesis is deferred to the dedicated sections in the respective parts.

Heuristics and threshold-based This category comprises all those techniques based on heuristics and thresholds. For example count-and-threshold techniques [20, 33, 81, 102] maintain counters for the occurrence of specific events in a given time frame and trigger alerts when one or more counters exceeds a given threshold. Such techniques are common for detecting hardware failures. However, they are often too simple to cope with the complexity of modern distributed systems, and their accuracy strongly depends on the optimal tuning of parameters (e.g., thresholds). For example in [81] a technique named Dispersion Frame Technique (DFT) is applied to system error logs. It is based on a set of 5 rules that fire when the inter-arrival
time of specific events in a given time frame is below a threshold. For example the 2-in-1 rule fires when the inter-arrival of two failures is less than one hour.

Tiresias [118] exploits DFT [81] with a reconfigured set of rules to predict failures in distributed systems. However, Tiresias cannot distinguish between different failures and therefore does not attempt to diagnose the source of the problem.

**Rule-based** Rule-based techniques rely on a set of predefined rules often expressed in the form of *if-then* statements, where the *if* part expresses a condition over a set of monitoring metrics (premises) and the *then* part contains the conclusion (e.g., if CPU utilization is high and L2 cache miss rate is low, then there is a busy loop). The set of rules may be derived empirically by observing cause-effects relations in the systems, or may be based on expert knowledge codified as a set of rules. Some rule-based techniques exploit machine learning algorithms to automatically derive the set of rules. When multiple rules in the rule-base fire techniques such as forward chaining inference are used to determine the most probable cause of the problems. The main advantage of rule-based techniques is that rules are often human-interpretable and thus easily understandable by operators and easily extensible with new rules tailored to the specific system. However, rule-based solutions are characterized by the inability to diagnose problems that are not expressed in the rule-base and the difficulty of maintaining an updated rule-base [79].

Chopstix [32] monitors low-level OS metrics as vital signs of the monitored system and considers abnormal values as symptoms of some problem. To diagnose the problem it uses a set of 10 rules that map one or more symptoms into a high level problem. Some of these rules are intuitive, other have been derived from repeated observations of cause-effect. The authors believe these rules to be general enough to be applied to a number of systems. However, these rules can be applied only to diagnose individual node problems, they can not diagnose problems that propagate across a network.

Systems based on a large set of rules, need sophisticated techniques to diagnose problems when multiple rules fire. JECTOR [82] describes a language for expressing rules that capture the timing relationship among correlated events and presents a prototype implementation.

Yemini et al. [119] represent a set of rules using codebooks, i.e., dependency matrices in which columns are associated with problems and rows with symptoms. They diagnose a problem by identifying the column with the closest match to the observed symptoms.

Due to their simplicity and ease of use, rule-based techniques are implemented by many commercial tools, such as IBM Tivoli Enterprise Console [6], HP Operation Managers [5] and EMC Ionix Codebook Correlation [46].

**Model-based** Model-based techniques are based on a mathematical description of the system that is used to model the dynamics of the system. Some techniques model the normal behavior of the system, and detect problems when the observed system behavior deviates significantly from the behavior predicted by the model. Queuing models, for example, are used to describe the relation between components and resource consumption. They detect a fault whenever the relations of the
model are violated. In [115] and [83] multi-tier Internet applications are modeled as network of queues and mean-value analysis is used to predict response times. They assume a closed queuing model in which the number of clients in the system remains constant. Cherkasova et al. [40] adopt an open queue model, in which clients can join/leave the system, to describe the relationship between CPU usage and transaction response times. Moreover they exploit regression to model the application through a performance signature which is used to distinguish anomalies from application updates and workload changes. These systems are targeted at performance anomalies and bottlenecks detection, rather than fault diagnosis.

Other model-based methodologies exploit graph-theoretic techniques to build fault propagation models. Examples of such models include dependency graphs and causality graph [108]. Dependency graphs, typically, have nodes corresponding to system components and directed edges expressing the dependencies between them. Nodes are often labeled with apriori failure probabilities, while directed edges \((A, B)\) are labeled with conditional probabilities, that is the probability of a failure on component \(B\), provided that component \(A\) fails. Causality graphs model the causal relation between events in the system. Nodes correspond to individual events (e.g., symptoms and faults) and directed edges express the causal relation between events. Also in this case the graph is often enriched with probabilities. Nodes can be labeled with the apriori probabilities of event occurrence, while directed edges can be labeled with conditional probabilities of the causal implication. Fault propagation models are used to track detected problems back to their root causes by analyzing the most probable paths of fault propagation with some inference technique, such as Bayesian inference. This task is known to be NP-hard in general [71], therefore, frequently the characteristics of the fault propagation model must be constrained [57, 67, 107], or assumptions are made on the maximum number of concurrent faults [24, 67]. The main limitation of model-based techniques is that they require a quite accurate description of the monitored system, which is not always easy to obtain. In general they require a deep understanding of the system.

Shrink [67] diagnoses faults in IP networks by modeling the system as a bipartite graph annotated with probabilities. A greedy algorithm determines the root-cause by computing the probabilities of fault propagation. Sherlock [24] automatically builds the dependency graph, by discovering the components and determining the dependencies. However it is not a general purpose diagnosis technique, it can only determine problems that affect end-users in network-based services systems. Both Shrink and Sherlock are based on exponential-time algorithms. However, they scale by assuming that there can only be a small number of concurrent problems in the system.

NetMedic [68] continuously updates nodes and edges weights of its dependency graph based on recent history of monitoring data. The most probable root-causes, with respect to a component affected by some anomaly, are determined as those that have paths of highest impact to that component in the dependency graph.

FChain [90] complements a fault propagation pattern technique based on timing with the automatic dependency graph discovery of Sherlock to localize faults in cloud systems.
Statistical Techniques in this category are based on the analysis of monitoring data with statistical tests and techniques such as correlation analysis, regression analysis, histogram comparison. Statistical techniques can be categorized into two broad categories: parametric and non-parametric. Parametric techniques assume that monitoring data is drawn from a known distribution (e.g., gaussian-based detection). These techniques, typically, detect anomalous system behaviors by searching for outliers in monitoring data. Non-parametric techniques assume that monitoring data is drawn from an unknown distribution. Correlation is often used to determine, e.g., the relation between monitored metrics and causal relation between system components.

Agarwal et al. [18] address problem determination through the change point detection technique. They detect abrupt changes of system behavior by monitoring changes to the mean value of monitored metrics over consecutive time frames. They exploit this technique to build problem signatures as combination of changes (or absence of changes) in different metrics.

SALSA [111] diagnoses problems in large clusters through histogram comparison of performance counters computed on logs. Shen et al. [105] diagnose performance problems through a reference-driven approach. They detect anomalous reference-to-target changes and exploit histogram comparison to determine the set of parameters that best explain the observed performance deviations.

WAP5 [96] collects network traces at each node and exploits message correlation algorithms to discover causal paths for performance debugging in wide-area systems.

A particular class of statistical techniques is based on invariant analysis [48,50,66,85,99,101,104]. Invariants are stable relationships between monitoring metrics that hold while the system behaves correctly. A broken relation suggests the presence of an anomalous system behavior. Invariant-based techniques will be further discussed in chapter 7.

The main advantage of statistical techniques is that they require little expert knowledge and details about the system. The main drawback is that their effectiveness strongly depends on the validity of the assumptions made on data (such as distributions, parameters, etc.)

Machine learning based Machine learning based techniques exploit machine learning algorithms to automatically build a model of the system out of training data. Machine learning techniques can be classified into three broad categories: supervised, unsupervised and semi-supervised learning. Supervised learning techniques require that the learning algorithm is provided with examples of input and the corresponding desired output of the model to be learned. The input of the model are, typically, monitoring metrics representing the current system behavior. The output depends on the particular technique. Regression-based techniques try to learn the correct behavior of the system with respect to a set of output monitoring metrics as a function of the input metrics. In this case an anomalous behavior is detected when the current behavior of the system deviates significantly from the behavior predicted by the regression model. Classification algorithms assign a class (e.g., “correct”, “faulty”) to the input of the model. The training dataset is labeled with the classes associated with each instance. Unsupervised learning techniques do not require
labeled training datasets. For example clustering-based techniques group input data into clusters based solely on some similarity measure or statistical properties. Semi-supervised techniques comprise hybrid approaches that combine the advantages of both supervised and unsupervised learning. These techniques assume that only a portion of the training dataset is labeled. The main advantage of machine learning based techniques is that they automatically build a model of the behavior of the system, which, otherwise, would require expert knowledge to be obtained. The main drawbacks of such approaches are that they typically require a lot of training data, and may suffer from the curse of dimensionality when the number of features is large (which can be sometimes mitigated through feature selection).

Pinpoint [38] exploits clustering to discover sets of components that are highly correlated with failed requests. MinEntropy [122] exploits decision trees for diagnosis in large Internet sites. It trains decision trees on request traces that contain failed requests. Leaves represent failed or successful requests, while paths from root to leaf represent root-causes. In order to diagnose problems, paths through the tree are ranked according to their degree of correlation with failures. Kircman and Fox [73] exploit probabilistic context free grammars (PCFG) for detection of application-level failures in Internet services, and use decision trees for localization. Yuan et al. [120] use multi-class support vector machines (SVM) to learn signatures of known problems. Signatures are represented through N-grams of system call sequences. However, this technique is suitable only for individual node diagnosis, it cannot be used to diagnose problems in distributed systems. In [63] and [62] support vector machines (SVM) are used to predict failures, respectively, on a single machine and on a web application server. Moreover, [62] addresses the problem of re-training in evolving systems, which is a problem common to many machine learning based techniques.

With respect to this categorization the solutions proposed in this thesis belong to several categories. NIRVANA (chapter 4) and NiTREC (chapter 5) are mainly based on a machine learning approach. NIRVANA exploits support vector machines to classify the system state with respect to monitoring data, while NiTREC employs artificial neural networks to model the behavior of the system. FLOW-NIBBLER (chapter 6) is realized with a hybrid approach which combines rule-based, graph-based and machine learning techniques to detect anomalies and track them back to their root causes. Finally chapter 7 presents a methodology to filter the invariant space in the context of invariant-based fault detection. Thus, it is based on invariant analysis, which is a kind of statistical technique.
Table 2.1. Classification of the main related works according to the employed techniques, the context in which they are used, and whether they are used for fault diagnosis, failure prediction or performance problems and bottlenecks identification.

<table>
<thead>
<tr>
<th>Work</th>
<th>Technique</th>
<th>Context</th>
<th>Fault Diagnosis</th>
<th>Failure Prediction</th>
<th>Performance Problems and Bottlenecks Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heuristics and Threshold-based</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bondavalli et al. [33]</td>
<td>Count-and-threshold</td>
<td>Legacy systems and COTS components</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serafini et al. [102]</td>
<td>Count-and-threshold</td>
<td>Distributed Systems</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lin and Siewiorek [81]</td>
<td>DFT</td>
<td>Hard Failures</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Tiresias [118]</td>
<td>DFT</td>
<td>Distributed Systems</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Rule-based</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chopstix [32]</td>
<td>Small set of rules</td>
<td>Distributed Systems</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JECTOR [82]</td>
<td>Specification language for expressing rules</td>
<td>Network Management Systems</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Yemini et al. [119]</td>
<td>Codebook</td>
<td>Network Management Systems</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹Events correlation to support fault diagnosis.
<table>
<thead>
<tr>
<th></th>
<th>Methodology</th>
<th>Field</th>
<th>Application Area</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model-based</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Urgaonkar et al. [115]</td>
<td>Queueing model</td>
<td>Multi-tier internet services</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Liu et al. [83]</td>
<td>Queueing model</td>
<td>Multi-tier internet services</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Cherkasova et al. [40]</td>
<td>Queueing model</td>
<td>Distributed Systems</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Katzela and Schwartz [71]</td>
<td>Dependency Graph</td>
<td>Communication Networks</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Hasan et al. [57]</td>
<td>Causality Graph</td>
<td>Network Management Systems</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Shrink [67]</td>
<td>Bayesian Networks</td>
<td>IP Networks</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Steinder and Sethi [107]</td>
<td>Causality Graph</td>
<td>Communication Networks</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Sherlock [23], NetMedic [68]</td>
<td>Dependency Graph</td>
<td>Large Enterprise Networks</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>FChain [90]</td>
<td>Dependency Graph</td>
<td>Cloud Systems</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Statistical</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agarwal et al. [18]</td>
<td>Change Point Detection</td>
<td>Enterprise Middleware Systems</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SALSA [111]</td>
<td>Histogram Comparison</td>
<td>Large Clusters</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Shen et al. [105]</td>
<td>Histogram Comparison</td>
<td>Complex Systems</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>WAP5 [95]</td>
<td>Correlation</td>
<td>Wide-area Systems</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Machine Learning</td>
<td>Pinpoint [38]</td>
<td>Clustering</td>
<td>Large Internet Services</td>
<td>✓</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>------------</td>
<td>-------------------------</td>
<td>---</td>
</tr>
<tr>
<td>MinEntropy [121]</td>
<td>Decision Trees</td>
<td>Large Internet Services</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Kiciman and Fox [73]</td>
<td>Probabilistic Context-free Grammars, Decision Trees</td>
<td>Component-based Internet Services</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Yuan et al. [120]</td>
<td>SVM</td>
<td>Individual Nodes</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Irrera et al. [63]</td>
<td>SVM</td>
<td>Individual Nodes</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Irrera et al. [62]</td>
<td>SVM</td>
<td>Evolving Software Systems</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

²Only on individual nodes.
Part I

Reducing Management Overhead
Chapter 3

Non-Intrusive and Black-Box Fault Detection and Diagnosis

This chapter discusses three factors that impact the practicality and applicability of fault detection and diagnosis methodologies. They affect several aspects related to practicality and applicability. Particularly the management overhead. These factors are the level of intrusiveness of the monitoring system, the application dependence of the methodology (i.e., the degree of dependence on application-specific information) and the training methodology. In the next sections we discuss all these factors. In section 3.4 we review the related work and, finally, in section 3.5 we categorize the related work and our solutions with respect to the three factors.

3.1 Intrusiveness

There is no single definition of non-intrusiveness in the context of computer systems monitoring. Many works found in the literature use the term non-intrusive in a vague way, without providing a formal definition, but rather relying on the common understanding of the term, which is not formal though. In some works the term non-intrusive is even used as a synonym of black-box. To the best of our knowledge the only works that give a precise definition of non-intrusiveness are \cite{27,88} in the context of failure prediction.

The characteristics more commonly associated with a non-intrusive monitoring system which are also consistent with the definition of non-intrusiveness provided by \cite{27,88}, are:

- no need to install monitoring software in the target system;
- no instrumentation of the operating system or application code;
- no active probing;
- no need to log into monitored hosts/components.

Therefore, we adopt the following definition of non-intrusiveness:

**Definition 3.1.1** (Non-intrusiveness). A non-intrusive monitoring system is one based on passive-probing that does not require to install software in the monitored
Non-Intrusive and Black-Box Fault Detection and Diagnosis

system, nor to instrument the operating system or applications, and that does not require to log into the monitored components.

Besides this definition, that allows to draw a sharp demarcation line between an intrusive and a non-intrusive monitoring approach, we should recognize that a monitoring system may implement different levels of intrusiveness. The level of intrusiveness mainly depends on the deployment of monitoring probes. Figure 3.1 shows how the level of intrusiveness increases as we move the deployment of the monitoring system at different architectural levels. At the highest architectural level, the entire data center, the monitoring system typically consists of some hardware sensors that measure very coarse-grained monitoring data, such as aisle temperature, humidity, power consumption of the entire data center, etc. At a rack-level hardware probes can be deployed, such as network sniffer and power meters to collect monitoring data from an entire server rack. At the virtualization level it is possible to collect monitoring data, for example, by querying the hypervisor through a known API. At a lower architectural level probes must be installed inside the operating system of the physical or virtual machines. In this setting monitoring agents and tools share resources with target system, with the possibility of interfere with it. An increased level of intrusiveness is realized by instrumenting the operating systems or applications to collect, e.g., application-specific monitoring data. The more we move towards a lower architectural level, the higher is the intrusiveness level of the monitoring system. At the same time, the more we move from a low to an high architectural level, the more monitoring data become unavailable (some metrics cannot be monitored) and aggregated. According to definition 3.1.1 we should put a line dividing a monitoring that can be defined as “non-intrusive” from a one that should be considered “intrusive” approximatively between a rack-level deployment and a hypervisor-level deployment.
Intrusive vs Non-intrusive Monitoring

An intrusive approach to monitoring has several disadvantages that impact the practicality and applicability of the solution:

- **Management overhead**: Installation, configuration and maintenance of probes within each component is a complex and human intensive task entailing costs that can become prohibitive as the size and complexity of the data center increase.

- **Policy compliance**: Some application scenarios do not allow the execution of third-party software (like software probes) within the monitored components; this may be due to security and privacy policies that restrict the possibility to install software on a secured machine (e.g., in mission-critical systems) or maybe to the fact that the component to be monitored runs on proprietary software and thus represents a “closed-box” (e.g., special purpose appliances for storage, anti-spam devices). Finally, installing software probes may be simply impractical from a business perspective (e.g., an IaaS provider should not install application-level probes on provided VMs, as the end-user only will control those machines [110]).

- **Degraded monitoring performance**: When components experience some kind of faults like CPU over-utilization, software probes may be affected as well [110]. In such cases the monitoring infrastructure may experience degraded performance as readings from those components may be delayed, corrupted or even dropped, hampering the possibility to quickly and correctly identify the faults.

- **Interference with the target system**: Since monitoring probes are installed within the components of the target system and share the same resources, there is the possibility that the monitoring system interferes with the target system.

A non-intrusive monitoring approach mitigates these drawbacks by not relying on installing software on the monitored components, nor on instrumenting the OS or the applications of the target system. Instead it relies on a much smaller set of hardware probes such as network sniffers, power meters, temperature sensors, etc., that passively monitor the components of the system from outside.

On the other hand, as already mentioned, a non-intrusive monitoring typically results in a reduced set of monitoring metrics and a higher level of data aggregation that might hamper the diagnosis task and reduce accuracy.

### 3.2 Application dependence

The application dependence of a fault detection and diagnosis methodology represents the degree of dependence from application-specific information. Being dependent on application-specific information includes, for example, relying on the number and type of applications running in the system, relying on the use of particular application level protocols (of the ISO/OSI protocol stack) or relying on the presence...
of a middleware. In the literature, techniques are commonly classified into three broad categories according to their degree of dependence from application-specific information. **White-box** approaches are the most application dependent. Techniques in such category exploit application-specific information to perform the detection and diagnosis context and are therefore tailored to specific applications. They rely on assumptions on the application running in the target system, that constraint the applicability of the solution to specific contexts. A **black-box** approach conversely is one that does not rely on any application-specific information. It is completely agnostic with respect to the application running in the system. The components of the system are treated as black-boxes that can be observed from outside, but nothing is known about their internal functioning. This is the least level of application dependence. A **gray-box** approach represents an intermediate level of application dependence between the white- and black-box approaches. Gray-box techniques rely on few application-specific information. They typically make little assumptions about the application running in the system, such as, for example, the assumption that they use a middleware to communicate.

**White-box vs Gray-box vs Black-box Approach**

With respect to the practicality and applicability the black-box approach represents certainly the most appealing approach. Both white- and gray-box approaches severely impact the applicability. Indeed, as already discussed, relying on application-specific information relegates the solutions to specific contexts. Moreover, application-specific information is sometimes difficult to obtain and maintain up-to-date. White-box and gray-box approaches often require extensive configuration from system operators, which impacts management overhead. A black-box approach mitigates these drawbacks. Indeed, application independence results in more reusable solutions, applicable to a wider range of contexts. Moreover, black-box approaches do not require operators to provide detailed information about the system applications, thus relieving the impact on management overhead.

On the other hand application-specific information could prove very useful to diagnose specific kind of faults. Indeed, relying on application-specific information can aid the diagnosis process in identifying faults that are peculiar to a particular application. In this respect, the diagnosis process of black-box approaches might suffer from this lack of information.

### 3.3 Training Methodology

Many detection and diagnosis techniques, especially those based on machine learning, need a preliminary training phase to build a model of the system. Some techniques aim to model the correct behavior of the system only, and consider every deviation from the normal behavior as anomalous. Conversely, other techniques model both the correct behavior of the system and faults propagation. In the first case the training data contain only instances related to the correct behavior of the system. Thus, training data can be obtained by simply collecting a sufficient amount of data during normal operation of the system. We will refer to this training methodology as **correct behavior based**. In the second case, instead, training data must contain
both instances related to the correct behavior of the system and instances related to the behavior of the system when affected by faults. To collect such a training set one could in principle monitor the target system and just wait for faults to “naturally” occur. This solution, however, is not practical at all. Indeed, it makes the duration of the training phase indeterminate and possibly too long, without guarantees that all possible faults will occur in a given time frame. Thus, this kind of training is typically performed through fault injection \[45,60,89\], that is by deliberating injecting faults in the system. We will refer to this training methodology as **faulty behavior based**.

**Correct vs Faulty Behavior Based Training**

Both approaches present advantages and drawbacks. Obviously, the faulty behavior based training methodology impacts the practicality and applicability of the approach for several reasons:

- **Management overhead**: the fault injection campaign imposes a considerable burden on operators. They must identify the set of faults to be injected, prepare and perform the injection campaign. The larger the set of faults the methodology is required to diagnose, the more onerous the burden imposed by the fault injection campaign.

- **Need for testing environment**: in many scenarios it is not possible to inject faults directly in the production environment, as this may expose the system to unbearable risks. In such cases, the presence of a testing environment in which performing the injection campaign would be necessary. The testing environment should be identical in all aspects to the production one and subject to the same workload. This aspect reduces both the practicality and the applicability of the solution adopting this kind of training.

