# Adaptive Anomalous Behavior Identification in Large-Scale Distributed Systems

**Javier Álvarez Cid-Fuentes**

School of Computer Science

The University of Adelaide

This dissertation is submitted for the degree of

*Doctor of Philosophy*

Supervisors: Dr. Claudia Szabo and Prof. Katrina Falkner

August 2017

*A mi familia.*

# Declaration

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Javier Álvarez Cid-Fuentes

August 2017

# Acknowledgements

First of all I would like to thank my two supervisors, Dr. Claudia Szabo and Prof. Katrina Falkner, for their support and guidance during these three years (and a bit). Thank you for your patience, help, insight, and above all, thank you for giving me the opportunity to work with you and for teaching me everything about research.

I would also like to thank the people who, besides my supervisors, helped me to carry out my studies in Adelaide. Francesc, Cruz, Rosa, and Wamberto, this thesis would not have been possible without your support.

I am also very thankful to the university staff and students for creating a really enjoyable atmosphere that makes things much easier. I am especially grateful to the people who, in one way or another, helped me during my studies. Thanks to Dan and Marianne for their immense technical support; to Yongrui and Ali for our fruitful talks; to Dídac for his brief but really insightful assistance; to Yuval for his help in the area of security; and to Lachlan and Jamal for the good times spent together. Finally, I would like to give special thanks to Pádraig and Zhigang for their academic and personal support. These years would not have been the same without you.

Por otro lado, también me gustaría darle las gracias a todas las personas que, a nivel personal, han estado a mi lado durante estos años. En primer lugar, muchísimas gracias a mis padres, que han trabajado durísimo toda su vida para darnos a mí y a mi hermana unas oportunidades que ellos no tuvieron. Sin vosotros no sería quien soy, ni estaría donde estoy. Gracias también a mi hermana, por estar siempre ahí, y a mis tíos, Maribel y José Luís, por haber hecho esta experiencia más llevadera aun estando a miles de kilómetros de aquí.

Gracias a todos los amigos que están lejos pero están, gracias a Esteve por el soporte y por haberme traído un trocito de casa contigo en los días en los que estuviste aquí, y gracias a Juan por aquel fin de semana en Montreal que se pareció a los míticos jueves. Agradecérselo también a todas esas *aves de paso* que han hecho de mi estancia en Australia una experiencia inolvidable: Carles, Vicent, Zulia, Sandra, Nacho, Mehdi, Luís, Tania, Andrea, Carmen, Gorka, Rafa, Cristian, Jonathan, Alex y Jorge. A Alfonso y Gerard por los buenísimos ratos y las risas. Y en especial a Pablo y Cassia por su gran ayuda y las muchas tardes y noches que hemos compartido.

Por último, *last but not least*, dar las gracias a Ana, una de las personas más importantes de mi vida, por haber compartido conmigo este viaje que no ha sido fácil, por la cantidad y la calidad de los momentos, y por haber estado a mi lado incondicionalmente.

# Abstract

Distributed systems have become pervasive in current society. From laptops and mobile phones, to servers and data centers, most computers communicate and coordinate their activities through some kind of network. Moreover, many economic and commercial activities of today's society rely on distributed systems. Examples range from widely used large-scale web services such as Google or Facebook, to enterprise networks and banking systems. However, as distributed systems become larger, more complex, and more pervasive, the probability of failures or malicious activities also increases, to the point that some system designers consider failures to be the norm rather than the exception.

The negative effects of failures in distributed systems range from economic losses, to sensitive information leaks. As an example, reports show that the the cost of downtime in industry ranges from \$100K to \$540K per hour on average. These undesired consequences can be avoided with better monitoring tools that can inform system administrators of the presence of anomalies in the system in a timely manner. However, key challenges remain, such as the difficulty in processing large amounts of information, the huge variety of anomalies that can appear, and the difficulty in characterizing these anomalies.

This thesis contributes a novel framework for the online detection and identification of anomalies in large-scale distributed systems that addresses these challenges. Our framework periodically collects system performance metrics, and builds a behavior characterization from these metrics in a way that maximizes the distance between normal and anomalous behaviors. Our framework then uses machine learning techniques to detect previously unseen anomalies, and to identify the type of known anomalies with high accuracy, while overcoming key limitations of existing works in the area. Our framework does not require historical data, can be employed in a *plug-and-play* manner, adapts to changes in the system behavior, and allows for a flexible deployment that can be tailored to numerous scenarios with different architectures and requirements.