- **Diagnosis of unforeseen problems**: diagnosis methodologies that adopt a faulty behavior based training are, typically, very accurate in diagnosing a predefined set of known faults, that is, the ones they have been trained to recognize through fault injection. However, such methodologies can be much less accurate in diagnosing unforeseen faults (i.e., faults not observed during the training), or may even be unable to. This aspect may limit the applicability of the approach to the context of recurrent faults detection.

Conversely, a correct behavior based training needs only to collect monitoring data when the system is operating correctly. Therefore, it has low impact on management overhead and can be performed directly on the production environment. Moreover, techniques based on learning a model of the normal behavior of the system, in principle, are able to detect any problem (including unforeseen problems) as long as it causes the behavior of the system to deviate from the learned correct behavior. On the other hand, correct behavior based training works well with systems that exhibit a stable behavior, but not so well with systems that exhibit a highly dynamic behavior. Moreover, a faulty behavior based training can provide valuable information about the behavior of the system when subject to faults, which a correct behavior based training cannot. In some contexts, such information may
allow to improve significantly the detection and diagnosis accuracy. For example, in chapter 7 we present a technique which considerably improves detection accuracy by relying on a faulty behavior based training.

3.4 Related Work

In this section we categorize the related works with respect to intrusiveness and application dependence. In section 2.4 we already reported the main related works in the wide context of detection and diagnosis. The majority of these works [24, 67, 68, 73, 111] are both very application dependent (white-box) and intrusive.

In the context of performance analysis and debugging in distributed systems a few solutions [21, 29, 38] have been proposed that adopt gray-box and black-box approaches and different levels of intrusiveness. Project 5 [21] provides performance debugging in local area networks by discovering causal paths between components without a priori knowledge of the application. It accomplishes this task by collecting network traces in a passive and non-intrusive way, and through two algorithms to discover causal paths, one based on a gray-box approach and one based on a black-box approach. However, Project 5 differs significantly from the solutions proposed in this thesis as it addresses offline analysis and targets performance analysis and debugging, while we address online fault detection and diagnosis. WAP5 [96] extends the work of Project 5 to address performance debugging in wide-area network. It is a gray-box approach like Project 5, and it is also intrusive, as it requires to install software on the monitored components. Similar works are [38] and [29]. However, they are based on an intrusive and only partially application independent approach as they make minimal application-specific assumptions (gray-box approach). Pinpoint [38] detects problems in dynamic Internet services. It assumes the presence of a middleware, to be able to trace requests (thus implementing a gray-box approach) and requires instrumentation (therefore it is very intrusive).

E2EProof [19] supports performance analysis with a black-box approach. However, it requires instrumentation of the kernel. EbAT [117] is an anomaly detector for cloud systems. It is based on sensors that collect data at each hierarchical level of the cloud infrastructure. Raw monitoring data is processed so as to produce a stream of entropy time series for each component in the hierarchy. The entropy time series are then analyzed with known methods, such as spike detection, subspace methods or signal processing, to identify anomalies at a hierarchical and global level. EbAT does not require knowledge about the applications running in the cloud system. Therefore, it implements a black-box approach, however it is intrusive. Tiresias [118] provides black-box failure prediction in distributed systems. However, it relies on software probes installed in the system and therefore is intrusive. Moreover, as already mentioned, it does not address diagnosis. ALERT [112] addresses anomaly prediction in large-scale hosting infrastructures with a black-box approach. FChain [90] (already discussed in section 2.4) is also based on a black-box approach. However, both works are based on intrusive monitoring.

Spanoudakis et al. [106] and Tripathy et al. [76] apply non-intrusive approaches to the monitoring of service-based systems. However, both solutions are not black-box as they are based on a complete description of the monitored system. Moreover, they
provide monitoring frameworks, but do not address diagnosis. Suneja et al. \cite{110} propose a non-intrusive and gray-box approach for system monitoring in cloud systems. They do not address detection and diagnosis.

CASPER \cite{27} models the target system as a Hidden Markov Model and performs online failure prediction with a non-intrusive and black-box approach. However, CASPER is different from the solutions proposed in this thesis, in that it targets failure prediction, while our solutions target fault detection and diagnosis (see section \ref{sec:related_work}). Moreover, CASPER needs a faulty behavior based training.

### 3.5 Proposed Methodologies

Figure \ref{fig:methodologies} shows a categorization of the related works along the two directions of intrusiveness and application dependence. We distinguish six categories corresponding to the combination of three increasing levels of application dependence (black-box, gray-box and white-box approach) with two classes partitioning the approaches in non-intrusive and intrusive. The majority of works proposed in the literature adopt an approach which is both intrusive and white-box (i.e., application-specific). A few solutions are based on a non-intrusive white-box approach, as discussed in section \ref{sec:non-intrusive}. However, they are mainly focused on the collection of monitoring data and do not address detection or diagnosis. Some works are based on a gray-box approach, they assume some application-specific information, such as the presence of a middleware to track requests across the system components. A number of solutions that implement a black-box approach have also been proposed. The majority of

![Figure 3.2. Categorization of the proposed methodologies and the related work with respect to intrusiveness and application dependence.](image)
them adopts an intrusive approach. In figure 3.2 we have highlighted the position of the methodologies proposed in this part of the thesis (NIRVANA, NiTREC and FLOW-NIBBLER) with respect to this categorization. All our proposed solutions are based on a non-intrusive and black-box approach, as it is the combination that introduces the lower management overhead. To the best of our knowledge only two related works fall in this category, CASPER [27] and Project5 [21]. However, as already mentioned, these works differ from our solutions. Indeed, Project5 only targets offline performance analysis, while in this thesis we focus on online techniques for fault detection and diagnosis. CASPER addresses failure prediction, which is orthogonal to fault diagnosis (see section 2.2.1).

Figure 3.3 categorizes the three proposed methodologies with respect to detection, diagnosis and training methodology. NIRVANA and FLOW-NIBBLER address both detection and diagnosis of faults, while NiTREC addresses anomaly detection only. With respect to diagnosis, NIRVANA provides both localization of the fault, and identification of the type of the fault, while FLOW-NIBBLER provides localization only. NIRVANA relies on a faulty behavior based training, obtained through fault injection, while NiTREC and FLOW-NIBBLER adopt a correct behavior based training methodology, that is, they need to observe only the correct behavior of the system during the training stage.
Chapter 4

NIRVANA: Non-intrusive Black-box Fault Diagnosis

In this chapter we present NIRVANA, a black-box and non-intrusive fault detection and diagnosis system. In section 4.1 we give an overview of the approach, in section 4.2 we present NIRVANA’s architecture and, finally, in section 4.3 we discuss the experimental evaluation.

4.1 Methodology Overview

Our methodology performs detection and diagnosis by classifying the current system behavior as either correct, or one of several possible faulty states. We characterize the target system behavior with a set of features that we extract from monitoring data (e.g., packet rate, RTT, power consumption). In particular, we aim to capture the behavior of the system with respect to the services hosted in it and also to the system as a whole. Thus, for each service in the system we extract a set of service features from the corresponding monitoring data. Such metrics characterize the behavior of the single service. Moreover, we extract some global features from monitoring data aggregated at the whole system level (e.g., rack-level, in case of a rack-level deployment). The set of all such metrics extracted at a given time \( t \) forms a vector that characterizes the behavior of the system at time \( t \) with respect to its services and as a whole. These feature vectors represent a snapshot, or a fingerprint, of the target system behavior at a given time. NIRVANA computes a new feature vector every \( \tau \) time units (for example, in our experiments \( \tau = 1s \)).

Since our methodology is application agnostic, the services hosted in the system must be discovered automatically by NIRVANA, without relying on application-specific information (such as analysis of application-level protocols). In particular, NIRVANA is agnostic with respect to the specific application providing such services. They are considered as black-boxes, identified by a network address and port number, whose internal functioning is unknown. We discuss the methodology to discover such services in a non-intrusive and black-box way in section 4.2.2.

Thus, at any given time \( t \), NIRVANA classifies the current system behavior on the basis of the feature vector computed at time \( t \). To this aim we employ a multi-class classifier. In our experiments we used support vector machines (SVM).
The classes of the classifier corresponds to the set of possible states of the system $S = \{c\} \cup F$, where $c$ corresponds to the correct system state, and $F = \{f_1, \ldots, f_p\}$ to the set of possible faulty states. A faulty state is a combination of fault type and localization (e.g., crash of a given service/application, wrong setting of a service parameter in a given configuration file, etc.) The number and types of faulty states in $F$ depend on the particular target system. In general, the determination of $F$ requires the intervention of an expert operator.

Once the set $S$ has been determined, NIRVANA must be trained to classify the feature vectors in one of the possible classes in $S$. The training set must contain instances for any class corresponding to the states in $S$. To collect such a training set, we perform a fault injection campaign [45], so as to observe the system in any state in $S$. The training set instances consist in the feature vectors extracted during the training stage labeled with the corresponding system state.

After the training stage, NIRVANA continuously gathers monitoring data from the target system, computes the feature vectors and classifies the current behavior. Whenever a feature vector is classified with one of the faulty states, NIRVANA raises an alert and outputs the diagnosis (i.e., the faulty state).

### 4.2 Architecture

In this section we present the architecture of NIRVANA. Figure 4.1 shows a high level overview of the architecture. It consists of two main parts: the monitoring subsystem and the diagnosis subsystem.

The monitoring subsystem is composed of a set of hardware probes deployed in the target system that continuously extract monitoring data from the system components. All these streams of raw monitoring data are sent to the diagnosis subsystem that processes them to assess the correctness of the system behavior. Whenever an incorrect system state is detected the diagnoser component raises an alert and outputs the diagnosis.
4.2 Architecture

The general architecture described so far is not necessarily non-intrusive. The non-intrusiveness requirement is fulfilled by NIRVANA by imposing a non-intrusive deployment of the monitoring subsystem (see section 4.2.1). In our experimental evaluation we compared our prototype implementation of NIRVANA, with an intrusive version of it (intr-NIRVANA), which is implemented by letting the deployment of the monitoring subsystem be intrusive. The results of this comparison are presented in section 4.3. In the subsequent sections the two subsystems are discussed in details.

4.2.1 Monitoring Subsystem

The monitoring subsystem is realized by means of a set of monitoring probes deployed in the target system. In general, a probe can be either a monitoring software installed in a component of the target system (e.g., in the guest OS of a virtual machine) or a hardware equipment (e.g., network sniffers, power meters, sensors, etc.). Each probe monitors a component, or a set of components, of the target system and sends monitoring data to the diagnosis subsystem. Therefore, each probe produces a stream of raw monitoring data, i.e., unprocessed data. Raw monitoring data can be collected at different architectural levels (e.g., rack, hypervisor, virtual machine, application, etc.) As already discussed in section 3.1, a deployment at a higher architectural level allows for a less intrusive solution at the price of a more coarse-grained monitoring. Conversely, a deployment at a lower architectural level allows for a more fine-grained monitoring at the expense of intrusiveness. To realize NIRVANA we opted for a rack-level deployment, which enables a non-intrusive solution, still providing monitoring data at a sufficient granularity for an effective diagnosis. In particular, we decided to monitor network traffic and power consumption, two types of data that can be easily obtained with a non-intrusive approach. We deployed a set of hardware network probes directly attached to the network switches of the rack to capture network traffic, and a set of smart-PDUs to collect power consumption data. All these probes are passive and connected through a switch external to the network of the data center so as to not introduce additional traffic in it. They must be synchronized, e.g., through NTP [14], and send raw monitoring data to the diagnosis subsystem. This latter can be deployed on a machine possibly external to the system (for example, one of the probes can serve the purpose). An instance of NIRVANA must be deployed for each rack to monitor.

4.2.2 Diagnosis Subsystem

Figure 4.2 presents the architecture of the diagnosis subsystem. The diagnosis is performed in four stages:

1. **Input**: where raw monitoring data streams are merged into a single global stream and split into a set of per-service data streams.

2. **Preprocessing**: where features are extracted from each stream of raw monitoring data (either service or global level) so as to produce feature vectors.

3. **Classification**: where feature vectors are classified. Rather than a single class the classifier provides a probability distribution $P_S$ over the set of possible states $S$. 
4. Output: where on the basis of $P_S$, the system decides whether to raise an alert and output the diagnosis.

In the following we will discuss the configuration and training of NIRVANA. Then, we will describe each stage of the diagnosis in more deep details.

Configuration and Training

As already mentioned, NIRVANA needs a preliminary configuration and training stage, before entering the operational state. This stage consists of three steps: services discovery, training data collection, feature selection and training. In the following we will describe each task.

Services Discovery — As the name suggests, this step is devoted to the discovery of the services running in the system. To accomplish such a task with a black-box approach, we do not rely of any application specific information (e.g., by analyzing application-level protocols). We start from the simple observation that service providers, typically, always listen on the same port number, while client components typically get a new port number for every new connection. Thus, if we monitor the network for a sufficiently long time, counting the number of messages exchanged by each network address and port pair, we should observe a relatively small number of address and port pairs with a large number of exchanged packets (corresponding to services), and a majority of network address and port pairs with a significantly smaller number of exchanged packets (corresponding to client connections). Thus, this statistics can be used to cluster these two classes. In particular, we observed that services correspond to outliers in the whole distribution of exchanged packets. Thus, in NIRVANA we use the Grubbs’ test [58] for outlier detection to automatically identify these services. We opted for this method as it does not require to set parameters (as they are derived directly from data), therefore easing the configuration process.

Thus, the output of this task is a set of services, in the form of network address and port pairs. This information is used to split monitoring data into a set of per-service streams. No application-specific information about these services is exploited by NIRVANA.
4.2 Architecture

Training Data Collection — In this step we collect the dataset used to train NIRVANA’s classifier. As already mentioned in section 4.1, the training set must contain instances for each possible class, that is, for each system state in $S$. We build such a dataset through a fault injection campaign in which we inject different faults in the system so as to observe each state in $S$. During this period we collect monitoring data and we extract the feature vectors. Each feature vector is relative to a given instant in time $t$ and is labeled with the state of the system at time $t$. The dataset is then split into a training set and a validation set, so that each set has a balanced proportion of feature vectors associated to each system state (to avoid bias towards a particular system state). Typically 60 to 80% of instances are used for the training set, and remaining 40 to 20% for the test set. When the amount of training data is limited, $k$-fold cross validation [74] can be used during training, instead of splitting the dataset.

Feature Selection and Training — Each feature vector consists of a set of service features, which are computed for each service, and a set of global features. As service features we consider the packet rate, bit rate, number of open connections, number of active connections, average, minimum and maximum round trip time (RTT). As global features we compute the packet rate of the whole traffic and the power consumption. Therefore, the complete set of features consists of $7n + 2$ features, where $n$ is the number of services discovered in the previous stage. It is well understood that choosing the appropriate set of features is fundamental to improve the accuracy of a classifier [55]. Indeed, some features may be redundant or irrelevant for the classification task, and may worsen its accuracy. Moreover, reducing the feature space allows to decrease training time and computational complexity. Determining the optimal set of features, that is, the set that allows to maximize the classifier accuracy, is a well-known problem and many solutions have been proposed [55]. NIRVANA adopts a wrapper approach [75], that is, it uses the classifier itself to assess the relative usefulness of subsets of features. Given a subset of features, the classifier is trained on that subset of features. Then, the accuracy of the classifier is evaluated on the validation set. The optimal subset of features is the one yielding to the most accurate classifier. Finding the optimum is obviously an NP-hard problem. A wide range of search strategies exists. We employ two alternative strategies: forward selection and backward elimination [55]. Each search strategy returns a subset of features and we choose the one that yields the best accuracy. The corresponding trained classifier is deployed in NIRVANA.

As already stated in [62, 63, 64, 73], using machine learning techniques for fault detection could require periodical retraining to cope with the unavoidable evolving nature of software systems in data centers. This is an open problem in literature, and existing solutions simply settle for detecting when it is time to issue a retraining or for using a continuous online learning. In this work we did not examine this aspect in depth, and leave the integration and assessment of existing solutions as a possible future work. Let us finally remark that configuration and training must be done on a per-rack basis and must be repeated every time the rack configuration is changed (e.g., new machines or applications are added/removed) or the set of faults to be detected is enlarged.
Fault Diagnosis Stages

Input Stage — In the input stage the streams of raw monitoring data coming from the monitoring subsystem enter the raw data splitter, which recombines the input streams into \( n \) service level streams, each one containing monitoring data related to a given service, and one global stream containing all monitoring data. Since all probes are synchronized and attach a timestamp to each piece of monitoring data, this recombination can be performed easily.

Preprocessing Stage — Each stream of raw monitoring data output by the previous stage enters a distinct feature extractor component (see figure 4.2). There is a service level feature extractor for each service and one global level feature extractor. These components extract from the streams of monitoring data the set of features determined during the configuration and training stage. The combination of the output of all feature extractors at any given time \( t \) represents the feature vector computed at time \( t \). A new feature vector is computed every \( \tau \) time units (in our experiments \( \tau = 1s \)). For example, assume one of the features determined during the configuration phase is the packet rate of a given service \( s \). For each sampling interval \([t, t + \tau)\) we count the number of packets exchanged by service \( s \) (by analyzing the streams of raw monitoring data related to service \( s \)) and we divide the count by the number of seconds in \( \tau \). Analogously, we compute all other features. The set of all features computed in the interval \([t, t + \tau)\) forms the feature vector computed at time \( t \). Since streams of monitoring data extracted from complex systems are, by their nature, often characterized by sudden and unexpected fluctuations and outliers, the feature calculators use moving averages computed over the last \( W \) samples of the input raw monitoring data streams to smooth the input values before feature calculation. The tuning of parameter \( W \) depends on the characteristics of the observed system. In our experiments we found that choosing \( W = 10 \) yielded good results (see section 4.3).

Classification Stage — In this stage the feature vectors produced in the previous stage are fed to the multi-class classifier. A feature vector computed at time \( t \) represents a snapshot of the system behavior at time \( t \). The multi-class classifier tries to infer the current system state on the basis of the current system behavior, by classifying the feature vector in one of the possible system states in \( S \). Actually, rather than outputting a single class, the classifier outputs a probability distribution \( P_S \) over the set of states in \( S \). Since it is fed with one feature vector every \( \tau \) time units, this stage outputs a new probability distribution every \( \tau \) time units.

Output stage — The fault detector and recognizer component (FDR) implements the output stage. On the basis of the probability distribution \( P_S \) output by the classifier in the previous stage, it decides whether to trigger any alert. Once an alert is triggered, the FDR enters in an alert state until an operator resets it by sending an alert reset signal, which indicates that the fault condition has been managed. An alert is triggered whenever the most probable system state according to \( P_S \) is a faulty state. In such case the FDR component also outputs the ranking of possible system states according to the probability distribution \( P_S \). We say that the FDR...
has detected a fault whenever it triggers an alert and the monitored system actually is in a faulty state. We say that the FDR has recognized a faulty state \( f \) whenever it triggers an alert, the top ranked state in the fault ranking list is \( f \), and the target system is in the faulty state \( f \).

4.3 Experimental Evaluation

This section reports the experimental evaluations we carried out to test and assess our methodology. We present the results obtained in three experimental campaigns. In the first experimental campaign we evaluated the detection and diagnosis accuracy of NIRVANA in a typical three-tier architecture. The results are presented in section 4.3.7. In the second experimental campaign we compared NIRVANA with an intrusive version of it, that we call intr-NIRVANA (see section 4.3.8). Finally, in the third experimental campaign, we evaluated the accuracy of NIRVANA in a multi-application environment with a larger set of applications and virtual machines (see section 4.3.9). Before presenting the results, we give some details about the experiments, the testbed, the workload and the injected faults.

4.3.1 Experiments

To evaluate our methodology we performed several experiments. The first set of experiments is devoted to the collection of the dataset used in the configuration and training stage of NIRVANA (see section 4.2.2). We perform a set of experiments for each possible state in \( S = \{c\} \cup F \). In the experiments relative to the correct system state \( c \) we do not inject any fault (golden runs). In the experiments relative to a given faulty state \( f \in F \) we start with a correct state and then after a variable amount of time, we inject the fault that induces the faulty state \( f \). In the experiments we also vary the workload. In particular, in our testbed we identified three ranges of workload (see section 4.3.4). For each system state we performed a run on each of the three workload ranges. The dataset that we collect consists of the raw monitoring data labeled with the corresponding system state.

In the second set of experiments, we collected the dataset on which we evaluated the detection and diagnosis accuracy of NIRVANA. We performed this set of experiments with the same methodology of the training experiments, but with different workloads (see section 4.3.4).

4.3.2 Settings and Implementation Details

The only two parameters of NIRVANA are the sampling interval \( \tau \) (see section 4.1) and the smoothing window \( W \) (see Fault Diagnosis Stages in section 4.2.2). Due to technical constraints, in our experimental deployment we set the sampling interval to the minimum possible of 1 second. This implies that NIRVANA produces a feature vector every second and, thus, the classifier outputs a probability distribution at the same rate. For what concerns the smoothing performed in the preprocessing stage, we employed a window of \( W = 10 \) samples, meaning that the value of each feature is actually computed as the average of the last \( W \) samples for that feature.

As machine learning method for the multi-class classifier we used support vector
machines (SVM) [59]. In particular, we employed C-Support Vector Classification (C-SVC) with probability estimation and a radial basis function (RBF) as kernel function.

4.3.3 Testbed

Our testbed is composed of a rack with 4 blade servers, each equipped with two Intel Xeon X5560 Quad-Core CPUs clocked at 2.28 GHz and with 24 GB of RAM. On these blades we deployed a standard 3-tier architecture. The interface tier is based on an Apache 2 Web Server (WS) [1] and mod_cluster 2.6.0 [9], an HTTP-based load balancer. The business tier includes a JBoss AS 7.1.1.Final [7] cluster running three JBoss instances. The storage layer consists of a database server (MySQL 5.5.38 [10]). On top of the JBoss cluster we deployed TicketMonster [12], a well-known e-commerce web application benchmark that models an online booking service.

4.3.4 Workload

We generated synthetic workloads using Tsung 1.5.0 [13] an open-source multi-protocol distributed load testing tool able to simulate a variable multi-user load. The workload consists in the number of requests per second sent to the web application. Each request involves the generation of $\sim 400$ packets in the testbed.

For the training experiments we generated workloads following a normal distribution. We varied mean and standard deviation to span three ranges of load levels high, medium and low. During a preliminary analysis, we identified the high level as the one that uses almost all the resources of the system. The medium and low levels are chosen so that they are, respectively, $2/3$ and $1/3$ of the high level.

For the evaluation experiments, we generated workloads according to the Pareto ON-OFF model [28]. The workload is generated from a physical machine external to the testbed.

4.3.5 Faultload

In our experiments we injected different types of faults. According to the taxonomy of Avižienis et al. [23], some of them are misconfiguration faults, that is, they are caused by a wrong configuration of some application parameters. Configuration errors are both common and highly detrimental, and diagnosing them is desirable [121]. The misconfiguration faults that we injected cause performance degradation of some components of the system. The other faults that we injected consist in an abnormal usage of resources and a node crash.

- **SQL misconfiguration**: we reduce the size of the connection pool used by the JBoss instances to communicate with the database server. The heavy contention for available connections impacts negatively on application performance and creates a bottleneck towards the database server. In particular, in our experiments we reduced the maximum size of the connection pool from the default value of 20 to a value of 1. We refer to the system state in presence of this fault as $f_{\text{SQL}}$. 
4.3 Experimental Evaluation

- **AJP misconfiguration**: we reduce the size of the thread pool associated with the AJP protocol (which allows the communication between the web server and the JBoss instances). Similarly to the previous fault, this misconfiguration creates a performance bottleneck in the communication between the web server and the JBoss instances. In our experiments we reduced the size from the default value of 50 to a value of 15. We refer to the system state in presence of this fault as \( f_{AJP} \).

- **Disk stress**: we cause an abnormal disk access activity on the server hosting the SQL server. This fault increases disk contention causing performance degradation of the SQL service. We refer to the system state in presence of this fault as \( f_{DSK} \).

- **JBoss node crash**: we simulate the crash of a JBoss instance by killing the corresponding process. This fault causes the load balancer to redistribute the workload among the remaining JBoss instances, which causes a slight degradation of performance. We refer to the system state in presence of this fault as \( f_{CRS} \).

The set of possible system states is \( S = \{ c, f_{SQL}, f_{AJP}, f_{DSK}, f_{CRS} \} \).

4.3.6 Configuration and Training

To train NIRVANA we followed the steps described in section 4.2.2 (“Configuration and Training”). First of all, in the services discovery step, we used the dataset collected during the training experiments (see section 4.3.1) to discover the set of services, with the methodology described in section 4.2.2. To this purpose, we used only the part of the dataset collected during golden runs (that is experiments performed without injecting faults in the system).

In our experimental setup, we automatically identified 5 such services. The first service is associated with the Apache web server, the second service is associated with the MySQL server, and the remaining three services are associated with the AJP connections between each of the three JBoss slaves and Apache. Note that this services are simply network address and port pairs to NIRVANA, which is completely agnostic with respect to the application providing these services. For each service we considered the 7 service level features reported in table 4.1 together with two global level features.

In the next step of the configuration and training stage we built the training set for the multi-class classifier by computing the feature vectors on the dataset collected during the training experiments. Each feature vector has been labeled with the corresponding system state.

Then we performed the feature selection and training stage. As described in section 4.2.2, we applied wrapper approach with two alternative search strategies: forward selection and backward elimination [55]. Between the two subsets of features, provided by these strategies, we chose the one giving better performance in terms of accuracy. In our experiments we chose the subset of features selected by forward selection as it provided slightly better performance with a smaller set of features, as shown by the comparison of the ROC curves in Figure 4.3. The final set of selected
Table 4.1. Features description.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global_PPS</td>
<td>Aggregate packet rate.</td>
</tr>
<tr>
<td>Global_Power</td>
<td>Aggregate power consumed by the rack.</td>
</tr>
<tr>
<td>ServiceX_Open_Conn</td>
<td>Number of open connections for service X.</td>
</tr>
<tr>
<td>ServiceX_Active_Conn</td>
<td>Number of active connections for service X.</td>
</tr>
<tr>
<td>ServiceX_PPS</td>
<td>Packet rate computed for service X.</td>
</tr>
<tr>
<td>ServiceX_BPS</td>
<td>Bits per second computed for service X.</td>
</tr>
<tr>
<td>ServiceX_RTT_Avg</td>
<td>Average round trip time computed for service X.</td>
</tr>
<tr>
<td>ServiceX_RTT_Min</td>
<td>Minimum round trip time computed for service X.</td>
</tr>
<tr>
<td>ServiceX_RTT_Max</td>
<td>Maximum round trip time computed for service X.</td>
</tr>
</tbody>
</table>

Figure 4.3. Comparison of ROC curves of classifiers using different sets of features.

Table 4.2. Features selected by forward selection.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Features selected by forward selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global_PPS</td>
<td>Service2_Active_Conn</td>
</tr>
<tr>
<td>Global_Power</td>
<td>Service2_Open_Conn</td>
</tr>
<tr>
<td>Service1_Open_Conn</td>
<td>Service2_RTT_Max</td>
</tr>
<tr>
<td>Service1_RTT_Avg</td>
<td>Service3_RTT_Avg</td>
</tr>
<tr>
<td>Service1_RTT_Min</td>
<td>Service4_RTT_Avg</td>
</tr>
<tr>
<td>Service1_RTT_Max</td>
<td>Service4_RTT_Max</td>
</tr>
</tbody>
</table>

features is listed in Table 4.2. The multi-class classifier trained with this set of features is the one we used in the experiments.
4.3 Experimental Evaluation

4.3.7 Results

In this section we report the results obtained on the evaluation experiments. Figure 4.4 shows an example of NIRVANA in action in an experiments in which we injected a fault of type crash (we caused the crash of one of the JBoss instances). For ease of presentation, the figure shows only a small portion of the experiment to focus on the injection and the subsequent detection and recognition of the faulty state. The chart in the figure reports the probability distribution $P_S$ output by the multi-class classifier over time. The topmost bar reports the actual state of the target system over time: it starts fault-free and then after some time (approximately at second 33 in the portion of the experiment shown in figure), one of the JBoss slaves crashes. The lowermost bar shows how NIRVANA classifies the state of the target system over time. This corresponds to the most probable system state according to the probability distribution $P_S$. A little more than 10 seconds after the injection, the probability associated with the correct state $c$ quickly drops, and the most probable state turns to $f_{AJP}$. In this case, NIRVANA has detected a transition from a correct state to a faulty one. However, the most probable state does not correspond to the actual state. Thus, NIRVANA has not output a correct diagnosis yet. Just after a few seconds, the most probable system state becomes $f_{CRS}$, which corresponds to the actual system state and, thus, to a correct diagnosis. The reason of the initial misclassification of NIRVANA is due to the fact that after the injection of a fault the transition from the correct state to a faulty state is usually not very sharp. During the transition, depending on the dynamics of the particular target system, different faults may exhibit similar effects on the system behavior.

With respect to the experiment shown in figure 4.4 the crash of the JBoss instance causes a significant drop in the number of active connections toward the SQL server. The AJP misconfiguration fault induces a similar effect. Indeed, it causes a bottleneck in the AJP communications between the web server and JBoss instances, which in turn produces a significant drop in the number of active connections toward the SQL server (due to the slowdown introduced by the bottleneck). The most likely reason why such change in the SQL traffic is associated with the $f_{AJP}$ state rather than to the $f_{SQL}$ lies in the latter fault being characterized by a much larger variation in the number of active SQL connections. Then, few seconds after the
injection the AJP traffic is rebalanced among the remaining JBoss instances, which makes the number of active SQL connections get back to its normal trend. Moreover, the remaining two JBoss instances cannot handle the same request rate and the HTTP request latency increases. An increased number of enqueued HTTP requests translates to a larger number of open HTTP connections. These two factors make the faulty state $f_{\text{CRS}}$ recognizable by the classifier.

The interdependencies between the features describing the behavior of the system can be very complex, and a clear understanding of how a given fault affects these features can be very difficult. This is where machine learning techniques come into play to enable automatic learning of such interdependencies. The following results highlight noteworthy numbers in support of the effectiveness of the proposed approach.

We first present the results concerning NIRVANA’s fault detection capabilities (that is the recognition of a faulty state regardless of the particular fault). The dataset that we used to evaluate NIRVANA consists of the feature vectors computed on the monitoring data collected during the experiments. Each feature vector has been labeled with the corresponding system state. During the evaluation NIRVANA assigns a system state (the most probable according to $P_S$) to each feature vector. The system state $\hat{s} \in S$ assigned by the classifier to a feature vector is compared to the label $s \in S$ associated with the feature vector. This leads to four possible cases:

True Negative: $\hat{s} = s = c$;

True Positive: $\hat{s} \neq c \land s \neq c$;

False Negative: $\hat{s} = c \land s \neq c$;

False Positive: $\hat{s} \neq c \land s = c$;

We evaluated the detection performance according to the well-known evaluation metrics that we discussed in section 2.3.1. These metrics are accuracy, precision, recall, F1-score and false positive rate, and are defined in terms of the number of true/false positives and negatives.

Figure 4.5 and figure 4.6 report the values of the evaluation metrics. Each bar of the histograms represents the value of an evaluation metric computed over the set of experiments in which a particular fault has been injected. The two figures show that NIRVANA provided a quite accurate detection. The accuracy is very high and never drops below 0.93 for each set of experiments. Precision represents the fraction of feature vectors classified as faulty that were actually associated with a faulty state, and NIRVANA proved to be perfectly precise for all faulty states but $f_{\text{DSK}}$, for which it anyway achieved an extremely high precision of more than 0.95. Recall is the fraction of feature vectors associated with faulty states correctly detected by NIRVANA. The recall values reported in figure 4.5 show that really few faulty states have been wrongly classified as correct by NIRVANA. As expected, the high precision and recall result in a high F1-score. A very important aspect in fault detection is that the detector does not raise too much false alarms. This aspect is captured by the false positive rate which is shown in figure 4.6. In all experiments NIRVANA achieved a false positive rate less than 0.08, and, in particular, in the
Figure 4.5. Fault detection performance of NIRVANA. Each bar of the histograms is relative to the set of experiments in which a particular fault has been injected, except for the bar labeled as “Correct”, which is relative to fault-free experiments.

Figure 4.6. False positive rate for each set of experiments.
experiments in which we injected the misconfiguration faults and the JBoss crash fault NIRVANA did not produce false positives.

Besides the detection, we also evaluated the diagnosis, in terms of NIRVANA’s capability of recognizing the system state. In particular, we evaluated the recognition accuracy, that is, the fraction of recognized system states. Given a feature vector, with associated system state \( s \), we say that NIRVANA has recognized the system state \( s \) if the most probable system state \( \hat{s} \) output by the classifier, when fed with the feature vector, is such that \( \hat{s} = s \). Figure 4.7 shows the recognition accuracy with respect to each state in \( S \). That is, each bar of the histogram represents the accuracy of NIRVANA in recognizing a particular system state. The figure shows that NIRVANA has achieved high recognition accuracy for all states in \( S \).

Another important aspect to consider when evaluating a fault detection and diagnosis methodology is the time it takes to spot the fault. The detection latency is the elapsed time from the injection of a fault and its detection. We also evaluated the recognition latency, that is, the elapsed time from the injection of a fault and the proper recognition of the faulty state. For example, with respect to figure 4.4, the detection latency is the elapsed time from the injection to the classification of the state as \( f_{\text{AJP}} \). The recognition latency, instead is the elapsed time from the injection to the classification of the state as \( f_{\text{CRS}} \).

Figure 4.8 reports the mean values of the detection and recognition latencies obtained in the set of experiments in which we injected a given fault, where the error bars represent the standard deviation. The results show that NIRVANA detected the two misconfiguration faults immediately, while for the other injected faults the detection latency is in the order of 10 seconds. The recognition latency follows the same trend, but reaching the order of the tens of seconds to recognize the disk stress fault. This difference is due to the fact that some faults (misconfiguration faults in this case) causes a very sharp transition from a correct to a faulty state, while other faults causes a more smooth transition.
4.3 Experimental Evaluation

4.3.8 Intrusive vs Non-intrusive Monitoring

In this section we present the results of the comparison between NIRVANA and an intrusive version of it, that we call intr-NIRVANA. We show that intr-NIRVANA does not improve significantly the detection and diagnosis performance of NIRVANA.

The diagnosis subsystem of intr-NIRVANA is the same as that of NIRVANA. The only difference between NIRVANA and intr-NIRVANA lies in the deployment of the monitoring subsystem. In particular the monitoring subsystem of intr-NIRVANA has been extended so as to collect additional streams of monitoring data, in addition to those collected by NIRVANA. Therefore, intr-NIRVANA extends the set of features with 120 additional features (30 for each monitored host) extracted from monitoring data collected through the well known Ganglia monitoring tool [87].

We performed the experiments with the methodology described in section 4.3.1. Then we performed the feature selection process for both NIRVANA and intr-NIRVANA. Figure 4.9 shows the ROC curves related to such process. After the feature selection process, we obtained the classifiers for NIRVANA and intr-NIRVANA, which have been trained on the same set of experiments. Then we evaluated their detection and recognition performance on the same evaluation dataset, including experiments for each state in S. In all the experiments, NIRVANA and intr-NIRVANA have succeeded in detecting and recognizing all the injected faults. The results of this evaluation are summarized in table 4.3 and show that the overall performance of the two systems in detecting and recognizing faults are very close, with the performance of intr-NIRVANA being in general slightly better. Furthermore, intr-NIRVANA reduces the mean time to detection and mean time to recognition by few seconds.

As expected, intrusiveness provides slightly better performance, as selecting the best features from an extended set of monitoring metrics enables more precise characterization of the state of the target system. However, the slight improvement in performance may not be worth the overhead costs derived from an intrusive deployment of the monitoring system. Thus, we conclude that a non-intrusive approach is a viable alternative to an intrusive monitoring when the priority is to
4. NIRVANA: Non-intrusive Black-box Fault Diagnosis

Figure 4.9. Comparison between the ROC curves of NIRVANA and intr-NIRVANA.

Table 4.3. Comparison of NIRVANA and intr-NIRVANA with respect to the the evaluation metrics and detection and recognition latencies.

<table>
<thead>
<tr>
<th>System State</th>
<th>Detector</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
<th>FPR</th>
<th>Recognition Accuracy</th>
<th>Detection Latency</th>
<th>Recognition Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>NIRVANA</td>
<td>0.919685</td>
<td>-</td>
<td>-</td>
<td>0.080315</td>
<td>0.919685</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>intr-NIRVANA</td>
<td>0.939655</td>
<td>-</td>
<td>-</td>
<td>0.060345</td>
<td>0.939655</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>f Appalachia</td>
<td>NIRVANA</td>
<td>0.919685</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>intr-NIRVANA</td>
<td>0.939655</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f Cornwall</td>
<td>NIRVANA</td>
<td>0.919685</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>intr-NIRVANA</td>
<td>0.939655</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>NIRVANA</td>
<td>0.946512</td>
<td>0.979548</td>
<td>0.988834</td>
<td>0.984659</td>
<td>0.02282</td>
<td>0.980860</td>
<td>1.25</td>
<td>12.25</td>
</tr>
<tr>
<td></td>
<td>intr-NIRVANA</td>
<td>0.956566</td>
<td>0.985362</td>
<td>0.994131</td>
<td>0.991301</td>
<td>0.053704</td>
<td>0.980027</td>
<td>2.75</td>
<td>7.75</td>
</tr>
</tbody>
</table>

cut management overhead.

4.3.9 Performance in Multi-application Environments

In this section we present the results of the evaluation of NIRVANA in a second testbed with a more heterogeneous set of applications.

Testbed

The testbed for this set of experiments is an extension of the testbed reported in section 4.3.3. In addition to the three-tier architecture, we deployed a stream processing application on Apache Storm. The storm cluster consists of four worker nodes and the nimbus node, that coordinates the workers. Zookeeper coordinates the communication between the nimbus and the worker nodes and HornetQ provides the input data to the storm cluster. On top of the storm
cluster, we deployed a stream processing application (a topology in storm jargon) that performs a word count on a dataset of twitter data. All these applications run in a virtualized environment comprising 11 virtual machines.

**Faultload**

In addition to a subset of the faults injected in the previous experiments (see section 4.3.5), namely SQL misconfiguration (associated with system state \( f_{\text{SQL}} \)) and the crash of a JBoss instance (associated with system state \( f_{\text{CRS}} \)), we injected two other types of fault: *Storm Node Crash* \( (f_{\text{SNC}}) \), that causes the abrupt termination of a storm node, and *ZooKeeper Crash* \( (f_{\text{ZKC}}) \), that causes the abrupt termination of ZooKeeper.

**Results**

To produce the results we followed the same evaluation methodology of the previous experimental analysis. In the services discovery step NIRVANA automatically identified 10 services to be monitored. These turn to be associated with the Apache web server, the MySQL server, the AJP connections between each of the three JBoss instances and the web server, the connections to HornetQ and the connections used by the four storm nodes to communicate. After the feature selection and training step, we evaluated the detection and diagnosis performance of NIRVANA. Figure 4.10 shows the outcome of an experiment in which we injected the crash of a storm node \( (f_{\text{SNC}}) \). For ease of presentation, the figure shows the small portion of the experiment in which the injection occurs. The chart shows the output of the classifier over time, that is, the probability distribution \( P_S \) over the set of possible states \( S \). Before the injection of the fault (which occurs at second 75), the probability that the classifier assigns to the system being in a correct state is always very close to 1. Nine seconds after the fault injection the probability of the system being in the state \( f_{\text{SNC}} \) becomes greater than that of the correct state and NIRVANA correctly detects and recognizes the fault. Table 4.4 summarizes the detection and recognition performance of NIRVANA for each of the faults injected during these experiments, showing that NIRVANA achieved high accuracy also in a more heterogeneous environment.
Table 4.4. Performance of NIRVANA in the second testbed.

<table>
<thead>
<tr>
<th>System State</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
<th>FPR</th>
<th>Detection Latency</th>
<th>Recognition Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_snc</td>
<td>0.980263</td>
<td>1</td>
<td>0.968085</td>
<td>0.983784</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>f_zkc</td>
<td>0.997849</td>
<td>1</td>
<td>0.992063</td>
<td>0.996016</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>f_crs</td>
<td>0.982055</td>
<td>1</td>
<td>0.969697</td>
<td>0.984615</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>f_sql</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Chapter 5

NiTREC: Non-intrusive Black-box Anomaly Detection

In the previous chapter we presented NIRVANA, a framework for fault detection and diagnosis which is based on a non-intrusive and black-box approach. However, the proposed methodology presents a considerable limitation: it requires fault injection to be trained. This reduces the practicality and applicability of the approach for several reasons (see section 3.3).

In this chapter we present a methodology for black-box non-intrusive anomaly detection that does not require fault injection during training. The proposed methodology, that we call NiTREC, is the result of an experimental campaign that we performed in one of the data centers of the Italian Ministry of Economy and Finance [25,26]. The aim of the experimentation was to prove the validity of the ideas leveraged by NiTREC. In the next chapter we will see how we exploit and expand some of these ideas to develop a black-box non-intrusive fault detection and diagnosis methodology that does not depend on a faulty behavior based training.

In section 5.1 we introduce the methodology. In section 5.2 we present the architecture of NiTREC. Finally, in section 5.3 we report the results.

5.1 Methodology Overview

NiTREC is based on a rack-level deployment similar to that of NIRVANA (see section 4.2.1). The monitoring system consists of a set of network snifferers directly attached to the network switches of the rack, which capture the network packets, and smart PDUs to collect power consumption data. In an early stage of the experimentation, we showed the correlation between network workload and power consumption at the rack level [25]. NiTREC exploits the correlation between network metrics and power consumption data to build a model of the correct behavior of the system. An anomalous system behavior is detected whenever we observe a significant deviation of the system behavior from the one predicted by the model.

The model of the system consists of one or more regression models and is automatically built during a preliminary training phase. In our experiments we employed artificial neural networks (ANN) for this purpose. The input and output variables of the regression models correspond to network and power consumption
metrics that we extract from the raw monitoring data. During the training phase we collect a dataset of such metrics, in which each instance contains the input and the corresponding output metrics. Then, we train the regression models on this dataset.

Once trained NiTREC continuously monitors the target system and assesses its correct behavior through the regression models built during the training stage. When the estimation error of one of the models exceeds a given threshold NiTREC raises an alert. NiTREC detects anomalies at the rack level. Therefore, at the data center level we can deploy an instance of NiTREC for each rack.

5.2 Architecture

Figure 5.1 shows the architecture of NiTREC. In input NiTREC receives the stream of network packets captured from the network probes deployed at the rack level and the stream of power consumption data. The network and power consumption probes are synchronized (e.g., through NTP) and attach a timestamp to each piece of raw monitoring data so that the streams of raw data can be synchronized.