In this thesis, we employ our framework in three anomaly detection application domains: distributed systems, large-scale systems, and malicious traffic detection. Extensive experimental studies in these three domains show that our framework is able to detect several types of anomalies with 0.80 *Recall* on average, and 0.68 mean *Precision* or 0.082 mean $FPR$ depending on the domain. Moreover, our framework achieves over 0.80 accuracy in the identification of various types of complex anomalous behaviors. These results significantly improve similar works in the three explored research areas.

Most importantly, our approach achieves these detection and identification rates with significant advantages over existing works. Specifically, our framework does not rely on historical anomalous data or on assumptions on the characteristics of the anomalies that can make anomaly detection easier. Moreover, our framework provides a flexible and highly scalable design, and an adaptive method that can incorporate new system information at run time.

# Table of contents

# List of figures

# List of tables

# Chapter 1

# Introduction

Computer and communication systems are a pivotal part of current society to the point that some authors have coined the term *the network society* [2] to refer to their profound impact on present economic, social, political and cultural aspects. Most current software systems can be considered distributed systems, that is, *"systems in which components located on networked computers communicate and coordinate their actions by passing messages"* [3]. Since the global spread of the Internet, most computers communicate across some kind of network, and often perform actions in response to messages received from other computers. Even more, with the increasing popularity of the *Internet of Things* (IoT) [4], even a ordinary lamps may communicate with each other [5].

Due to this global interconnectivity, most distributed systems are also large-scale systems with hundreds, thousands or even millions of components. Examples are web services such as the widely used Google search engine [6], Facebook [7], or Amazon [8]; peer-to-peer (P2P) networks used for file sharing such as BitTorrent [9]; high performance computing (HPC) clusters; privacy oriented networks like TOR [10]; the internal networks of companies and universities; vast networks of small devices in the context of IoT; virtualized infrastructures on demand such as Amazon EC2 [11]; or large-scale systems from other sectors like finance, health, or the military.

These computer systems have allowed unprecedented advances at numerous levels, from the health sector to space exploration. However, they have also made society much more fragile in the sense that a single computer failure can produce enormous economic and personal losses [12, 13, 14, 15]. As a recent example, a single router malfunction that caused a cascading failure forced Southwest Airlines to cancel or delay more than 2,000 flights, which generated between \$54 million and \$82 million in losses [14]. Therefore, it is essential for society that computer system experts and researchers provide the adequate tools to avoid failures, or at least to minimize their catastrophic consequences.

The area of research that studies how and why computer systems fail, and how to avoid or ameliorate the negative effects of failures is the area of *dependability*. Dependability has been defined as a measure of availability, reliability, maintainability, confidentiality, integrity, and safety [16]. Availability is the capacity to deliver a service when requested; reliability is the capacity to remain delivering the correct service over time; maintainability is the ability to allow modifications and repairs; confidentiality is the capacity to ensure the privacy of users; integrity is the absence of improper alterations; and safety is the absence of catastrophic consequences on the users and environment [16]. For obvious reasons, the goal of any system designer or administrator is to achieve the highest possible degree of dependability. However, this is often an arduous task due to the size and complexity of current computer systems, which are composed of thousands of computers that interact with each other. This vast amount of interactions generates nonlinear dependencies between system components that make the system extremely hard to diagnose and maintain.

The different threats to dependability can be organized into three categories: *failures*, *errors* and *faults* [17]. Failures are events that occur when there is an incorrect service delivery. Failures can occur at software, hardware, or system level. For example, a failure in a software component implies that the component stops delivering correct functionality to other components, whereas a failure in a group of networked computers can imply that a common service, such as the delivery of a web page, becomes unavailable to users. Failures are the final consequence of errors, which are deviations from the correct internal state of the different entities that participate in a computer system [16]. Thus, an error produces a failure when an internal deviation reaches the boundaries of the entity in question, affecting other external entities, which can be other software components, computers or the users. Finally, the causes of errors are called faults. Faults range from software and hardware flaws to input or configuration mistakes to intrusion attempts and attacks. In other words, anything that can generate a deviation from the correct state of a system is considered a fault. Fig. 1.1 shows the relationship between failures, errors and faults.