The network streams merger component combines the input network streams into a single network stream. The merged stream of packets enters the network features extractor component which processes the stream of monitoring data and computes some aggregate network statistics (e.g., packet rate, byte rate, average packet size, etc.) These statistics are computed over time intervals of fixed duration $\tau$ (in our experiments $\tau = 1s$). Similarly, the stream of power data are merged into a single stream by the power streams merger component, which is then processed by the power features extractor component that outputs some power features (e.g., active power, reactive power, power factor, phase, etc.) with the same sampling rate of the network features extractor ($1/\tau$).

The stream of feature enters in the inferential engine. This component exploits the regression models to detect anomalies. Each regression model has some input variables corresponding to a subset of network and power consumption features, and one output variable corresponding to one such feature. To assess the correctness of the current system behavior, the inferential engine computes the estimation error for each regression model. The estimation error of a given regression model at a given time $t$ is the difference between the output of the model, that is, the estimated value
of a feature, and the corresponding actual value of the same feature. Whenever, the estimation error of a regression model exceeds a given threshold (possibly different for each model) NiTREC raises an alert.

The choice of the thresholds strongly impacts the detection accuracy of the methodology. A too large threshold will possibly miss a lot of anomalies, while a too small threshold may overwhelm the system of false alarms. In this preliminary work we did not develop a methodology to automatically tune the threshold. The correct configuration of the thresholds requires the intervention of an expert operator, which is not practical and also impacts the management overhead. In the next chapter we further elaborate and improve this methodology, removing the need for manually tuning thresholds.

5.3 Experimental Evaluation

In this section we present the results that we obtained on a dataset collected during an experimental campaign at one of the data centers of the Italian Ministry of Economy and Finance (MEF) [25]. In particular, we monitored a single enclosure within a rack of the data center. Section 5.3.1 reports details of the testbed and the collected dataset. In section 5.3.2 we provide implementation details of the inferential engine. In section 5.3.3 we report the results of the experimentation.

5.3.1 Testbed and Dataset

The data center in which we performed the experiments is a medium-size facility, featuring 80 physical servers, 250 virtual servers, 20 network devices, 8 security devices, more than 50 different web applications, 2 storage area networks, more than 1000 internal users and more than 80,000 external managed single users. Within the data center we monitored a single enclosure containing 5 blade servers, running 40 virtual machines, and 4 network switches. Each blade server has 24 cores and 64 GB of RAM. To monitor the enclosure we deployed four hardware network probes attached to the network switches of the enclosure, and two Smart-PDUs to monitor the power consumption. The dataset that we used to evaluate NiTREC corresponds to an observation period of approximately 13 days. We split the dataset in a training set of approximately 10 days, and a validation set covering 3 days.

5.3.2 Inferential Engine

As regression models for the inferential engine we employed artificial neural networks (ANN). In particular we trained two Elman Recurrent neural Networks [86]. The first neural network (RNN1) takes in input the packet rate at time $t$, the day of the year, the hour of the day and the previous sample of the active power consumed by the enclosure (namely, the sample at time $t - \tau$, where $\tau$ is the sampling period). The output of RNN1 is the estimate of the active power consumed at time $t$. The second neural network (RNN2), is specular to RNN1. It takes in input the active power consumed by the enclosure at time $t$, the day of the year, the hour of the day and the previous sample of the packet rate (i.e., the sample at time $t - T$). The output of RNN2 is the estimate of the packet rate at time $t$. Both networks have 4
input neurons, one hidden layer with 5 neurons and one output neuron. Both neural networks have been trained through resilient backpropagation \cite{97}. To implement and train the neural networks we employed the encog framework \cite{15}.

### 5.3.3 Results

Since the enclosure that we monitored during the experimentation at the MEF data center is part of a production environment of a critical infrastructure, we were not allowed to inject faults to evaluate the accuracy of NiTREC in detecting them. Therefore, we performed off-line experiments in which we introduced synthetic deviations in the dataset. To evaluate the accuracy of RNN1 we performed experiments in which at a certain time we introduce an anomalous increment of power consumption. Similarly, to evaluate the accuracy of RNN2, in some experiments we introduced an abnormal increase in the packet rate. In the experiments we compute the percent estimation error of the neural network $i$ as

$$e_i = 100 \times \left| \frac{o_i - \hat{o}_i}{o_i} \right|$$

where $\hat{o}_i$ is the output of the neural network $i$ (i.e., the estimate) and $o_i$ is the actual value of the output feature. We detect an anomaly when $e_i > \tilde{e}_i$. We chose the thresholds $\tilde{e}_1 = 0.1\%$ and $\tilde{e}_2 = 30\%$ so as to maximize the F1-score. The large difference between the two thresholds is due to the fact that RNN1 is much more accurate in its estimation of the output feature than RNN2 (this, in turn, is due to the fact that the active power presents a much smaller variability than the packet rate).
Figure 5.3. Example of detection of an anomalous deviation of the packet rate with RNN1.

Figure 5.2 shows an example of an experiment. The top chart shows the actual value of the active power contained in the dataset and the value of the same metric after the insertion of the deviation. In particular at 12.00 we introduce an abnormal increase of the active power. The middle chart shows the actual value of the packet rate and the corresponding estimate of RNN2. The bottom chart shows the percent estimation error, the horizontal line corresponding to the error threshold, and the alerts, that is the points corresponding to values of the percent error exceeding the threshold. In these experiments as soon as we start the deviation the percent error increases to the point that just a few minutes after the injection NiTREC raises some alerts. The points marked with an “x” represent alerts. Marked points before the beginning of the deviation represent false positives, while unmarked points after the beginning of the deviation represent false negatives.

Figure 5.3 shows another example of an experiment. In this case we assess the accuracy of RNN1. The top chart show the actual value of the packet rate read from the dataset and the value of the same metric after the introduction of our deviation. Also in this case we start the deviation at 12.00. The middle chart shows the actual value and the corresponding RNN1 estimate of the active power. The bottom chart, as before, shows the value of the percent estimation error, the error threshold and the alerts. In this case we notice that, even though the error threshold is much smaller (due to the fact that the active power has much smaller variability than packet rate), in this experiment the error threshold is exceeded only around 100 minutes later the introduction of the deviation. It should be noted, however, that time here plays a marginal role, as what really counts is the extent of the deviation. In this case a more relevant deviation of the packet rate is required to detect an anomaly, due to higher variability of the packet rate. Table 5.1 reports the
Table 5.1. Evaluation of NiTREC’s accuracy.

<table>
<thead>
<tr>
<th></th>
<th>RNN1</th>
<th>RNN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>0.8534</td>
<td>0.9067</td>
</tr>
<tr>
<td>Recall</td>
<td>0.8745</td>
<td>0.7142</td>
</tr>
<tr>
<td>F1-score</td>
<td>0.8638</td>
<td>0.7990</td>
</tr>
<tr>
<td>False Positive Rate</td>
<td>0.03</td>
<td>0.0147</td>
</tr>
</tbody>
</table>

detection accuracy of RNN1 and RNN2 computed over the respective experiments with respect to precision, recall, F1-score and false positive rate (see section 2.3.1 for the definition of such metrics).

Discussion of the results  Obviously the obtained results should be interpreted by considering that they have been obtained on synthetic deviations introduced during off-line experiments in the dataset and that the threshold used in the detection have been tuned manually so as to maximize the accuracy. However, the aim of this experimentation was not to show the absolute accuracy of such a methodology, but rather to assess its feasibility. Thus, these preliminary results encouraged us to develop further this methodology. In the next chapter we will see how we leverage the basic idea of NiTREC to develop a fault detection and diagnosis methodology.
Chapter 6

FLOW-NIBBLER

In chapter 4 we presented NIRVANA, a framework for fault detection and diagnosis which is based on a non-intrusive and black-box approach. As already pointed out, the main limitation of such approach consists in its dependence on fault injection during the training phase. In section 3.3 we already remarked the disadvantages of such a training approach with respect to practicality and applicability of the methodology. In the previous chapter we described a methodology that can be used to overcome this limitation. In this chapter we present a novel non-intrusive and black-box fault detection and diagnosis methodology that exploits some of the same ideas to remove the reliance on fault injection. Similarly to NiTREC, this methodology relies only on a model of the correct behavior of the system, which can be built during a training stage without the need to observe how the system behaves when subject to faults. The detection of an anomalous system state is realized by comparing the current system behavior to the reference model built during the training stage. A significant deviation triggers the diagnosis process, which is based on a novel approach. Differently from NiTREC, this methodology aims at modeling the system with respect to the interactions between the individual components, rather than some aggregate metrics of the whole system. This enables the diagnosis process. We implemented such methodology in a prototype that we call FLOW-NIBBLER (FLOW-based Non-Intrusive Black-Box faLt diagnosER).

In section 6.1 we introduce an overview of our methodology. In section 6.2 we describe the model built by our approach and the processes to automatically discover and train it. In section 6.3 we present the architecture of FLOW-NIBBLER, our prototype implementation of the methodology. Finally, in section 6.4 we discuss the results of the experimental evaluation.

6.1 Methodology Overview

Figure 6.1 shows a high level overview of our methodology. First, during an initial training stage, we build a model of the target system. The model represents the interdependencies between the components of the system and is intended to model the correct behavior of the system. Such model is automatically discovered and trained by observing the system for a sufficiently long period. Since we are interested in modeling only the correct behavior of the system, we do not need to observe how
the system behaves when subject to faults. This removes the need to perform fault injection in the training stage. At the same time, we assume that the system is always correct during the training stage. If something anomalous happens during this period, our model would be trained so as to consider that anomaly as part of the correct behavior of the system. Therefore, it should be a requirement that the system is correct during the whole training period. Such a requirement may be too stringent for some production environments. In practice, by using models and learning algorithms that are tolerant to a certain amount of errors and noise in training data (e.g., artificial neural networks), it is sufficient to assume that during the training stage the system is mostly fault free, most of the time (which is a reasonable assumption for the majority of the systems).

In order to be non-intrusive our approach relies on a monitoring deployment similar to the one of NIRVANA (see chapter 4), except that in this case we only monitor network traffic. Network traffic can be monitored by deploying hardware probes directly attached to the network equipment of the target system, that passively capture network packets exchanged by system components.

The training stage consists of three steps:

1. Data Collection;
2. Model Discovery;

In the first step of the training stage we collect a sufficiently large dataset of network traces. The network traces consist in a dump of all the packets exchanged in the
target system network during the observation period. The dump of a packet contains the packet itself, as well as some metadata, such as a timestamp indicating the time at which the packet was caught. The observation period during which we collect such traces should be long enough to observe network interactions among all the components of the system. Therefore, the duration of the training stage depends on the evolution of the particular target system.

In the second step, we process the collected dataset of network traces to build a model of the system. The construction of the model consists in the automatic discovery of the system components and their interdependencies from the collected network traces. In addition, the model aims at modeling the correct behavior of the target system, that is, how the system is expected to evolve at any given time when it is correct. As we will detail in section 6.2, this is realized through a set of regression models. This part of the model needs to be trained to correctly model the behavior of the system. Thus the model discovery step returns an untrained model.

In the third step of the training stage, the model is trained on the dataset of network traces collected during the first step. Section 6.2 will discuss all the details related to the model as well as the automatic discovery and training steps to build it.

After the training stage completes, the diagnosis system enters into operation. Once in operation, the target system is continuously monitored to assess its current behavior. Its correctness is verified using the reference model of the correct behavior built during the training stage. Whenever a significant deviation from the expected correct behavior (as estimated by the model) is detected, the system is considered as behaving anomalously. The anomalies output by the detection component are associated with the particular components of the system that exhibit abnormal behaviors.

The detection of an anomalous system behavior triggers the diagnosis process. This process exploits the system model to determine the root cause of the problem from the set of anomalies discovered by the anomalous behavior detection component. The detection and diagnosis process will be discussed in more details in section 6.3.

6.2 Model Discovery and Training

The model built during the training stage is based on the central concept of what we call a flow. We refer to a flow as the network traffic exchanged between two software components of the system, one designated as the client and the other as the server component. A flow is identified by a pair \((c, s)\), where \(c\) is the client and \(s\) is the server component. Flows represent the fundamental unit of our monitoring and diagnosis approach. The model that we build around this concept consists of three parts:

1. Topology Graph
2. Dependency Graph
3. Flow Processor Models
The first two parts of the model represent, respectively, the topological structure of the system with respect to its client and server components, and the interdependencies between the identified flows in the system. This part of the model is automatically discovered by analyzing the network traces collected during the training stage.

The third part of the model is intended to model the behavior of the system, that is, how the system evolves with respect to some features computed on the monitored network traffic. Rather than building a global model of the behavior of the whole system, we aim at modeling the target system with a set of models characterizing the relations between flows. We call such models flow processor models. This is the part of the model that needs to be trained.

In the next section we give a description of the training data collected during the data collection stage. Subsequent sections discuss each part of the model in details, as well as the corresponding discovery and training processes.

6.2.1 Training Data

The dataset of network traces $T$ collected during the data collection step of the training stage, consists in a dump of the packets exchanged in the network. The dump of a packet contains the packet details, as well as some metadata, such as a timestamp corresponding to the time at which the packet has been captured. Given a flow $f$, identified by a client $c$ and a server $s$, we can isolate from $T$ the network traffic $T_f \subseteq T$ related to flow $f$, that is, $T_f$ contains only packets exchanged between $c$ and $s$.

During the discovery and training of the model, we process this dataset in different ways. As detailed later, for some discovery and training tasks we need to analyze the single packets within the network traces. For other training tasks we want to compute some aggregate function on the network traffic. In particular, in this second case, we want to extract some network features (e.g., packet rate, average packet size, RTT, etc.) over time. Given a feature $F$ that can be computed on a set of captured packets, we refer to $F(t)$, with $t \in \mathbb{N}$, as a time series where the sample computed at time $t$ refers to the feature $F$ computed on the packets captured in the time interval $(t_0 + (t - 1)\tau, t_0 + t\tau]$, where $t_0$ is the initial time (e.g., the timestamp of the first captured packet) and $\tau$ is the sampling interval. Note that in the rest of the chapter, $F(t)$ will be used to denote the whole time series, or the specific sample computed at time $t$, as long as it will be clear from the context. Analogously to the case of a single feature, given a vector of features $F = (F_1, \ldots, F_q)$, we refer to $F(t)$ as a time series of vectors, where the vector computed at time $t$ is $F(t) = (F_1(t), \ldots, F_q(t))$. Finally, we refer to $F_f(t)$ as the time series, associated with the feature vector $F$, computed on the network traffic of flow $f$.

6.2.2 Topology Graph

The topology graph captures the topological structure of the monitored system in terms of its client and server components, and the corresponding flows.

$$G_T = (C \cup S \cup H, F \cup E_H)$$

It consists of three different types of node, namely client ($C$), server ($S$), host ($H$) nodes, and two types of edge, flow ($F$) and hosting edges ($E_H$). Client and server
nodes represent system’s components acting as, respectively, a client or a server. Note that such nodes may correspond to a process running on a physical/virtual machine, a single thread, or a whole application, depending on the particular application design. From our point of view, they are just black-boxes exchanging packets with each other.

We identify each server node \( s \in S \) with a pair consisting of the network address and the port number used by the corresponding component to send and receive packets. While a server, typically, always listens on the same port number and accepts several inbound connections, a client gets assigned a different port number for every new connection, with the same port possibly being assigned to several client components at different times. Thus, we cannot identify a client node \( c \in C \) by its network address and port number, but we rather identify it with its network address and the server it communicates with. Indeed, each client component communicates with a single server component, while a server may communicate with several client components. We stress the fact that such client/server nodes represent single components (e.g., routines, threads, processes) of an application. A distributed application may consist of several client/server components.

In the topology graph each client is connected to the server it communicates with through a flow edge. Such an edge corresponds to the fundamental unit of our monitoring and diagnosis approach, namely the flow. Thus, a flow is identified by a client/server pair and represents the network traffic exchanged between these two components.

An host node \( h \in H \) is identified by a network address, and is connected to any client/server node having the same address, through an hosting edge. In our typical target environment an host node may correspond to a virtual machine, possibly hosting several services. However, since host nodes are mapped to network addresses, they more precisely correspond to network interfaces.

**Topology Graph Discovery**

The discovery of the topology graph is performed automatically by analyzing the network traces collected during the training phase. From these traces we select the datagram packets, that is, the packets of the transport layer of the ISO/OSI protocol stack. For each packet we process only the network and transport layer headers and we ignore the payload to keep the approach application-independent. Every such packet has two endpoints, the source and the destination, both characterized by a network address and a port number.

The topology graph is built from an intermediate graph representation. The endpoints graph \( G_P = (P, E_P) \) has a node for each endpoint observed in the traces and an edge \( \{p, p'\} \) if \( p \) and \( p' \) have exchanged at least one packet. A weight \( w(e) \) is associated with each edge \( e \in E_P \) that corresponds to the number of packets exchanged between the endpoints of \( e \) in the whole trace.

In order to build the topology graph from the endpoints graph we apply a simple greedy algorithm whose pseudo-code is reported in listing 1. The aim of the algorithm is to determine which nodes in the endpoints graph corresponds to server components and which ones correspond to client components. This task is often accomplished by considering well-known or operator-provided port number to service
Listing 1 Topology Graph Discovery Algorithm

\[
\begin{align*}
\text{address}(p) &= \text{network address of endpoint } p \\
N(p) &= \{p' \in P \mid \{p, p'\} \in E_P\} \quad \triangleright \text{neighbors set} \\
E(p) &= \{e \in E_P \mid p \in e\} \quad \triangleright \text{incident edge set} \\
\text{deg}_w(p) &= \sum_{e \in E(p)} w(e) \quad \triangleright \text{weighted degree}
\end{align*}
\]

1: function \( \text{TOPOLOGYGRAPHDISCOVERY}(G_P = (P, E_P)) \)
2: \( S \leftarrow \emptyset \)
3: \( C \leftarrow \emptyset \)
4: \( F \leftarrow \emptyset \)
5: \( \text{while } E_P \neq \emptyset \text{ do} \)
6: \( s = \arg \max_{p \in P} \text{deg}_w(p) \)
7: \( S \leftarrow S \cup \{s\} \)
8: \( C_s = \{(\text{address}(p), s) \mid p \in N(s)\} \)
9: \( C \leftarrow C \cup C_s \)
10: \( F \leftarrow F \cup \{(c, s) \mid c \in C_s\} \)
11: \( E_P \leftarrow E_P \setminus E(s) \)
12: \( \text{end while} \)
13: \( H \leftarrow \{\text{address}(n) \mid n \in C \cup S\} \)
14: \( E_H \leftarrow \{\{n, h\} \mid n \in C \cup S \land h = \text{address}(n)\} \)
15: return \( G_T = (C \cup S \cup H, F \cup E_H) \)
16: end function

mappings. However, such an approach is not application-agnostic, it may require operator intervention (which we want to avoid) and it is prone to errors due to, e.g., non-standard port assignments. Our algorithm instead is driven by the simple observation that, typically, a server always listens on the same port number, while clients get assigned different port numbers for every new connection. Therefore, in an endpoint graph, built from a sufficiently long network trace, the endpoints with higher weighted degree should correspond to server nodes and their neighborhood should correspond to their clients.

Thus, the algorithm works as follows. It iteratively builds the topology graph by adding one server node, and the related clients, at each step, as well as the corresponding edges (lines 5-12). The main loop of the algorithm iterates over the nodes in the endpoints graph in non-increasing weighted degree order. At each step the node \( s \) with the highest weighted degree in the endpoints graph is selected and is inserted as a new server in the topology graph (lines 6-7). All endpoints adjacent to \( s \) in \( G_P \) (i.e., \( N(s) \)) are considered its clients. Obviously, the port numbers of the endpoints associated with the client nodes are discarded, and, therefore, the actual set of clients of server \( s \) that is added to the topology graph corresponds to the set of distinct network addresses that communicated with \( s \) (lines 8-9). After the insertion of the server and client nodes, the corresponding flow edges are also inserted (line 10).

One important point to note is that while the topology graph is augmented with one server \( s \) and its clients at each step, at the same time the endpoints graph is
deprived of the edges incident to \( s \) (line 11). Therefore, the functions \( N(p) \), \( E(p) \) and \( \text{deg}^{w}(p) \) are obviously computed on the *current* instance of the endpoints graph. When there are no more edges in the endpoints graph (which implies that each endpoint has null weighted degree) the main loop terminates. Finally, the topology graph is completed with the host nodes and the related edges (lines 13-14).

Note that, since we discard port numbers while identifying clients, we basically aggregate as a single client component all endpoints having the same network address that are clients of the same server. Unfortunately, this means that we cannot distinguish different instances of a client of the same server sharing the same network address (e.g., located on the same virtual machine, or behind a gateway). In such an event we would treat this set of endpoints as a single client instance.

### 6.2.3 Dependency Graph

The dependency graph represents the causal relationships between *flows* of the topology graph.

\[
\mathcal{G}_D = (\mathcal{F}, E_D)
\]

Each flow in the topology graph, is a node in the dependency graph. Therefore, the set of nodes of the dependency graph corresponds exactly to the set of flows of the topology graph \( \mathcal{F} \). A directed edge \((f', f) \in E_D\) in the dependency graph represents the fact that flow \( f' \) is generated as a consequence of flow \( f \).

Consider for example the simple three-tier architecture in figure 6.2. The web server listens for incoming connections from a gateway and forwards received requests to the web application servers. These latter contact the database server in order to serve incoming requests. By observing the traffic exchanged in this simple network the discovery algorithm presented in the previous section should return a topology graph similar to the one reported in figure 6.3a. For ease of presentation we have represented the different node and edge types with different graphical symbols: server nodes are represented as triangles, clients as circles and host nodes as rounded squares; the flow edges are the thicker ones and are always directed from a client node to a server, while the hosting edges are represented with undirected dashed lines. In the example, the web server, characterized by network address 192.168.1.2, consists of three components: a server component which listens on port 80 and has a single client (the gateway), and two client components that forward requests to the
application servers. Both application servers consist of two components: a server that listens on port 8009 for incoming requests from the web server and a client component that sends requests to the database server. Finally, the database server consists of a single component: a server that listens on port 3306 and has two clients, namely the application servers’ client components.