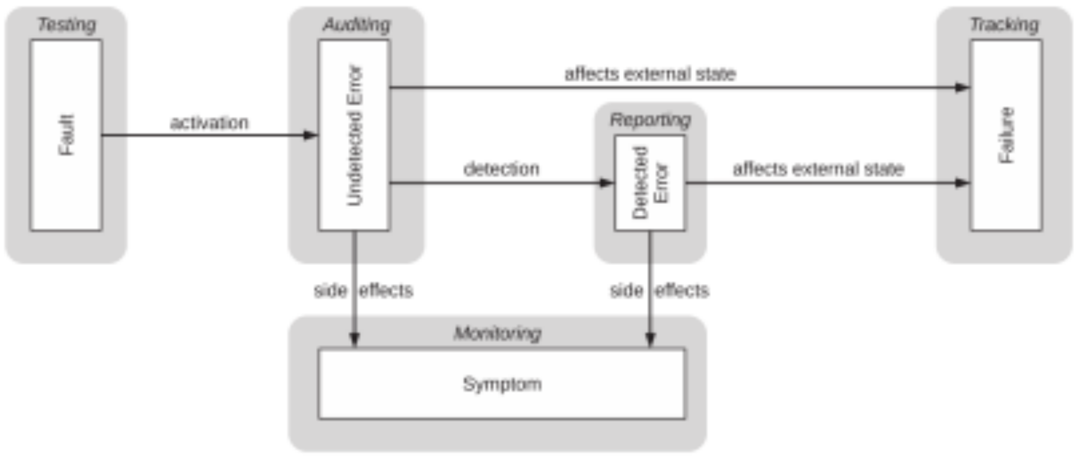


**Fig. 1.1.** Relationship between faults, errors and failures (extracted from Salfner et al. [17]).

Some faults can exist in a computer system for long periods of time without generating errors, and are only activated when a very specific condition is met. For example, a software bug can remain inactive until a particular execution path or state is reached. Faults that have not generated an error are called *dormant faults* [16]. Errors can be detected or undetected, and typically have some side effects before generating a failure. These side effects usually manifest in certain system metrics as symptoms, which are often the only indication of the presence of an error.

The means to achieve dependability, and thus avoid failures, can be organized into four categories [16]: fault prevention, fault tolerance, fault removal and fault forecasting. *Fault prevention* involves the use of design and development techniques to avoid introducing faults into computer systems, such as formal verification techniques [18]. *Fault tolerance* is the ability of detecting and handling errors to avoid failures. *Fault removal* involves testing and verification methods to find and remove faults, and *fault forecasting* involves constructing system models to analyze the possible faults that can occur.

In this thesis, we focus on fault tolerance for two main reasons. Firstly, as computer systems grow in size and complexity, ensuring the complete absence of faults through fault prevention becomes impossible [19]. Secondly, certain types of faults are extremely difficult to find and remove without running the system in a production environment [20, 21], which reduces the effectiveness of fault removal and forecasting techniques.

Fault-tolerance is composed of two main activities: error detection and recovery. As shown in Fig. 1.1, errors have side effects that can be observed through monitoring. Thus, employing the adequate tools, errors can be detected and handled by means of recovery methods to avoid and ameliorate the negative effects of costly failures. More precisely, error detection, also referred to as *anomaly detection* [22], consists of analyzing system information to find symptoms that can be indicative of the presence of an error. Anomaly detection is categorized into white, gray or black box depending on the type of information analyzed [23]. *White box analysis* is based on the analysis of internal system information, such as application execution paths or the internal variables. As such, white box analysis is typically achieved through instrumentation of the source code of the applications in a computer system [24]. *Gray box analysis* employs information located at a more external level of the system, such as the call stack or the processes memory, and *black box analysis* exclusively relies on external system variables, such as resource consumption or generated outputs. White box analysis is the most accurate of the three, but also the most expensive in computational terms [24]. Once the relevant system information has been collected, anomaly detection typically relies on statistical and machine learning techniques to find abnormal patterns caused by errors [22].

Since current computer systems display a broad range of architectures, purposes and characteristics, anomaly detection has plenty of areas of application [22]. Examples include distributed systems [23], IP networks [25], sensor networks [26], and security [27]. In essence, the problem of anomaly detection remains the same across all application domains, however, the information gathered, the means to collect it, and the anomaly detection techniques are sometimes tailored to meet the particular requirements of the systems under study.