The dependency graph should capture the cause-effect relation between the network traffic flows represented by the flow edges of the dependency graph. That is, we say that a flow $f'$ depends on a flow $f$, if the workload on $f'$ is a consequence of the workload on $f$. In this example flows $f_2$ and $f_4$, namely the network traffic between the web server and the two application servers, are generated as a direct consequence of flow $f_1$, namely the network traffic corresponding to the requests directed to the web server through the gateway. Flows $f_3$ and $f_5$, in turn, are generated as a direct consequence of, respectively, flows $f_2$ and $f_4$. Figure 6.3b shows the expected dependency graph for this example. Note that edges of the dependency graph correspond to direct dependencies, that is, in the example, flow $f_3$ is certainly caused by flow $f_1$, but indirectly through flow $f_2$. This is why we do not put an edge from $f_3$ to $f_1$.

Finally, we want to remark that our concept of dependency between flows differs significantly from the common notion of dependencies between network services [39]. With respect to network services, typically, one would say that a service $s$ depends on a service $s'$ if $s$, in order to provide its service, requires the service provided by $s'$. In such case, one would put a direct edge from $s$ to $s'$ in a hypothetical services dependency graph. In our example, the web server requires (and thus, in this sense, depends on) the service provided by the application server, which in turn depends on the service provided by the database server. Therefore, in an hypothetical services dependency graph we would put a direct edge from the web server to the application
server, and one edge from the latter to the database server. That is, we would obtain a graph similar to the one depicted in figure 6.3b but with edges in the opposite direction. However, the two notions of dependency are only apparently similar, and do not differ only by the direction of edges. First of all our notion focuses on dependencies between network flows rather than services (thus the nodes of our dependency graph are flows rather than services). More importantly, our notion of dependency is intended to capture the cause-effect relation between flows, rather than the commonly adopted notion of requires.

Dependency Graph Discovery

To build the dependency graph, we start from two simple observations:

1. in order for a flow \( f' = (c', s') \) to be generated as a direct consequence of another flow \( f = (c, s) \), there must be some form of communication between the components of the two flows;
2. a communication between two components in a network is typically initiated by the client component towards the server component, rather than the opposite.

The first observation suggests us to consider as potential dependent flows, those in which an end of a flow can directly communicate with an end of the second flow. If we look at the topology graph structure, we can easily note that client/server nodes have only two ways to communicate directly: through a flow connecting them, or through a common adjacent host node. The first represents a remote communication between client/server components that we can observe through monitoring. The second represents a local intra-host communication that we cannot observe with a non-intrusive approach. Since two flows cannot be incident on the same node in the topology graph, the only way for two flows to be connected by a direct communication path is through the local intra-host communication.

The second observation is quite obvious, and suggests us to look in the client-to-server direction when searching for paths of potential causation, rather than in the opposite server-to-client direction.

Listing 2 Dependency Graph Discovery Algorithm

1: function DEPENDENCYGRAPHDISCOVERY(\( G_T = (C \cup S \cup H, F \cup E_H), \hat{\rho} \))
   // Candidate dependency edges
2: \( E = \{(f', f) \in F^2 \mid f = (c, s) \land f' = (c', s') \land \exists h \in H : \{s, h\}, \{c', h\} \in E_H\} \)
   // Pruning
3: \( E_D = \{(f', f) \in E \mid \text{corr}(f', f) \geq \hat{\rho}\} \)
4: return \( G_D = (F, E_D) \)
5: end function

Driven by these considerations, when searching for dependent flows, we consider all paths of the form \( c \rightarrow s \rightarrow h \rightarrow c' \rightarrow s' \), with flow \( f' = (c', s') \) potentially generated as a consequence of flow \( f = (c, s) \). The set of ordered flow pairs \( (f', f) \) such that

---

\(^1\)We use the notation \( c \rightarrow s \rightarrow h \rightarrow c' \rightarrow s' \) as a compact way to represent the path \( c, (c, s), s, \{s, h\}, h, \{h, c'\}, c', (c', s'), s' \).
there exists a path of the form $c \rightarrow s - h - c' \rightarrow s'$ in the topology graph represents the initial set of candidate dependencies (cfr. algorithm 2, line 2). Clearly, we do not expect that all pairs in this initial set correspond to actual dependency relations. Thus, we want to filter the candidate dependency set so as to retain only those pairs that correspond to real dependent flows. To do so we evaluate the correlation between flow pairs in the candidate dependency set, and we discard all those pairs whose correlation is below a given threshold $\hat{\rho}$ (line 3), which is a parameter of the algorithm. Even though correlation does not imply causation, nor scant correlation implies independence, in practice, network traffic generated by dependent services tend to have a certain degree of correlation, while flows of packets generated by unrelated services typically present a very low correlation. Thus, in this context, correlation can provide a useful estimate of causation [24]. Clearly, the parameter $\hat{\rho}$ affects the results of the algorithm. A too large threshold would possibly discard pairs related to real dependent flows. Conversely, a too small threshold would introduce spurious dependencies. Section 6.3.3 discusses how to determine the parameters of our methodology, including the correlation threshold $\hat{\rho}$.

The correlation between two flows is computed by exploiting the dataset of network traces $T$ collected during the data collection step of the training stage. For each candidate dependency $(f', f)$ we retrieve from the dataset $T$ the corresponding traces $T_{f'}$ and $T_f$, and we compute a given feature $F$ on them at a given sampling time interval $\tau$ so as to obtain two time series $F_{f'}(t), F_f(t)$. In our experiments, we compute the packet rate at a time interval $\tau = 1$ s. The correlation between $f'$ and $f$, then, is computed as the Pearson’s correlation coefficient between the corresponding time series.

### 6.2.4 Flow Processor Models

The fundamental unit of our model, the flow, represents the part of the system that we can directly observe through monitoring. Since this is the observable part of the system, we want to characterize the correct behavior of the system with respect to flows. Specifically, we want this model to capture the relation between dependent flows. To this purpose we introduce the concept of flow processor. A flow processor is a pair of flows $(f', f)$ such that $(f', f) \in E_D$, that is flow $f'$ depends on flow $f$ according to the dependency graph. Given a flow processor $(f, f')$, $f$ is called the input flow and $f'$ is called the output (or dependent) flow. The model associated with a flow processor $(f, f')$ is a pair $FP_{f,f'} = (M_{f,f'}, E_{f,f'})$.

The first part of the flow processor model, namely $M_{f,f'}$, is a regression model describing the relation between the input flow $f$ and the output flow $f'$:

$$\hat{F}_{f',f}(t) = M_{f,f'}(F_f(t), \ldots, F_f(t - W^i + 1), F_f'(t - 1), \ldots, F_f'(t - W^o + 1), \theta)$$

where $F_f$ and $F_f'$ are the vectors of features computed on, respectively, the input flow $f$ and the output flow $f'$. $F_f(t)$ and $F_f'(t)$ here refers to the samples computed at generic time $t$, with a given sampling interval $\tau$ ($\tau = 1$ s in our experiments). The set of features of the input feature vector $F_f$ and the output feature vector $F_f'$ may be different. $W^i$ and $W^o$ are, respectively, the input and the output window, and determine how many past input/output samples are considered by the model. These are parameters of our methodology and their configuration will be discussed
in section 6.3.3 \( F^j(t), \ldots, F^j(t-W^j+1), F^\alpha_j(t-1), \ldots, F^\alpha_j(t-W^\alpha+1) \) constitute the input variables of the model, while \( \hat{F}^j_{f,f'}(t) \) represents the output of the model, that is an estimate of the output feature vector \( F^\alpha_j(t) \) computed at time \( t \). Finally, \( \theta \) are the unknown parameters of the model. These latter are determined by the training algorithm so that the model best approximates the actual function that relates the input and output variables.

The second part of the flow processor model, namely \( \mathcal{E}_{f,f'} \), characterizes the estimation error of the regression model. This is used in the anomaly detection process, as detailed in section 6.3. In particular, given the estimate of the output vector \( \hat{F}^\alpha_j(t) \), provided by the model at a given time \( t \), and the corresponding actual output \( F^\alpha_j(t) \), as observed through monitoring, we can compute the error vector \( e_{f,f'}(t) \), defined so that the \( i \)-th element of \( e_{f,f'}(t) \) is the relative error of the model on the \( i \)-th output feature:

\[
\left( e_{f,f'}(t) \right)_i = \frac{\left( \hat{F}^\alpha_j(t) \right)_i - \left( F^\alpha_j(t) \right)_i}{\left( F^\alpha_j(t) \right)_i}
\]

Given enough samples of \( e_{f,f'}(t) \), we compute the element-wise mean and standard deviations, that is, we compute two vectors \( \mu_{f,f'}^e \) and \( \sigma_{f,f'}^e \) such that the \( i \)-th element of \( \mu_{f,f'}^e \), respectively \( \sigma_{f,f'}^e \), is the mean, respectively the standard deviation, of the \( i \)-th component of the vectors \( e_{f,f'} \). Thus, \( \mathcal{E}_{f,f'} = (\mu_{f,f'}^e, \sigma_{f,f'}^e) \).

We build a flow processor model \( FP_{f,f'} \) for each pair of flows \( (f,f') \) such that \( (f',f) \in E_D \), that is flow \( f' \) depends on flow \( f \) according to the dependency graph. Section 6.3 will clarify how we use these models to detect anomalous behaviors.

**Flow Processor Models Training**

To train each flow processor model \( FP_{f,f'} \), we first divide the dataset of network traffic traces into a training set and a validation set. We use the training set and the validation set to train the regression models \( M_{f,f'} \), that is, to determine the unknown parameters \( \theta \) that best fit the training data. The specific training algorithm which is used depends on the specific regression model in use. In our experiments we employed neural networks with the resilient backpropagation algorithm [97]. Once the training of the regression model is completed, we compute the second part of the flow processor models, namely \( \mathcal{E}_{f,f'} \), on the validation set.

**6.3 Architecture**

In this section we present the architecture of FLOW-NIBBLER, a prototype implementation of our methodology, and we discuss the anomaly detection and diagnosis process.

Figure 6.4 shows FLOW-NIBBLER’s architecture. Deployed in the network of the target system we have a monitoring sub-system similar to the one of NIRVANA (see chapter 4) except that in this case we do not monitor power consumption. Network traffic data coming from the monitoring sub-system is split according to flows, so as to have a stream of captured network packets for each flow in the model.
Each such stream is processed by a flow feature extractor which extracts from the stream the features defined in the flow processor models. Thus, the flow feature extractor associated with flow $f$ receives as input the packets exchanged between the client and server endpoints of $f$ and computes the current input and output feature vectors $F^i_f$, $F^o_f$ associated with $f$. These feature vectors are computed over a sampling interval of $\tau$ seconds, therefore, each extractor outputs a couple of vectors (the input and output one) every $\tau$ seconds ($\tau = 1s$ in our experiments). All flow feature extractors are synchronized. The extracted feature vectors are the input of the anomaly detector, the component in charge of assessing the current behavior of the system. As detailed in section 6.3.1 the anomaly detector component uses the model to determine which flows are behaving anomalously. The output of the anomaly detector is the current set of anomalous flows, each tagged with a label indicating the anomaly type. The output rate of the anomaly detector matches the sampling rate of the feature vectors. At any time the fault diagnoser analyzes the current set of anomalous flows and outputs the adjudged root cause (or causes) of the observed anomalies. That is, the output of the diagnoser is the set of flows affected by the anomalies that are considered the root causes of all the other anomalies. If the set output by the anomaly detector is empty, so is the set output by the diagnoser, meaning that the system is behaving correctly.

### 6.3.1 Anomaly Detection

Even though, for ease of presentation, figure 6.4 shows a centralized anomaly detector component, it can be thought of (and easily realized) as a set of independent components, each one associated with a given flow processor. The component associated with a flow processor model $FP_{f,f'}$ receives in input the stream of input feature vectors $F^i_f$ from the feature vector extractor associated with flow $f$, and the stream of output feature vectors $F^o_{f'}$ from the extractor associated with flow $f'$. The anomaly detector component of each flow processor maintains a rolling window containing the $W^i$ most recent input feature vectors and the $W^o$ most recent output
feature vectors (including current vectors).

Whenever the anomaly detector associated with a flow processor $FP_{f,f'}$ receives new input/output vectors $F_i^f(t)$ and $F_o^f(t)$, it uses the regression model $M_{f,f'}$ to compute the expected current output feature vector $\hat{F}_o^f(t)$. This represents the model’s estimate of $F_o^f(t)$ in the hypothesis that the system is behaving correctly. Then, the detector computes the error vector $e_{f,f'}(t)$ defined in section 6.2.4. We recall that the $i$-th component of the error vector is the relative error of the model in estimating the $i$-th component of the output vector. Given the error vector, the anomaly detector computes the anomaly vector $a_{f,f'}(t)$. The $i$-th component of the anomaly vector is computed as follows:

$$a_{f,f'}(t)_i = \begin{cases} 1, & \text{if } \left| (e_{f,f'}(t))_i - (\mu_{f,f'}^e)_{i} \right| > \alpha_{f,f'} \left( \sigma_{f,f'}^e \right)_i \\ 0, & \text{otherwise.} \end{cases}$$

where $\mu_{f,f'}^e$ and $\sigma_{f,f'}^e$ are, respectively, the estimated (component-wise) mean and standard deviation of the model relative error. These two vectors are part of the flow processor model (as detailed in section 6.2.4). Therefore, the $i$-th component of the anomaly vector is 1 if the corresponding component of the error vector is more than $\alpha_{f,f'}$ standard deviations away from the mean value. This is a quite standard outlier detection technique [30] and common values for $\alpha_{f,f'}$ are 2, 2.5 and 3. Section 6.3.3 discusses how to configure $\alpha_{f,f'}$, which is a parameter of FLOW-NIBBLER.

Each anomaly detector maintains a rolling window with the $W^a$ most recent anomaly vectors. If at any time $t$, the following equation holds, for any index $i$ of the anomaly vector:

$$\sum_{t'=t-W^a+1}^{t} (a_{f,f'}(t'))_i = W^a$$

then the output flow $f'$ is considered to be anomalous. That is, before tagging a flow $f'$ as anomalous, we must collect at least $W^a$ consecutive output feature vectors of $f'$ in which a given feature is always classified as an outlier. The rationale of using a rolling window for the anomaly detection (instead of having just $W^a = 1$) is that we want to filter out outliers due to noise and fluctuations normally present in monitoring data. However, the larger the anomaly window size $W^a$, the larger the latency of the detection. Therefore, $W^a$ is another parameter of FLOW-NIBBLER and its configuration is discussed in section 6.3.3.

**Anomaly Type Classification**

When equation (6.1) holds for a given index $i$, the output flow $f'$ is considered anomalous and, in the specific, the $i$-th output feature is considered to be anomalous. When a flow is anomalous it is tagged with an anomaly type. The anomaly type is determined by a simple classification algorithm which depends only on the type of the anomalous feature and the signed error of the model. We identified two types of features and three anomaly types.

The feature type can be:

- **Workload Intensity**: features that measure the workload intensity. This type includes features such as packet rate, bit rate, average packet length, number of active connections, etc.
• Congestion/Latency: features that measure the congestion or the latency of the flow. Among the others, this type includes number of active connections, number of aborted connections, average round-trip time, etc.

The only constraint that we require with respect to the features is that in the output vector $F^o$ there is at least a feature of each type.

The possible anomaly types are:

• "overload": identifies a state in which the workload on the output flow is abnormally higher than expected, given the workload on the input flow.

• "bottleneck": identifies a state in which the output flow represents a bottleneck, that is, the workload on the output flow is lower than expected, given the workload on the input flow. Note that, despite the name, this type comprises also the extreme case in which the output workload collapses to zero, which may correspond, e.g., to the crash of an endpoint of the flow or a malfunctioning.

• "congestion": identifies a state in which the output flow is experiencing an abnormal congestion or a too high latency, given the workload on the input flow.

To understand if the model has underestimated or overestimated a given feature, we have to check the sign of the corresponding component of the model’s (signed) error. The signed error of the model is the difference between the model’s estimate of the output feature vector and the actual output feature vector:

$$\delta(t) = \hat{F}_{f,f'}^o(t) - F_{f'}^o(t)$$

The algorithm that classifies the anomaly type of a flow is sketched in figure 6.5.

For each feature tagged as anomalous in the output feature vector we have a simple decision tree. Suppose the $i$-th feature of the output vector is anomalous. If the type of the $i$-th feature is "workload intensity", then if $\delta_i(t) > 0$ it means that the actual workload intensity on the flow, as measured by the feature, is lower than expected, according to the model. Thus, the decision tree outputs "bottleneck". If $\delta_i(t) < 0$, the output of the tree is "overload", as the workload of the output flow is higher than expected, according to the model. If, instead, the feature type is "congestion/latency", then if $\delta_i(t) < 0$, the flow is experiencing a congestion or latency higher than expected, according to the model. Therefore, the decision tree outputs the type "congestion". Finally, with this feature type, the case $\delta_i(t) > 0$ should not correspond to a problem. Should this unlikely case occur, the decision tree will output the special symbol $\perp$, meaning that the particular feature is not classified.

Then, the output of each decision tree is given in input to a voter. As already stated, each decision tree is associated with a different anomalous feature of the output vector. It should be noted, however, that a flow $f'$ may be the output flow of multiple flow processors. In such case the output of all related decision trees is given in input to the voter, that is, all flow processors in which the $f'$ is the output flow are taken into consideration. The voter chooses an anomaly type among those in input that are different from $\perp$ (in case all decision trees output $\perp$, the anomalous
state of the output flow is revoked). Several voting strategies are possible, such as weighted or unweighted majority voting, etc. In our prototype we implemented a voting strategy based on priority. That is, type “overload” has priority 2, “bottleneck” has priority 1 and “congestion” has priority 0. The voter simply chooses the type with highest priority.

Each anomalous flow is tagged with the anomaly type output by the voter. The output of the anomaly detection component is, then, the set of anomalous flows $\mathcal{F}_a \subseteq \mathcal{F}$, with each flow tagged with the anomaly type, that is, $\forall f \in \mathcal{F}_a$, $\text{tag}(f) \in \{"O","B","C"\}$, where “O” is the tag corresponding to the anomaly type “overload”, “B” to the anomaly type “bottleneck” and “C” to “congestion”.

6.3.2 Fault Diagnosis

The diagnosis process analyzes the set of anomalous flows $\mathcal{F}_a$ output by the anomaly detector and searches for the root causes exploiting the dependency graph. From a high level point of view, the diagnoser tries to assign to each anomalous flow $f \in \mathcal{F}_a$, its direct cause (or causes), that is another anomalous flow $f' \in \mathcal{F}_a$ (or a set of anomalous flows) that is considered the direct cause of $f$\footnote{For the sake of conciseness, we will often say that an anomalous flow $f$ is caused by an anomalous flow $f'$, even though, to be precise, we should say that the anomaly on $f$ is caused by the anomaly on $f'$.}. Here we should distinguish between the direct and indirect cause of an anomaly. Leaving out philosophical reasonings about the transitivity of the cause-effect relation, it may happen that an anomaly on a flow $f''$ causes an anomaly on a flow $f'$, which in turn, causes an anomaly on flow $f$. In this case $f'$ is the direct cause of $f$ (because there are no intermediate cause-effect relations between $f'$ and $f$), while $f''$ is an indirect cause of $f$. Once the diagnoser has attempted to assign a cause to each anomalous flow, the subset $D \subseteq \mathcal{F}_a$ of flows for which the diagnoser did not find a cause is considered the set of root causes, that is, the diagnosis.

Given an anomalous flow $f$ its direct cause is searched in the dependency graph $\mathcal{G}_D$, moving from $f$ to other anomalous flows through the edges of $\mathcal{G}_D$. This has two implications:
Observation 6.3.1. Anomalies on flows which are not connected in the dependency graph $G_D$ are always considered as independent by the diagnoser.

Observation 6.3.2. The diagnosis algorithm must find at least one root cause for each connected component of the dependency graph $G_D$ that contains at least one anomalous flow.

The search in the dependency graph is guided by a set of three heuristic rules. These rules have been derived from simple observations on how the different anomaly types typically spread in a network. Before introducing the rules, we give some definitions:

Definition 6.3.1 (Set of upstream paths). The set of upstream paths starting from a flow $f$ in the dependency graph $G_D$, is the set $U_{G_D}(f)$ of all directed paths from $f$ to any node $f' \in V(G_D)$.

Definition 6.3.2 (Set of downstream paths). The set of downstream paths starting from a flow $f$ in the dependency graph $G_D$, is the set $D_{G_D}(f)$ of all directed paths from any node $f' \in V(G_D)$ to $f$.