Anomaly detection is one of the pillars of fault tolerance, and fault tolerance is crucial for ensuring the dependability of computer systems and, by extension, for ensuring the correct functioning of most of our economic, social, political, and cultural activities. Despite the immense relevance of anomaly detection and the numerous research efforts to address it [22], key challenges remain, such as:

- **Definition of anomalies:** the number of anomalies that can appear in a particular computer system is often very large and increases over time as the system and the environment evolve [22]. Thus, obtaining historical data for all the anomalies that can occur is unattainable. This makes the anomaly detection problem very challenging, as detection systems do not have access to a precise definition of what anomalies look like before these appear.

- **Boundary of normality:** since a precise definition of anomalies is often unavailable for a given system, defining what is considered normal in the system is also a difficult task that depends on the area of application [22]. Defining a boundary of normality typically requires the choice of a threshold, which has a direct impact on the accuracy of the detection system. A threshold that is too permissive will consider many anomalies as normal, whereas a threshold that is too restrictive will generate a lot of false positives. However, since anomalies are not well defined, the optimal threshold value is usually unknown and cannot be determined easily. In addition, what is considered normal can change over time as the system and the environment evolve, there can exist various separate definitions of normality within the same system, and what is considered normal often depends on the context. For example, high resource consumption can be normal during system maintenance and abnormal otherwise.

- **Scale:** most current computer systems consist of a large number of software components, often running in various computers that communicate through a network. Anomaly detection in large computing systems is especially challenging due to the high computational resources required to process large amounts of information, and the need for an scalable, possibly distributed, approach [23].

Existing works in anomaly detection deal with these challenges in different ways. Since a general definition of anomalies is difficult to build, some works obtain this definition from historical anomalous data [28, 29, 30, 31], or limit their detection to anomalies that have very specific characteristics, such as point anomalies [21, 32], sudden changes [29, 21, 33], or system specific anomalies [34, 35, 36, 37]. In this manner, anomaly detection becomes easier because there is a more detailed definition of certain anomalies. However, detection is restricted to the anomalies that match this detailed definition.

The boundary of normality is typically controlled by a configuration parameter and the characteristics of the available data. Approaches that rely on historical anomalous data [28, 29, 30, 31] can also build a more precise boundary of normality. However, these approaches cannot detect previously unseen anomalies. As mentioned before, what is considered normal can change over time and might depend on contextual information. However, many existing works employ fixed boundaries [29, 21, 33], and few approaches take into account contextual information [38, 39, 40].

Several existing works are concerned with the scalability of their methods [21, 37, 41, 42], and some approaches experiment with large-scale scenarios [30, 34, 35, 39, 43]. However, there is still a gap between the size of real world large-scale systems and the size of the systems employed in the literature. More precisely, existing works typically analyze systems composed of around 300 computers, whereas real world data centers can be composed of 50,000 to 80,000 servers [44, 45].

In addition to this, identifying the type of the anomalies is also necessary since it can reduce the time spent in debugging, and help system administrators in finding the root causes of problems.

## 1.1 Contributions

The work presented in this thesis focuses on black box anomaly detection in large-scale distributed systems. Most current computer systems can be defined as distributed, however, we are mostly interested in large-scale systems composed of tens to thousands of computers due to their pervasiveness and relevance in present society [44, 45].

The main contribution of this thesis is defining an adaptive framework for black box anomaly detection without historical data and its successful application in three domains: distributed systems, large-scale systems, and malicious traffic detection. More precisely, the main contributions of this thesis are:

**A black box framework for anomaly detection in large-scale distributed systems [46, 47]:** we present a framework designed to overcome several of the existing limitations in anomaly detection when specifically applied to large-scale distributed systems. Our framework is designed with a flexible architecture and employs machine learning techniques that enable its application to numerous infrastructures. Our framework does not rely on historical anomalous data, is designed to detect anomalies regardless of their characteristics, adapts to changes in normal behavior, can detect contextual anomalies, and can be deployed in a decentralized and scalable manner.

**A methodology for the identification of complex anomalous behaviors in distributed systems [47]:** we employ our framework for the detection and identification of complex anomalous behaviors, such as deadlock, livelock, and unwanted synchronization, in distributed systems. In addition, we analyze different ways of representing the behavior in distributed systems in order to maximize anomaly detection and identification without historical anomalous data. This is crucial as obtaining historical anomalous data is challenging in most real world scenarios [48], and anomaly identification can help system administrators to find the root causes of problems more easily.