Let us consider the problem of determining where to search the cause of an anomalous flow from a flow processor’s point of view. At first let us focus on a single flow processor $(f, f')$. The output flow $f'$ is typically given enough resources to manage the “normal” input workload coming from the input flow $f$. Now suppose $f'$ is affected by an anomaly of type “bottleneck”. This kind of anomaly is characterized by the fact that the workload on $f'$ is abnormally lower than expected, given the input workload on flow $f$. This means that some element in the flow processor has become “incapable” of managing the intensity of the input workload. Thus, there are two possible hypothesis: either (i) the input workload is normal, while an endpoint (or both) in the output flow has some trouble, or (ii) the endpoints on the output flow are working correctly, but the intensity of the input flow is exceptionally higher than normal. Thus, if we observe, for example, that the input flow $f$ is affected by an anomaly of type “overload”, it is probable that the anomaly on flow $f'$ is caused by the abnormal workload coming from $f$. By extending the reasoning, the “bottleneck” anomaly on $f'$ may be caused not only by an “overload” anomaly on $f'$, but, more in general, by any flow on an upstream path starting from $f'$. That is, when searching for a cause of an anomaly of type “bottleneck” on a flow $f$, the rule suggests to move upstream from $f$ in the dependency graph. If there is no “overload” anomaly on each upstream path from $f$, then the anomaly on $f$ is probably a root cause itself. An analogous reasoning can be done for the “overload” anomaly type itself. That is, when searching for the cause of an anomaly of type “overload”, we should move upstream in the dependency graph. Thus the first heuristic, namely the upstream search rule, can be defined as follows:

**Upstream Search Rule.** The set of direct causes of a flow $f$ with anomaly type “bottleneck” or “overload”, is the set of flows $f'$ such that $\exists p \in U_{G_D}(f)$ such that $f'$ is the first flow, affected by an anomaly of type “overload”, which is encountered by moving upstream from $f$ to $f'$ along the path $p$.

If the set selected by the upstream search rule for a given anomalous flow $f$ is empty, then $f$ is considered a root cause.
Now, let us take into consideration the “congestion” anomaly type. As before, let us consider the problem from a flow processor’s point of view. Considering a flow processor \((f, f')\), the fact that flow \(f'\) is affected by an anomaly of type “congestion”, means that, given the workload on flow \(f\), the congestion or latency measured on \(f'\) is abnormally high. An increasing congestion or latency on a component of the system typically arises when there is a bottleneck somewhere between the component itself and a downstream component. To illustrate this situation let us consider again the three-tier architecture reported in figure 6.2. Assume the flow from the first application server to the database server becomes a bottleneck at some time. Due to this bottleneck the time needed to serve a request coming from the web server increases, and thus, the latency on the flow between the web server and the application server increases as well. Thus, the increase in latency on the flow between the web server and the application server is caused by the downstream bottleneck located at the flow between the application server and the database server. This example suggests to move downstream in the dependency graph when searching for the direct cause of an anomaly of type “congestion”, and, in particular, to search for an anomaly of type “bottleneck”. Thus, the second heuristic, that we call the downstream search rule, can be defined as follows:

**Downstream Search Rule.** The set of direct causes of a flow \(f\) with anomaly type “congestion”, is the set of flows \(f'\) such that \(\exists p \in D_G(f)\) such that \(f'\) is the first flow, affected by an anomaly of type “bottleneck”, which is encountered by moving downstream from \(f\) to \(f'\) along the path \(p\).

If the set selected by the downstream search rule for a given anomalous flow \(f\) is empty, then the bottleneck causing the anomaly on \(f\) may be located in some element of \(f\) itself (probably the server endpoint). In this case, we should search for the cause of such bottleneck as well. Thus, in case the downstream search rule fails at finding some cause downstream, the upstream search rule should be applied on \(f\). This consideration has been taken into account in the design of the diagnosis algorithm.

So far we have explored only the cases in which the direct causes of an anomaly are searched either in the downstream or the upstream direction in the dependency graph. The third rule explores a different pattern of anomaly propagation. Think of a network element that distributes the input workload among a set of workers, such as, for example, a load balancer. From the dependency graph \(G_D\) point of view, this situation translates into a set of flows \(B\) (the output flows) and an input flow \(f^i\), such that \(\forall f \in B, (f, f^i) \in E(G_D)\). Typically, if some flow \(f \in B\) becomes a bottleneck or experiences high latencies (that is, it is affected by an anomaly of type “bottleneck” or “congestion”), the load balancer may redistribute the load on the other flows in \(B\). This re-balancing may induce an anomalous overload on the flows in \(B \setminus \{f\}\), which is caused by the anomaly on \(f\). This example highlights the fact that the direct cause of an anomaly of type “overload” affecting a flow \(f\) has to be searched, not only upstream from \(f\), but also in what we call the LB-group of \(f\).

**Definition 6.3.3 (LB-group).** The LB-group of a flow \(f \in V(G_D)\) is the set:

\[
\{f''|(f,f'') \in E(G_D) \land (f',f'') \in E(G_D)\}
\]
In the load balancer example, the LB-group of any flow in $B$ corresponds to $B$ itself (that is, the LB-group is the set of output flows). Note that the definition of LB-group is general enough to include the case of a structure with multiple input and output flows. The third rule can be defined as follows:

**LB-rule.** The set of root causes of an anomalous flow $f$ with anomaly type “overload” is the subset of flows in the LB-group of $f$ with anomaly type “bottleneck” or “congestion”.

**Diagnosis Algorithm**

The first step of the diagnosis algorithm (see listing 3) consists in building a new graph $G_C = (F^a, E_C)$, that we call the *causes graph*, whose nodes are the anomalous flows and in which an edge $(f, f') \in E_C$ means that $f'$ is the direct cause of $f$.

The causes graph is built by searching the direct causes of each anomalous flow, moving in the dependency graph through the three heuristic rules introduced earlier. Listing 3 reports the algorithm that creates the causes graph.

**Listing 3 Diagnosis Algorithm**

```
1: function Diagnosis($G_D, F^a$)
2:     $G_C \leftarrow$ BuildCausesGraph($G_D, F^a$)
3:     $K =$ StronglyConnectedComponents($G_C$)
4:     $K^* = \{G_C \in K \mid \text{deg}^-_i (V(G_C)) = 0\}$
5:     return $D = \bigcup_{G_C \in K^*} V(G_C)$
6: end function
```

The heuristic rules are implemented by means of the functions LBRule and SearchCauses. As the name suggests, the first function implements the LB-rule. It takes in input a flow $f$ and the dependency graph. First of all, it checks if $f$ is tagged as “overload”. If this is not the case, the function returns the empty set, meaning that the LB-rule cannot be applied. Otherwise, it retrieves the LB-group of $f$ (line 29) and returns the subset containing only flows tagged as “bottleneck” or “congestion” (line 30).

Both, the upstream search rule and the downstream search rule are implemented through the same generic function SearchCauses (see listing 4). This function takes three parameters: the anomalous flow $f$, representing the starting point of the search, the set of anomaly types that may have caused the anomaly on $f$, and the set of paths (starting from $f$) of the dependency graph where to perform the search. The last parameter determines how the algorithm moves in the dependency graph. By passing $U_{G_D}(f)$, the search will be performed on the upstream paths from $f$. Conversely, the search on the downstream paths from $f$ is realized by passing $D_{G_D}(f)$. The SearchCauses function searches in each path the first flow (i.e., the nearest to $f$), matching the searching criteria (line 22), that is, being tagged with one of the anomaly types passed as second argument to the function. The set of all such flows (which may be empty) is returned as the set of the adjudged direct causes of $f$. 
Listing 4 Causes Graph Creation Algorithm

“O” = “overload” tag
“B” = “bottleneck” tag
“C” = “congestion” tag

1: function BuildCausesGraph($G_D$, $F^a$)
2: $E_C \leftarrow \emptyset$
3: for each $f \in F^a$ do
4: $C \leftarrow$ LBRule($f$, $G_D$)
5: if $C = \emptyset$ then
6: if tag($f$) $\in \{“B”$,$“O”\}$ then
7: // Search causes upstream
8: $C \leftarrow$ SearchCauses($f$, $\{“O”\}$, $U_{GD}(f)$)
9: else if tag($f$) = “C” then
10: // Search causes downstream
11: $C \leftarrow$ SearchCauses($f$, $\{“B”$,$“C”\}$, $D_{GD}(f)$)
12: end if
13: end if
14: end if
15: $E_C \leftarrow E_C \cup \{(f,f')| f' \in C\}$
16: end for
17: return $G_C = (F^a, E_C)$
18: end function

19: function SearchCauses($f$, $T$, $P$)
20: $C \leftarrow \emptyset$
21: for each path $p$ in $P$ do
22: $f' =$ nearest flow to $f$ in $p$ such that tag($f'$) $\in T$
23: if $f'$ exists then $C \leftarrow C \cup \{f'\}$
24: end for
25: return $C$
26: end function

27: function LBRule($f$, $G_D = (F,E_D)$)
28: if tag($f$) = “O” then
29: $B = \{f'| (f,f'') \in E_D \land (f',f'') \in E_D\}$
30: return $\{f' \in B \mid$ tag($f'$) $\in \{“B”$,$“C”\}\} $
31: end if
32: return $\emptyset$
33: end function
The main function \texttt{BuildCausesGraph} of listing 4 implements the creation of the causes graph. At each step of the main loop (lines 2-16) the algorithm determines the set of direct causes $C$ of an anomalous flow $f \in F^a$. The set $C$ is determined by applying the heuristic rules. First of all, the algorithm attempts to apply the LB-rule (line 4). If the set $C$ returned by the \texttt{LBRule} function is not empty, then no other rules are applied. Otherwise, either the upstream or the downstream search rule is applied depending on the anomaly type of the flow $f$. If the tag of $f$ is either “B” (“bottleneck”) or “O” (“overload”), its causes are searched on all upstream paths (line 7). In this case the algorithm will search for anomalies of type “overload”. If the tag of $f$ is “C” (“congestion”), its causes are searched downstream instead (among those with anomaly type “congestion” or “bottleneck”). If this search returns the empty set, then an upstream search is performed. At the end of each iteration, for each flow $f' \in C$ an edge $(f, f')$ is added to the causes graph $\mathcal{G}_C$ (line 15). Finally, at the end of the loop, when an attempt to assign a cause to all anomalous flows have been performed, \texttt{BuildCausesGraph} returns the causal graph.

The construction of the causes graph is only the first step of the diagnosis task. Listing 3 reports the pseudo-code of the diagnosis algorithm. Intuitively, the set of flows that are sink nodes in the causes graph (i.e., anomalous flows with no causes) should be considered the root causes. According to observation 6.3.2, we should have at least one root cause for each connected component of the dependency graph that contains at least an anomalous flow. Thus, ideally, we should have at least one flow for each such component, which is a sink node in the causes graph. It is easy to prove that such a condition is guaranteed only if the causes graph is a directed acyclic graph (DAG). However, the causes graph $\mathcal{G}_C$ is not guaranteed to be a DAG, even if the dependency graph $\mathcal{G}_D$ is a DAG (which, however, is not guaranteed as well). So, in practice, the causes graph may contain directed cycles. Such directed cycles may arise for different reasons, e.g., because cycles of dependency/cause-effect relations naturally exist/occur in complex systems, or because of spurious dependencies erroneously introduced by the dependency discovery algorithm, etc.

Whether, spurious or occurring naturally, directed cycles in the causes graph may prevent the diagnosis algorithm from isolating a single root cause within the cycle. This is due to the fact that directed cycles (by definition) do not contain sink nodes, therefore, by following the edges of the loop we would never reach a root cause. However, an outgoing edge, that is an edge directed from a node in the cycle towards one which is not part of the same cycle, allows to escape from the cycle and possibly find a root cause of the anomalies in the cycle. Conversely, a directed cycle without outgoing edges cannot be escaped, and we cannot do better than returning the whole cycle as a root cause.

By generalizing, rather than directed cycles, we should consider the more general concept of a strongly connected component (SCC). A SCC is a connected component in which there is a directed path, in both directions, between every pair of nodes. A SCC, by definition, does not contain sink nodes. Thus, given a SCC without outgoing edges (i.e., edges leaving the SCC itself), we cannot do better than returning the whole SCC. We call such SCC a sink SCC. Thus, the idea is to return all sink SCCs of the causes graph $\mathcal{G}_C$ as the set of root causes. Note, that in the best case, a SCC

\textsuperscript{3}A sink node $v$ is one such that $\deg^+(v) = 0$. 
Table 6.1. FLOW-NIBBLER’s parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\rho}$</td>
<td>Flows correlation threshold.</td>
</tr>
<tr>
<td>$W^i$</td>
<td>Flow processor models input window.</td>
</tr>
<tr>
<td>$W^o$</td>
<td>Flow processor models output window.</td>
</tr>
<tr>
<td>$\alpha_{f,f'}$</td>
<td>Number of standard deviations (outlier detection).</td>
</tr>
<tr>
<td>$W^a$</td>
<td>Anomaly window.</td>
</tr>
</tbody>
</table>

is just a single node.

Therefore, the diagnosis algorithm retrieves the set of SCCs of $G_C$, which can be efficiently realized through, e.g., the Tarjan’s algorithm [113], and selects the subset $K^s$ of sink SCCs. Finally, it returns the set of flows of each component of $K^s$ as the set of root causes (i.e., the diagnosis $D$).

**Theorem 6.3.1.** The diagnosis algorithm (listing 3) returns at least one root cause for each connected component of the dependency graph $G_D$ with at least one anomalous flow. That is, the algorithm finds at least one root cause for each independent anomaly.

**Proof.** By construction, there is at least one connected component of $G_C$ for each connected component of $G_D$ with at least one anomalous flow. Therefore, it is sufficient to prove that the algorithm finds at least a root cause for the generic connected component $G^i_C$ of the causes graph $G_C$. The condensation $C(G)$ of a directed graph $G$ is a directed graph that have a node for each SCC of $G$, and an edge $(u,v)$ if and only if there is an edge in $G$ from a node of the SCC corresponding to $u$ to a node of the SCC corresponding to $v$. The condensation $C(G)$ of any directed graph $G$ is a DAG [34]. Since a DAG has at least a sink node [34], $C(G^i_C)$ has at least a sink node. From the definition of $C(G^i_C)$ it follows directly that any sink node in $C(G^i_C)$ corresponds to a sink SCC in $G^i_C$. Therefore, $G^i_C$ has at least a sink SCC, which corresponds to a non-empty set of root causes.

Moreover, if the causes graph $G^i_C$ is a DAG (which is not guaranteed though), then every SCC is a single node. This follows directly from the fact that every SCC with more than one node, contains at least one directed cycle (by definition). Therefore, a DAG cannot contain strongly connected components of size greater than one.

### 6.3.3 Parameters Tuning

Some components of FLOW-NIBBLER require the configuration of parameters, which we briefly reported in table 6.1. In this section we discuss how to configure these parameters.

**The correlation threshold $\hat{\rho}$**

The first parameter to determine is the correlation threshold $\hat{\rho}$, that is used during the dependency graph discovery to filter candidate dependencies. Specifically a
candidate dependency \((f, f')\) is discarded if the correlation of flows \(f\) and \(f'\) is below the threshold \(\hat{\rho}\) (see section 6.2.3). The parameter \(\hat{\rho}\), obviously, affects the result of the dependency discovery algorithm. A too large threshold would possibly discard real dependencies, while a too small threshold would introduce false dependencies.

There are many possible interpretations of the value of the correlation coefficient, that mainly depend on the context. A common interpretation of the absolute value of the Pearson’s correlation is the following [114]: a value between 0.5 and 1.0 is associated with strong correlation, a value between 0.3 and 0.5 to moderate correlation, a value between 0.1 to 0.3 to weak correlation and a value below 0.1 to the absence of correlation. In our experience, pairs of independent flows tend to have very low correlation values. So, probably, a wise choice when determining the value of \(\hat{\rho}\), would be to stay a little bit more on the conservative side, so as to reduce the probability of discarding real dependencies. So, setting \(\hat{\rho} = 0.3\) should be a quite safe choice.

However, in our experience, there is a quite neat separation between the correlation values of dependent and independent flows, which makes it easy to draw a line that separates the two categories. Therefore, a possible way of automating the filtering based on correlation is by means of a clustering algorithm. That is, instead of explicitly identifying a threshold \(\hat{\rho}\), we use a clustering algorithm to partition the candidate dependency set into two clusters, with respect to the correlation values. Given the nature of data, the clustering algorithm should produce two sets \(C^H\) and \(C^L\) such that \(\forall (r, r') \in C^H \times C^L, r > r'\). The cluster with lower correlation values \(C^L\) corresponds to the candidate dependencies that should be discarded. Note, that this is equivalent to choose any threshold \(\hat{\rho} \in (\max(C^L), \min(C^H))\). Figure 6.6 shows the results obtained in our experimental evaluation. We clustered the candidate dependency set by means of the \(k\)-means algorithm [69], using the training set collected during the experiments to compute the correlation between flows. As evident from the figure, in this case the \(k\)-means clustering provides the exact same result as the threshold-based approach with \(\hat{\rho} = 0.3\).

Flow Processor Models Input/Output Windows

The flow processor models input and output windows \(W^i\), \(W^o\) determine how many past input and output feature vectors are passed as input to the model (see section 6.2.4). When dealing with time series data, passing past samples to the model generally improves the model estimate [49]. However, the optimal number of past samples to give in input to the model depends on the temporal characteristics of the particular system to model. To determine these parameters, we build and train a model for each pair of values of \(W^i\) and \(W^o\), and we compute the error of each model on the validation set. Then, we choose the combination of \(W^i\) and \(W^o\) which produced the model with the least error. With such approach, if we decide to vary the values of \(W^i\) and \(W^o\) from 1 to a maximum of, respectively, \(W^i_{\text{max}}\) and \(W^o_{\text{max}}\), we need to train and validate \(W^i_{\text{max}} \cdot W^o_{\text{max}}\) different models to determine the best pair of values. This task can be onerous if \(W^i_{\text{max}}\) and \(W^o_{\text{max}}\) are large numbers. However, in our experience, \(W^i\) and \(W^o\) do not need to be vary large. For example, in our experiments we set \(W^i = W^o = 2\) (that is, we pass as input to the model the current and past input feature vector, and the past output feature vector).
6.3 Architecture

Anomaly Detector Parameters

The anomaly detector has a parameter $\alpha_{f,f'}$, for each flow processor, and a parameter $W^a$. The parameter $\alpha_{f,f'}$, relative to a given flow processor model $FP_{f,f'}$, is used by the anomaly detector component to determine if the output feature vector $F_{f'}^o$, extracted from flow $f'$, has any anomalous components (see section 6.3.1). In particular, the anomaly detector computes for each feature of the output vector $F_{f'}^o$, the relative error of the model in estimating that feature, and then checks if the error is within $\alpha_{f,f'}$ standard deviations from the mean error on that feature. If the test fails for a given feature, that feature is considered to be an outlier. If a feature is an outlier in $W^a$ consecutive output feature vectors, the feature (as well as the output flow $f'$) is considered anomalous.

Typically this kind of parameters are determined so that they provide a good trade-off between true positive and false positive rate (i.e., a good point in the ROC curve). Unfortunately, such an approach is not possible in our methodology. Indeed, since during the training stage we only observe the correct behavior of the system, we do not collect any positive example (i.e., we do not observe any anomalies) and, thus, the training and validation set contain only negative samples (i.e., correct feature vectors). Maximizing the true positive rate, while minimizing the false positive rate on such an unbalanced dataset would most probably lead to an anomaly detector that never classifies a feature vector as anomalous.

A possible strategy to determine the value of $W^a$, is to start from a minimum value which is incremented at each step. For each value, we evaluate the performance of the anomaly detector in terms of the false positive rate. We stop when the current value of the parameter does not provide a significant improvement in terms of false positive rate with respect to the previous value of the parameter. The reason to stop as soon as we have no further improvement of the false positive rate is that

![Figure 6.6. $k$-means clustering of the candidate dependency set based on the value of the absolute Pearson’s correlation coefficient.](image-url)
the true positive rate typically decreases with the size of the window. For example, in our experiments, we obtained no significant improvement of the false positive rate when extending the size of the anomaly window from 4 to 5, as shown by the plot in figure 6.7. Therefore, we set \( W^a = 4 \). Note also that the anomaly window introduces a latency in the detection, due to the fact that the detector cannot tag a flow as anomalous as soon as it observes an anomalous feature on the corresponding feature vector, but it has to wait for the anomaly to be present in all feature vectors in the anomaly window. In particular, the anomaly window introduces a delay \( d = (W^a - 1)\tau \). This is another reason to prefer a smaller window when increasing the size would not improve performances significantly.

For the parameter \( \alpha_{f,f'} \) common values include 2, 2.5 and 3. Given that it is perfectly normal to find a certain fraction of outliers even in correct data (this is the main reason why we use the anomaly window after all), a wise choice may be to be conservative when choosing the value of \( \alpha_{f,f'} \). In our experiments, we found that \( \alpha_{f,f'} = 2 \) produced a too high number of outliers, while we obtained good performances by setting \( \alpha_{f,f'} = 2.5 \), thus we chose this value.

### 6.4 Experimental Evaluation

In this section we discuss the experimental evaluation of FLOW-NIBBLER, our prototype implementation developed in the Scala programming language [92]. First, we describe how we carried out the experiments (section 6.4.1). Then, we report details on the testbed in which we performed the experiments (section 6.4.2) and how we generated the workloads (section 6.4.3). Section 6.4.4 reports details on the kind of faults that we injected during the experiments to evaluate the diagnosis performance of our methodology. In section 6.4.5 we report the configuration of FLOW-NIBBLER and details on the training. Finally, in section 6.4.6 we discuss the results of the experiments.
6.4 Experimental Evaluation

6.4.1 Experiments
To evaluate FLOW-NIBBLER we performed several experiments with the following methodology. In the first set of experiments, we collected a dataset of network traces used for the training stage of our methodology. Therefore, during these experiments we did not inject any fault in the system. We used this dataset to build and train the model needed by FLOW-NIBBLER to perform the diagnosis.

In the second set of experiments, we collected the dataset of network traces used to evaluate the detection and diagnosis accuracy of our methodology. We performed both runs in which the system is always fault-free (i.e., golden runs) and runs in which we injected a fault in the system (faulty runs). In each faulty run we started with a fault-free system and after a variable amount of time we injected a fault.

Then, we gave the collected network traces in input to our prototype and we gathered the diagnoser output, that is, the set of anomalous flows whose anomalies are considered root causes. For each experiment we built a diagnosis dataset. Each such dataset consists of a tuple \((t, D, R)\) for each sampling time \(t\), where \(D\) is the diagnosis output by FLOW-NIBBLER at time \(t\) (i.e., the set of root causes output by the diagnoser component), and \(R\) is the ground truth, that is, the “real” set of root causes at time \(t\). If \(R = \emptyset\) for a given time \(t\), then the monitored system was correct (that is, fault-free) at time \(t\), otherwise it was faulty. The evaluation of such datasets is presented in section 6.4.6.

6.4.2 Testbed
The testbed in which we performed the experiments is the second testbed described in the experimental evaluation of NIRVANA (see section 4.3.9). The testbed is a multi-application environment hosting two main applications: a web-based e-commerce application and a stream processing application. The web-based application is realized through a standard 3-tier architecture, hosting an Apache web server \([1]\) with a load balancer, a JBoss cluster \([7]\), and a MySQL \([10]\) server. The stream processing application is deployed in an Apache Storm \([2]\) cluster, comprising a nimbus and 4 workers. Two other services are Zookeeper \([3]\), which provides the communication between the storm nodes, and HornetQ \([4]\), which provides the input data to the storm cluster. For further details about the testbed see section 4.3.9.

6.4.3 Workload
The workload for the web-based application has been generated through Tsung 1.5.0 \([13]\), an open-source multi-protocol distributed load testing tool able to simulate a variable multi-user load. The workload consists of HTTP requests and has been synthetically generated according to the Pareto ON-OFF model \([28]\). The workload is generated from a physical machine external to the testbed.

6.4.4 Faultload
In our experiments we injected several types of fault. Some of them are misconfiguration faults \([23]\), that is, they are caused by a wrong configuration of some application parameters. The particular misconfiguration faults that we injected, do not cause
a service to completely fail, but rather they cause performance degradation in the system. The other faults that we inject, instead, consist in an unexpected abrupt termination of a node of the network, that is, a node crash. We injected this kind of fault on different application components and on different instances of the same application component.

- **SQL misconfiguration**: we reduce the size of the connection pool used by the JBoss instances to communicate with the database server. The heavy contention for available connections impacts negatively on application performance and creates a bottleneck towards the database server. In particular, in our experiments we reduced the maximum size of the connection pool from the default value of 20 to a value of 1.

- **AJP misconfiguration**: we reduce the size of the thread pool associated with the AJP protocol (which allows the communication between the web server and the JBoss instances). Similarly to the previous fault, this misconfiguration creates a performance bottleneck in the communication between the web server and the JBoss instances. In our experiments we reduced the size from the default value of 50 to a value of 15.

- **JBoss node crash**: we simulate the crash of a JBoss instance by killing the corresponding process. This fault causes the load balancer to redistribute the workload among the remaining JBoss instances, which causes a slight degradation of performance.

- **Storm node crash**: we cause the abrupt termination of a storm worker node by killing the corresponding processes (both the worker and supervisor). This fault causes the storm nimbus node to reassign the work of the crashed node to another node.

- **HornetQ crash**: we cause the crash of HornetQ by killing the corresponding process. This fault impacts on the entire storm cluster as the service provided by HornetQ is used by the storm workers to retrieve their input data.

### 6.4.5 Configuration and Training

In section 6.3.3 we already discussed how to determine the parameters of FLOW-NIBBLER. In particular, in our experiments we set the following configuration: \( \hat{\rho} = 0.3, W^i = W^a = 2, \alpha_{f,p} = 2.5, W^a = 4 \). As already discussed, the anomaly window \( W^a \) introduces a delay \( d = (W^a - 1)\tau \) in the anomaly detection, where \( \tau \) is the feature extractor sampling interval. In our experiments \( W^a = 4 \) and \( \tau = 1 \text{ s} \), therefore \( d = 3 \text{ s} \). The set of features that we extract from each flow is the same for input and output vectors and is reported in table 6.2.

As regression models for the flow processors we employed Artificial Neural Networks (ANN) [49]. In particular, we used Elman Recurrent Neural Networks [86]. The inputs of the neural networks are the current and past input feature vectors \( F_i(t), F_i(t-1) \) and the past output feature vector \( F_o(t-1) \) (because \( W^i = W^o = 2 \)). Each feature vector has 12 elements (the features shown in table 6.2), therefore, each neural network has 36 input neurons. The output of the model is the estimate of the
### Table 6.2: Set of features extracted from flows. The type of each feature is either Workload Intensity (WI) or Congestion/Latency (C/L).

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Packet Rate</td>
<td>WI</td>
<td>number of packets per second sent by the client component of the flow to the server component.</td>
</tr>
<tr>
<td>Client Byte Rate</td>
<td>WI</td>
<td>amount of bytes per second sent by the client component of the flow to the server component.</td>
</tr>
<tr>
<td>Client Average Packet Size</td>
<td>WI</td>
<td>average size of the packets sent by the client component of the flow to the server component.</td>
</tr>
<tr>
<td>Server Packet Rate</td>
<td>WI</td>
<td>number of packets per second sent by the server component of the flow to the client component.</td>
</tr>
<tr>
<td>Server Byte Rate</td>
<td>WI</td>
<td>amount of bytes per second sent by the server component of the flow to the client component.</td>
</tr>
<tr>
<td>Server Average Packet Size</td>
<td>WI</td>
<td>average size of the packets sent by the server component of the flow to the client component.</td>
</tr>
<tr>
<td># Open Connections</td>
<td>C/L</td>
<td>number of open connections.</td>
</tr>
<tr>
<td># Active Connections</td>
<td>WI</td>
<td>number of active connections.</td>
</tr>
<tr>
<td># Aborted Connections</td>
<td>C/L</td>
<td>number of aborted connections.</td>
</tr>
<tr>
<td>Minimum RTT</td>
<td>C/L</td>
<td>minimum round trip time.</td>
</tr>
<tr>
<td>Maximum RTT</td>
<td>C/L</td>
<td>maximum round trip time.</td>
</tr>
<tr>
<td>Average RTT</td>
<td>C/L</td>
<td>average round trip time.</td>
</tr>
</tbody>
</table>
current output feature vector, therefore, each neural network has 12 outputs. Each neural network has also an hidden layer with 36 neurons, and, since it is an Elman recurrent neural network, it has a context layer with 36 neurons linked to the hidden layer. We decided to employ recurrent neural networks as they are particularly suited for time series data [43]. To train the models we split the dataset collected during the first set of experiments into a training set and a validation set and we trained our model through the resilient backpropagation algorithm [97]. The ANNs have been implemented through the Encog framework [15].

6.4.6 Results

In this section we discuss the results obtained in the experimental analysis. First we comment the model discovery process, that is, the discovery of the topology and the dependency graphs corresponding to our testbed. Then, we present the results of the evaluation of the detection and the diagnosis process.

Model Discovery

During the model discovery step FLOW-NIBBLER was able to discover the topology of the system almost perfectly. All server and client components in the testbed have been discovered by FLOW-NIBBLER, as well as the corresponding flows except for a pair of flows connecting two storm worker nodes. This slight inaccuracy, however, is not due to the discovery algorithm itself, but to a technical limitation. Indeed, in our testbed we could not monitor the network traffic exchanged between the two mentioned storm workers, because they were co-located on the same blade server whose internal traffic is not visible from outside.

Figure 6.8 shows the portion of the topology graph corresponding to the web-based application. FLOW-NIBBLER discovered all the involved server and client components. The Apache web server consists of a server (listening on port 80), and three clients (corresponding to mod_cluster) communicating with the three JBoss instances on port 8259. Each JBoss instance has a server component, listening to requests from the web server, and a client component that communicates with the MySQL server. This latter consists of a single server component, that responds to
Figure 6.9. Portion of the topology graph corresponding to the storm cluster.

The three flows corresponding to the requests forwarded by the web server to the JBoss instances all depend on the input flow coming from the load generator to the web server. This dependence is correct, because the three flows are generated as a consequence of the input flow. The flow from each JBoss instance to the MySQL server is generated as a consequence of the flow of requests coming from the web server, and this dependence is also reflected in the dependency graph.

The portion of the topology graph corresponding to the Apache Storm cluster is reported in figure 6.9. As already stated, in this case, all the involved flows have been discovered with the exception of two missing flows (between the two storm workers identified by, respectively, the network addresses 192.168.0.125 and 192.168.1.47). Each worker node of the storm cluster has a server component listening on port 6700, and several clients associated with the flows towards the other storm workers, the nimbus node (i.e., the cluster coordinator), ZooKeeper (which allows the communication among all storm nodes) and HornetQ. This latter consists of two server components, corresponding to two different protocol interfaces. The corresponding portion of the dependency graph is not shown, however, FLOW-NIBBLER discovered all the relevant dependencies between flows in the storm cluster.

---

4Remember our definition of the dependency relation: a flow \( f \) depends on a flow \( f' \), if \( f \) is generated as a consequence of \( f' \).
Detection

We evaluated both the detection and diagnosis performances of FLOW-NIBBLER on the diagnosis datasets produced during the experiments (see section 6.4.1). The instances of the diagnosis datasets are triples $(t, D, R)$, where $D$ is the diagnosis output by FLOW-NIBBLER at time $t$ and $R$ is the actual set of root causes at that time (i.e., the ground truth). We evaluated the detection performance of FLOW-NIBBLER according to the well-known evaluation metrics reported in section 2.3.1, namely, accuracy, precision, recall, F1-score and false positive rate. These metrics are defined in terms of the number of true/false positive and negative instances of the diagnosis dataset. The conditions under which we consider an instance $(t, D, R)$ as a true/false positive or negative, are defined as follows:

**True Negative:** $D = R = \emptyset$;

**True Positive:** $D \neq \emptyset \land R \neq \emptyset$;

**False Negative:** $D = \emptyset \land R \neq \emptyset$;

**False Positive:** $D \neq \emptyset \land R = \emptyset$;

Figures 6.10 and 6.11 report the values of the evaluation metrics grouped by fault type. That is, each bar of the histograms represents the value of an evaluation metric computed over the set of experiments in which a particular fault has been injected, with the only exception of the last bar (Overall), that is relative to the complete set of experiments. In our experiments, FLOW-NIBBLER achieved quite good detection performances, with accuracy, precision, recall and F1-score always above 0.92 for each group of experiments, and the false positive rate always below 0.14 (FPR over all the experiments is 0.083). All injected faults have been detected by FLOW-NIBBLER.
Figure 6.11. Detection false positive rate. Each bar is relative to the set of experiments in which a particular fault has been injected, except for the bar labeled as “Correct”, which is relative to fault-free experiments, and the bar labeled as “Overall”, which is relative to all experiments.

We also evaluated the detection latency, that is the elapsed time from the injection of a fault and its detection. The obtained results are summarized in figure 6.12, where, similarly to the previous figures, each bar is relative to the set of experiments in which a given fault has been injected, with the exception of the last bar which is relative to the all experiments. Each bar represents the mean value of the detection latencies obtained in the corresponding group of experiments, while the error bars are the standard deviations. The mean detection latency computed over the whole set of experiments is 6.78 seconds, even though we observed a high variability in this metric (standard deviation over the complete set of experiments is 8.7 seconds), both in the single experiments and with respect to different injected faults. The minimum mean latency value of 3 seconds has been achieved for the set of experiments relative to the SQL misconfiguration fault, while the maximum mean value of 15.82 seconds is relative to the crash of the HornetQ service. In all experiments the detection latency was in the order of tens of seconds. It should be noted that the minimum latency of three seconds, that we achieve in some experiments, correspond to the minimum possible detection latency due to the delay introduced by the anomaly window \( W^a \) (see section 6.4.5).

**Diagnosis**

The quality of the diagnosis of FLOW-NIBBLER has been evaluated through precision, recall and F1-score, as defined in section 2.3.2 with respect to the sets \( D \) and \( R \). As motivated in section 2.3.2, accuracy and false positive rate do not provide good measures of the quality of the diagnosis. Figure 6.13 reports the values of the evaluation metrics, as before, grouped by fault type, with the last bar of the histogram being relative to the complete set of experiments. Also for the diagnosis we obtained quite good results. Precision and F1-score are always above 0.88, while recall is always above 0.84.
Figure 6.12. Detection latency. Each bar of the histograms is relative to the set of experiments in which a particular fault has been injected, except for the bar labeled as “Overall”, which is relative to all experiments.

Figure 6.13. Evaluation of diagnosis. Each bar of the histograms is relative to the set of experiments in which a particular fault has been injected, except for the bar labeled as “Overall”, which is relative to all experiments.
Part II

Reducing Monitoring Overhead
Chapter 7

Invariants Space Filtering

Invariants are stable relationships between system metrics that are expected to hold when the target system behaves correctly. A broken invariant, that is, the violation of such a relationship, may be an indicator of an anomalous system behavior. Thus, invariants can be exploited to perform fault detection. In a preliminary stage, invariant-based approaches analyze the target system to discover invariant relationships between its metrics, a process known as invariant mining. The number of invariants quickly grows with the number of monitoring metrics. Monitoring all such invariants in large scale complex systems incurs a significant overhead, both in terms of volume of data to be collected and computational resources to check the invariants. A moderately complex system may expose hundreds of invariants, with only a portion of them actually useful for the detection process. In general, some mined invariants may be useless, that is, they are never violated, or redundant. Some of them may be excessively unstable, that is, they are violated too easily, possibly producing a high number of false positives and, thus, worsening the detection accuracy. In this chapter we propose a filtering methodology of the invariant space, with the twofold purpose of reducing the monitoring overhead and improving detection accuracy. To assess such improvement, we describe and evaluate an invariant-based fault detection methodology which incorporates the filtering strategy. However, we believe that the filtering methodology is general enough to be applied to a wide range of invariant-based techniques.

In section 7.1 we provide a background on invariant analysis and we report the related works. An overview of our invariant-based detection methodology and the filtering strategy is presented in section 7.2. In the next three sections we describe in details the invariant mining process, the filtering process, which represents our main contribution, and the detection process. Finally, in section 7.6 we present the experimental results.

7.1 Background and Related Work

Invariants have been applied to several contexts, including program evolution and software bug detection [42, 47, 48, 56, 123], autonomic system management [66, 103], and fault detection [50, 85, 85, 99, 101, 104].

A sound invariant for a program $P$ is a property that is guaranteed to hold for
every execution of $P$ [99]. Discovering sound invariants, typically, requires static analysis and theorem proving techniques. However, such techniques do not scale enough to be applicable to real programs [91]. A likely invariant [99] is a property of a program that holds on many executions of $P$ on the observed inputs and is expected to hold on other inputs. Likely invariants may be unsound (i.e., they may not hold for some input), but discovering them is easier and less expensive than extracting sound invariants. For this reason, there has been a lot of interest in the development of techniques for discovering likely invariants [47,48,66,99].

Invariants can be classified into three main categories [99]:

- **Value-based**: invariants specifying properties involving only program values [48,56,70,80].

- **Control-flow-based**: invariants specified in terms of properties of the control flow of the program [53,116].

- **PC-based**: invariants specifying properties that involve program counter values [123].

Lou et al. [85] propose an extension of control-flow-based invariants, which they call execution flow invariants, and exploit them for detecting problems in distributed systems. Execution flow invariants are defined in terms of linear relations between the counts of different type of log messages. A violation of such invariants is used to detect anomalous execution paths.

In the context of value-based invariants, Jiang et al. [66] introduce flow-intensity invariants to measure the intensity with which internal monitoring data reacts to the volume of user requests. The methodology proposed in this chapter focuses on this kind of invariants. Flow-intensity invariants can be mined by analyzing time series extracted from the available monitoring metrics (e.g., CPU usage, memory usage, packet rate, etc.), which makes techniques based on this kind of invariants independent from the particular system under monitoring.

In [66] an invariant relationship is inferred between each pair of metrics as soon as measurements are available. Then, each invariant is incrementally validated, by computing a confidence score, as new measurements are available. After a given time period the invariants whose confidence score is below a given threshold are discarded and are no more validated. However, this approach does not scale with the size of the system. Sharma et al. [104] proposed to use invariants to detect and localize faults in distributed systems. They build a graph in which nodes are system metrics and edges represent the invariant relationship between pair of metrics. They use this graph to capture the interdependencies between system components and localize faults. In [101] and [50] similar approaches are used to mine invariants from application logs to support anomaly detection in cloud system.

Invariant mining techniques, typically, extract a large number of invariants, with many of them possibly be spurious. With the aim of discarding such unwanted invariants, in the context of software evolution, Ernst et al. [48] estimate the probability that an invariant would be discovered by chance with a random input. If this probability is smaller than a user-specified threshold, the invariant is selected, otherwise it is discarded.
Our methodology \cite{22} is different from such approaches in that it introduces an automatic filtering of the invariants space which benefits from the analysis of monitoring metrics related to the faulty behavior of the target system to improve fault detection accuracy while reducing monitoring overhead.

### 7.2 Methodology Overview

The invariant-based fault detection methodology that we propose (figure 7.1) is based on three stages:

1. **Invariants Mining**
2. **Filtering**
3. **Detection**

As in most invariant-based approaches, the *invariants mining* stage consists in the analysis of monitoring data collected when the target system is behaving correctly, with the aim of identifying invariant relationships between pairs of observed metrics. The second stage consists in the automatic *filtering* of the invariants found at step 1, with the twofold benefit of decreasing monitoring overhead, by reducing the number of metrics to monitor and process, and improving the detection accuracy. Indeed, commonly, invariant-based anomaly detection techniques filter the invariant space with very simple filtering strategies such as those based on goodness of fit and fixed thresholds. This way, a lot of invariants are mined, with many of them possibly be:

- **Useless**: an invariant that is never broken is not useful for detecting anomalies.
- **Redundant**: two invariants which are always broken together provides the same information (thus, we can discard one of the two invariants).
- **Excessively unstable**: an invariant that is too weak (i.e., broken too easily) may cause a high number of false positives and an overall decreased accuracy.
- **Broken too seldom**: an invariant that is broken too seldom may contribute to raise the number of false negatives.

To discard such unwanted invariants we follow this strategy. Given the fault model for the target system, the idea is to inject instances of such faults in the system in order to check which invariants are actually violated. In particular, on the basis of this information we perform five filtering steps, which are described in section 7.4. The automatic filtering step is the main contribution of our methodology. Even though in this work we propose an invariant-based fault detection methodology which embeds the filtering strategy, we believe that the latter is general enough to be applied to a wide range of invariant-based techniques.

Once the mining and filtering stages are completed, the detection system becomes operational and enters in the detection stage. In this stage the detection system continuously monitors the filtered set of invariants to check when an invariant is broken. An alert is raised whenever one of the invariants is broken, indicating that a fault is present in the system.
Figure 7.1. Stages of the invariant-based fault detection methodology.

Figure 7.1 outlines the three steps, whose details will discussed in the following sections.

7.3 Invariant Mining

As shown in figure 7.1, the input of this stage is the characterizing data obtained by observing the target system when it is operating correctly. The characterizing data consists of monitoring metrics which characterize the target system behavior (e.g., CPU usage, memory usage, incoming/outgoing network packets, etc.) The first step of the invariant mining stage consists in sampling the characterizing data so as to obtain a time series for each metric. All time series must be sampled with the same period $\tau$, and are relative to the same observation period. Therefore, for each metric $x$, we obtain a time series $x(t)$, where $t$ is a discrete variable expressed in units of $\tau$. The sampling period impacts the results both of the mining and the detection process. A smaller sampling period results in a larger dataset to be analyzed, which in turn results in larger computational overhead for the mining process (as well as a larger monitoring overhead in the detection stage). Conversely, a too large sampling period may result in information loss and inaccurate modeling. In [101], it is shown that a finer grain of data used in the mining process improves the modeling and the detection. In our experiments, due to technical constraints, we set the sampling interval to the minimum possible value of $\tau = 1$ s.

The invariant mining algorithm that we employ to discover the initial set of invariants is the one proposed in [101]. It takes as input the set of time series and mines invariants with AutoRegressive models with eXogenous (ARX) inputs, as proposed in [104]. Given a pair of metrics and corresponding time series $x(t)$ and $y(t)$, the ARX model describes the following relationship between them:

$$y(t) + a_1 y(t-1) + \cdots + a_n y(t-n) = b_0 x(t-k) + \cdots + b_m x(t-k-m)$$  \hspace{1cm} (7.1)

where $y$ and $x$ are, respectively, the output and input variables of the model; $n$ and $m$ are, respectively, the number of past outputs and inputs considered in the model; $k$ is the delay between the output variable $y$ and the input variable $x$, and
the coefficients $a_i$ and $b_j$ are the parameters of the ARX model that reflects how strongly a past sample is affecting the current output.

Let us denote the model’s coefficients and the input/output variables in the vector form:

$$\theta = [a_1, \ldots, a_n, b_0, \ldots, b_m]^T$$

$$\varphi(t) = [-y(t-1), \ldots, -y(t-n), x(t-k), \ldots, x(t-k-m)]^T$$

We can rewrite the model (equation 7.1) as:

$$y(t) = \varphi(t)^T \theta$$ \hspace{1cm} (7.2)

To determine the coefficients $\theta$ of the ARX models, we employ a recursive least squares (RLS) algorithm [84], which reduces computational and memory requirements. Assume the period of observation of the time series is such that $t \in [1, N]$. The least squares method generates an estimation of the model $\hat{y}(t|\hat{\theta}) = \varphi(t)^T \hat{\theta}$ in which:

$$\hat{\theta}(N) = \left[ \sum_{t=1}^{N} \varphi(t) \varphi^T(t) \right]^{-1} \left[ \sum_{t=1}^{N} \varphi(t) y(t) \right]$$ \hspace{1cm} (7.3)

Equation 7.3 can be written in a recursive way as:

$$\begin{align*}
\hat{\theta}(t) &= \hat{\theta}(t-1) + P(t) \varphi(t) \left( y(t) - \varphi^T(t) \hat{\theta}(t-1) \right) \\
P(t) &= P(t-1) - \frac{P(t-1) \varphi(t) \varphi^T(t) P(t-1)}{1 + \varphi^T(t) P(t-1) \varphi(t)}
\end{align*}$$

Initial values for $P$, $\hat{\theta}$, and $\varphi$ must be provided. Typically, $\varphi(0)$ is initialized with values of $x$ and $y$ collected from monitoring data. When no knowledge about the parameters is available, it is common practice [84] to set $\hat{\theta}(0) = 0$ and $P(0) = cI$, where $I$ is the identity matrix, and $c$ is a large number (e.g., $\geq 1000$).

We use the RLS algorithm on each pair of metrics $x$ and $y$, thus obtaining an ARX model (i.e., a potential invariant) for each pair of metrics. The output of the mining stage is the complete set of candidate invariants.

### 7.4 Filtering

As shown in figure 7.1, the filtering stage receives in input the set of candidate invariants from the mining stage. This unfiltered set of invariants is likely to contain a lot of spurious invariants, and using it in the detection stage would result in very poor performance. The filtering stage aims at filtering the set of candidate invariants with the twofold benefit of drastically reducing the number of metrics to be monitored and improving the detection accuracy at the same time.

The filtering stage consists of three steps: checking, filtering 1, and filtering 2. In the following sections we discuss each step.
7.4.1 Checking

In the checking step we verify which invariants are broken when faults are present in the system. As discussed later, the filtering 2 step benefits from the output of this stage. To accomplish this task we need to collect characterizing data during periods in which the system is affected by faults. We obtain this data through fault injection [45]. The mechanism used to detect broken invariants is discussed in section 7.5.

The filtering 2 step is performed on vectors reporting the violation of invariants in a given observation period. Considering an observation period with sampling times $t_0, \ldots, t_{N-1}$, we associate with each invariant $I_j$ a vector $v_j$ of size $N$, where $N$ is the number of observations, and the $k$-th element of $v_j$ is defined as:

$$(v_j)_k = \begin{cases} 
1, & \text{if } I_j \text{ is broken at time } t_k \\
0, & \text{otherwise}
\end{cases}$$

7.4.2 Filtering 1

The filtering 1 step performs a filtering on the basis of the goodness of fit. This is a quite standard filtering [50,101] of the invariant space and exploits only characterizing data collected during periods in which the target system behaves correctly. The idea is to discard the candidate invariants with a goodness of fit (GoF) outside a specific range. The rationale is that candidate invariants with a low GoF do not provide a good modeling of the relation between the input and output metrics. Conversely, candidate invariants with a too large GoF are likely mined from too similar time series and are thus meaningless. Consider, for example, an invariant relating CPU utilization $\text{CPU\_util}$ and CPU idle time $\text{CPU\_idle}$. The two time series are linearly dependent, thus, the resulting invariant has a very large GoF. However, since the relation $\text{CPU\_util} = 1 - \text{CPU\_idle}$, would never be broken even in the presence of faults, such an invariant is useless.

To assess the GoF of a given invariant we compute the coefficient of determination $R^2$, that represents the percentage of the variation that can be explained by the model. The closer the value of $R^2$ to 1, the better the regression. In particular,

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

$$SS_{res} = \sum_{t=1}^{N} (y(t) - \hat{y}(t))^2$$

$$SS_{tot} = \sum_{t=1}^{N} (y(t) - \bar{y})^2$$

where $N$ is the number of observations and $\bar{y}$ is the average value of $y(t)$ over all the observations. $R^2 = 1$ means that the regression perfectly fits the data.
7.4 Filtering

7.4.3 Filtering 2

Filtering 2 includes five subsequent filtering steps. These filtering steps exploit the information coming from the checking stage, that is, the information about the status of the invariants when the system is subject to different faults. This information is provided in the form of the vectors defined in section 7.4.1.

1) Never-broken invariants filtering. We discard the invariants that are never broken. Namely, we discard an invariant $I_j$ if the corresponding vector $v_j = 0$. As a matter of fact, there could be invariants whose goodness of fit is such that they are not discarded by filtering 1, and whose relation always holds. Such invariants are not useful to detect anomalous events.

2) Correlated invariants filtering. When applying the GoF filtering, invariants consisting of too similar time series are discarded. However, there could still be similar invariants, that is, invariants representing similar relations and that are always broken at the same time. Consider the following example. Assume that there is an invariant in the system relating CPU utilization and memory utilization $\text{Mem}_\text{util} = \alpha \text{CPU}_\text{util}$. Since, CPU utilization and CPU idle are linearly dependent there must be also an invariant $\text{Mem}_\text{util} = \alpha' \text{CPU}_\text{idle}$. However, those invariants provide the same information and considering both would be redundant, thus, we can keep just one of them and discard the other. To filter such invariants, for each pair of invariants $I_j, I_k$ we compute the Pearson’s correlation coefficient of the corresponding vectors $v_j, v_k$. If the correlation is larger than a threshold $\text{Corr}_{th}$, invariant $I_j$ is discarded from the set of mined invariants. The threshold $\text{Corr}_{th}$ is a parameter of our filtering methodology. In section 7.6.4 we will discuss how we determine this parameter.

3) Too-often broken invariants filtering. Some invariants may be too weak, that is, they are broken too often. These invariants may significantly raise the number of false positives, thus impacting detection accuracy. Therefore, we filter invariants that are broken more than $Oft_{all}$ times the average number of times all the invariants are broken, in the observation period. The threshold $Oft_{all}$ is another parameter of our filtering strategy. In section 7.6.4 we will discuss how we determine this parameter.

4) Invariants broken before injection filtering. Some invariants may be broken before a fault is injected, when the system behavior is expected to be correct. Consider an invariant that is not broken too often (thus, it is not filtered at step 3) but it is always broken before an injected fault is activated, i.e., before the anomaly occurs. This invariant would generate false positives. Thus, we filter all the invariants that are broken when no anomaly is occurring in the system.

5) Seldom broken invariants fault-specific filtering. Invariants that are seldom broken for a specific fault are discarded. Even though at step 3 invariants broken too often are filtered, the remaining ones should break often enough to detect the injected faults and avoid false negatives. Thus, we remove invariants that are...
broken less than $Sld_{spec}$ times the average number of times the invariants are broken, in the observation period, for a specific injected fault. In section 7.6.4 we will discuss how we determine the parameter $Sld_{spec}$.

7.5 Detection

In the detection stage, we continuously monitor system metrics related to the set of filtered invariants. We detect the presence of a fault whenever an invariant is broken. To detect such a situation we compute the residual function of the ARX models associated with invariants. Given an invariant, related to metrics $x$ and $y$, the residual function computed at time $t$ is defined as the absolute difference between the actual value of the output variable at time $t$ and the corresponding estimated value

$$R_{x,y}(t) = |y(t) - \hat{y}(t|\hat{\theta})|$$

where $\hat{y}(t|\hat{\theta})$ is the estimate of the output variable provided by the ARX model associated with the invariant and $y(t)$ is the actual value of the output variable as measured through monitoring.

We consider an invariant as broken at a given time $t$ if $R_{x,y}(t) > \delta$, for a given threshold $\delta$. Obviously, the choice of the parameter $\delta$ strongly impacts the detection accuracy. A too small threshold may cause invariants to break too easily, thus, producing many false positives. Conversely, a too large threshold may raise the number of false negatives. In [101] it is shown that using a threshold $\delta$ which adapts to the prediction interval of the output variable with respect to a specific input [65], provides better results than choosing a fixed threshold. In particular, we consider an invariant as broken if the actual output of the system is outside the prediction interval of the model’s output. This happens if the residual function is larger than the standard deviation of $\hat{y}(t|\hat{\theta})$ for the specific value of the input variable $x(t) = x_p$.

The standard deviation is computed as:

$$\sigma = S_{err} \sqrt{1 + \frac{1}{N} + \frac{(x_p - \bar{x})^2}{\sum x^2 - N\bar{x}^2}}$$

where $S_{err}$ is the standard deviation of the model error (square root of the mean squared error), $\bar{x}$ is the sample mean of the input variable and $N$ is the number of samples.

7.6 Experimental Evaluation

In this section we present the results of the evaluation of our methodology. We evaluated both effects of the invariant space filtering: the reduction in monitoring overhead and the increase in detection accuracy. To assess the impact of our filtering strategy (filtering 2) on monitoring overhead and detection accuracy we compare our methodology with the one using only the goodness of fit filtering (filtering 1). In the following sections we report some details about the testbed, the monitoring deployment, the workload and the faults injected during the experiments. In section 7.6.4 we introduce the plan of experiments, and, finally, in section 7.6.5 we discuss the results of the evaluation.
7.6 Experimental Evaluation

7.6.1 Testbed and Monitoring

The testbed in which we performed the experiments is the one described in section 4.3.3. Since one of the objectives of this work is to show that our filtering strategy can significantly reduce the monitoring overhead, we deployed an intrusive and white-box monitoring system to collect a large number of metrics. In particular, we deployed the well known monitoring tool Ganglia [87], which collects monitoring data through agents installed in the components of the target system. In addition to Ganglia built-in metrics, we also monitored some application-specific metrics (Apache web server and Java Virtual Machine (JVM) related metrics) with freely available Ganglia plugins. Along with the Ganglia monitoring system, we deployed some hardware probes to collect network traffic and power consumption monitoring data. With this setup we monitored 78 distinct metrics.

7.6.2 Workload

The workload has been generated from a machine external to the testbed with Tsung 1.5.0 [13]. We generated three levels of workload (high, medium, low) according to the methodology described in section 4.3.4.

7.6.3 Faultload

In the following we report a description of the faults injected in the experiments, classified according to the taxonomy defined by Avižienis et al. in [23].

Misconfiguration faults: this type of fault derive, typically, from human errors and consists in the wrong configuration of the system or a component.

- SQL misconfiguration: we reduce significantly the connection pool used by the application server to connect with the DB.
- AJP-long misconfiguration: we reduce the thread pool for the AJP protocol (which allows the communication between the Apache web server and the JBoss slaves) to a very small size.
- AJP-short misconfiguration: same as AJP-long misconfiguration, but we also reduced the length of the queue associated with the thread pool.

Reconfiguration faults: these faults are introduced by changes of configuration during maintenance.

- Write permissions: we revoke write permissions to one of the JBoss instances on its working directory.

Denials of service faults: either malicious or not, these faults may lead to system unavailability due to the saturation of some hardware resources.

- CPU stress: we impose an abnormal CPU load on the target machine by running a highly CPU-intensive task.
Table 7.1. Factors and levels of the design of experiments.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Level 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oftail</td>
<td>0.1</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
<td>0.8</td>
<td>1.0</td>
<td>...</td>
<td>2.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Sldspec</td>
<td>0.1</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
<td>0.8</td>
<td>1.0</td>
<td>...</td>
<td>2.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Corrh</td>
<td>0.70</td>
<td>0.75</td>
<td>0.80</td>
<td>0.85</td>
<td>0.90</td>
<td>0.95</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Memory stress**: we impose an abnormal level of memory activity that causes high memory contention on the target machine.

- **Disk stress**: we cause an abnormal disk access activity on the server hosting the SQL server. **Full partition**: we cause the disk partition on the machine hosting Apache and the SQL server to become full.

*Development faults*: these faults, typically, produce erratic output or software aging phenomena [44].

- **Memory Leak**: we run a process affected by memory leak that causes the memory of the target machine to progressively saturate. The memory exhaustion in turn triggers the thrashing phenomenon.

### 7.6.4 Plan of experiments

We performed several experiments to collect monitoring data from both, golden runs, in which we did not inject faults, and faulty runs, in which we injected the faults reported in section 7.6.3. From the experiments we generated a *training set*, which is used in the mining and filtering stages, and a *test set* which is used for assessing the detection accuracy. The experiments performed to collect the training were 9 minutes long (time needed to reach a steady state, where metrics can be collected while excluding impact from any transient effect). We performed each such experiment with a different combination of workload levels and injected faults. To build the test dataset we performed a single 90 minutes long experiment in which we varied the workload and we injected all kind of faults reported in section 7.6.3.

The filtering stage is performed on the training set. In order to determine the best configuration for the parameters of the filtering methodology (i.e., Corrh, Oftail and Sldspec) we considered a design of experiments [65] in which the factors are the filtering parameters and the *levels* (i.e., the values assigned to the factors) have been chosen so as to cover a wide range of cases. Table 7.1 reports the levels for each factor. As *response variables* we considered coverage, precision, recall and detection latency. Coverage measures the fraction of types of fault that are detected. If $p$ types of faults occur, and the detector finds $r$ of them, then coverage = $r/p$. The detection latency is the elapsed time from the injection of a fault to its detection. Precision and recall have been already defined in section 2.3.

The results of this design of experiments are presented in the next section.

### 7.6.5 Results

In this section, we present the results of the experiments. We first discuss the results obtained on the training set, in terms of reduction of monitoring overhead due to
7.6 Experimental Evaluation

Table 7.2. Results of the training related to the best combination of the filtering parameters. 
$$Corr_{th} = 0.85, \ Oft_{all} = 1.8, \ Sld_{spec} = 3.$$ 

<table>
<thead>
<tr>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
<th>Latency</th>
<th># Inv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.80</td>
<td>1.00</td>
<td>0.89</td>
<td>33.33</td>
<td>25</td>
</tr>
</tbody>
</table>

the filtering of the invariant space.

Then we discuss the results obtained on the test set, in terms of detection accuracy. To assess the increase in accuracy we compare our methodology with the one using only the goodness of fit filtering (filtering 1).

Training

The mining and filtering stages have been applied on the training set in order to identify the invariants and find the best configuration of filtering parameters. As a result of the design of experiments discussed in section 7.6.4, the experimentation produced 1,176 outputs for each response variable. The larger the precision and recall, the better the detection quality of the detector, since it avoids false positives and false negatives. To identify the best configuration for the parameters $$Corr_{th}, \ Oft_{all}, \text{ and } \ Sld_{spec},$$ we used the Pareto multi-objective optimization algorithm [52]. The algorithm returns a Pareto front with 16 combinations of the configuration parameters. Among such configurations, we chose the one which allowed to detect all faults occurred in the system ($$Corr_{th} = 0.85, \ Oft_{all} = 1.8, \text{ and } \ Sld_{spec} = 3$$). The results (with respect to the evaluation metrics) obtained with this combination of parameters are reported in table 7.2. On the training set, the invariant based approach detected all injected faults (thus, coverage and recall are both 1). Precision is 0.80 and the mean detection latency is 33 seconds.

The initial set of 78 monitoring metrics, led to 3,003 possible invariants (i.e., all pairs of metrics). The goodness of fit filtering selected 265 of them. These invariants involved all 78 metrics, therefore, the goodness of fit did not allow to reduce the volume of monitoring data at all. Filtering 2 reduced the number of invariants by about 90%, from 265 to only 25. Moreover, the 25 invariants involved only 19 metrics out of 78, which corresponds to a reduction of about 75.6% of monitoring metrics. This implies a significant reduction of the monitoring overhead both in terms of volume of data to be collected and computational overhead to check the invariants.

Test

In order to validate our claim that filtering 2 increases the accuracy of the detection we compared two detectors: one using the set of invariants selected by our filtering strategy and the other using the set selected by the GoF filtering only. The detection technique is the one described in section 7.3. Both detectors are evaluated on the same test set. The results of the comparison are summarized in table 7.3. The first row of the table reports the results obtained by the detector using the set of invariants selected by our filtering methodology, while the second row reports the results obtained by the detector using the set of invariants selected by the GoF filtering.
Table 7.3. Comparison between the detection performance obtained with the set of invariants selected by our filtering methodology and with the set of invariants selected by the GoF filtering only.

<table>
<thead>
<tr>
<th>Filtering</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
<th>Latency</th>
<th># inv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our approach</td>
<td>0.76</td>
<td>0.99</td>
<td>0.86</td>
<td>65.55</td>
<td>25</td>
</tr>
<tr>
<td>GoF</td>
<td>0.57</td>
<td>1.00</td>
<td>0.73</td>
<td>47.78</td>
<td>265</td>
</tr>
</tbody>
</table>

The comparison of the two detectors shows that the one using the proposed filtering methodology largely outperforms the detector not using it. The detector based on the GoF filtering achieved a perfect recall and a lower detection latency. Indeed, typically, a larger set of invariants allows to achieve a very high recall and a lower latency due to the higher chance that at least one invariant is violated when there is a fault in the system. However, the higher recall and lower latency are obtained at the cost of a significant reduction in precision, due to the higher probability of generating false positives. Our filtering methodology strongly improved the precision from 0.57 to 0.76, still achieving a very high recall (0.99). The better combination of precision and recall is reflected in the F1-score which is improved by our filtering methodology from 0.73 to 0.86.
Conclusion

In this thesis we proposed methodologies to develop more practical and applicable fault detection and diagnosis solutions. We identified and discussed several aspects that impact the practicality and limit the applicability of a methodology. We pointed out that solutions based on an intrusive monitoring impose a significant burden on operators, due to the installation, configuration and maintenance of the diagnostic system. Moreover, operators are often reluctant to install intrusive monitoring software as it may interfere with the system. Finally, in some environments, security and privacy policies may limit or prevent the deployment of an intrusive monitoring system. A non-intrusive approach allows to mitigate such issues.

Another factor that we identified as having a strong impact on practicality and applicability is the reliance of the diagnostic system on application specific-information. Such information is sometimes difficult to obtain and maintain, and in some environments may be unavailable. Moreover, it may limit the applicability of the solution to specific contexts. Therefore, we identified application independence, which can be realized through a black-box approach, as an important aspect to enhance practicality and applicability.

Some diagnosis solutions also need to be trained by observing how the system behaves when it is affected by faults. To observe the faulty behavior of the system during the training stage, faults must be deliberately injected in the system. This imposes an additional burden on operators, which must carefully plan and perform the fault injection campaign. Moreover, it limits the applicability of the solution. Indeed, in many environments it is not possible to inject faults in the production site, and the presence of a testing site would be required.

We identified these three factors (intrusiveness, application dependence and faulty behavior based training) as mainly impacting management overhead, that is, the additional effort imposed to operators to manage the diagnostic system. In the first part of the thesis we proposed three methodologies aimed at reducing this kind of overhead.

We first presented NIRVANA, a detection and diagnosis solution based on a black-box non-intrusive approach. We have evaluated the accuracy of NIRVANA and we have compared it to an intrusive version of it, showing that the intrusiveness did not provide significantly better results. One of the main limitations of NIRVANA is its reliance on fault injection in order to be trained to recognize the target system state. To overcome this limitation we first presented NiTREC a black-box non-intrusive anomaly detection methodology that does not require a faulty behavior based training, thus removing the reliance on fault injection. Then, we presented FLOW-NIBBLER, a black-box non intrusive solution, which further elaborates and
expands the methodology employed by NiTREC, to perform detection and diagnosis without relying on fault injection for the training phase.

In the second part of the thesis we focused on another aspect that impacts practicality and applicability, namely monitoring overhead. It consists in the additional computational and communication resources required to collect and process monitoring data. Given the large scale and complexity of today’s systems, the monitoring may impose a significant overhead, thus impacting both practicality and applicability. We proposed a general solution to reduce monitoring overhead while improving detection accuracy, in the context of invariant-based fault detection. Invariants are stable relationships between monitoring metrics that are expected to hold while the system behaves correctly. Invariant-based techniques exploit such relationships to detect anomalous behaviors. The number of such invariants quickly grows with the number of monitoring metrics. We pointed out that many discovered invariants may be actually useless, redundant or excessively unstable to the point of worsening the detection accuracy. Thus, we proposed a general methodology to filter such unwanted invariants and, with the purpose of assessing its impact on fault detection, we described an invariant-based fault detection technique embedding it. We showed that the filtering strategy is able to reduce considerably the volume of data to be collected and processed. Moreover, we compared the performance of two versions of our detection methodology: one using our filtering strategy, while the other not. We showed that the detector using our filtering methodology significantly outperformed the other in terms of accuracy, thus, proving that our filtering strategy is able to also improve performance.

Future Work

In chapter 3 we introduced the intrusiveness and application dependence factors, and we discussed different levels of intrusiveness (depending on the deployment of the monitoring system) and application dependence (namely, the white-, gray- and black-box approaches). Yet, all the methodologies proposed in the first part of the thesis implement the same combination of intrusiveness and application dependence levels, namely a non-intrusive and black-box approach. Indeed, this is the combination that allows to improve practicality and applicability the most. However, as discussed in chapter 3 such benefits come at costs, mainly in terms of diagnosis accuracy and granularity. An interesting future direction would be to explore methodologies adopting different combinations of increasing levels of intrusiveness and application dependence, assessing the trade-off between the impact on practicality and applicability, and the impact on the diagnosis process.

In the second part of the thesis we only evaluated the impact on fault detection accuracy of our filtering strategy of the invariant space. However, we believe that also the diagnosis process might benefit from our filtering strategy. Indeed, carefully selecting the set of invariants that allow to distinguish different kind of faults would be critical for the diagnosis task. As future work we plan to explore this direction of research.

Moreover, the filtering strategy that we proposed is general enough to be applicable to a wide range of invariant-based fault detection systems. However, it is
not general enough to be applicable to other methodologies, such as, for example, NIRVANA and FLOW-NIBBLER. Nonetheless, the main idea might be generalizable. An interesting direction of research would be to explore the possibility of generalizing such methodology, that is discarding part of the model built by the diagnostic system with the aim of improving detection and diagnosis accuracy, while reducing monitoring overhead.
Bibliography