**A decentralized architecture for anomaly detection in large-scale systems:** we propose a decentralized architecture to detect anomalies in large-scale systems composed of hundreds to thousands of computers, generating tens of thousands of readings per unit time. This architecture is highly scalable and consumes an insignificant portion of the overall computing resources, making it suitable for real world large-scale infrastructures.

**A method for the detection of previously unseen malicious traffic in large-scale networks [49]:** we present a method for representing the individual behavior of computers in a large-scale network based on the traffic they generate. We employ this method together with our anomaly detection framework to detect malicious traffic without historical data, and without any assumptions on the traffic type.

In addition to these contributions, the work presented in this thesis has produced the following publications:

- [46] Javier Álvarez Cid-Fuentes, Claudia Szabo, and Katrina Falkner, “Online Behavior Identification in Distributed Systems,” in *Proceedings of the 34th Symposium on Reliable Distributed Systems*, 2015, pp.202–211.
- [47] Javier Álvarez Cid-Fuentes, Claudia Szabo, and Katrina Falkner, “Anomaly Detection in Distributed Systems without Historical Data,” to be submitted to *IEEE Transactions on Dependable and Secure Computing*.
- [49] Javier Álvarez Cid-Fuentes, Claudia Szabo, and Katrina Falkner, “A Framework for the Online Detection of Novel Botnets,” under review for *24th ACM Conference on Computer and Communications Security*, 2017.

## 1.2 Thesis Organization

The remainder of this thesis is organized as follows.

- **Chapter 2** presents a review of the state-of-the-art in anomaly detection in the three domains where our framework has been applied: distributed systems, large-scale systems, and malicious traffic detection. Chapter 2 also gives an analysis on common key issues in existing works in these three areas, as well as on issues that are specific of each field.
- **Chapter 3** overviews the limitations in the area of anomaly detection and presents our black box anomaly detection framework. Chapter 3 describes the assumptions and design choices behind our framework construction, and the theoretical foundations, algorithms, and machine learning and statistical techniques employed to overcome many of the limitations identified in the area.

Chapter 3 also proposes three different architectures that can be employed in infrastructures with different characteristics and requirements.

- **Chapter 4** analyzes the characteristics of distributed systems and a particular class of anomalies that we denote as complex anomalous behaviors. Chapter 4 describes the design of a synthetic distributed system that we employ to evaluate our framework's performance. In the experimental section, Chapter 4 presents a parametric study that shows how different configuration parameters affect this performance.

- **Chapter 5** Chapter 5 describes how the framework presented in Chapter 3 can be employed in the detection and identification of complex anomalous behaviors and change point anomalies with great success. Chapter 5 presents a set of experiments that study our framework detection and identification capabilities when employing different features to represent distributed systems behavior. These experiments are carried out with data from our synthetic distributed system and with a publicly available dataset by Yahoo! [50].

- **Chapter 6** tackles the problem of anomaly detection in particularly large systems such as high performance computing clusters or large-scale data centers. Chapter 6 analyzes the challenges in anomaly detection that are specific to these kind of systems, and then describes how the framework presented in Chapter 3 can be employed in a decentralized manner to overcome these challenges. Chapter 6 also describes the process that we employ to simulate a large-scale system composed of 1,000 computers using monitoring data from the synthetic distributed system presented in Chapter 4. Chapter 6 then presents an experimental analysis where the detection of anomalies in large-scale systems, as well as the scalability of our approach, is evaluated from the perspective of four anomaly profiles.

- **Chapter 7** analyzes the anomaly detection problem when applied to the detection of malicious traffic in large-scale networks. Chapter 7 reviews the challenges and limitations in this area, and describes how the framework proposed in Chapter 3 can be employed to overcome them. Chapter 7 also proposes a methodology to transform network traffic data to a representation of computer's behavior that can be exploited by our framework. Chapter 7 then conducts an experimental study using the most varied and realistic publicly available dataset in the area [1], which contains network traffic from more than 6,000 computers, including 32 types of malicious traffic of different characteristics. Chapter 7 compares the results obtained to similar works in the area [1], proving the superiority of our approach.

- **Chapter 8** concludes this thesis and analyzes the future research directions in anomaly detection as a whole, as well as in anomaly detection when applied to the three domains explored in this thesis.

# Chapter 2

# Literature Review

In this chapter, we present a review of existing works related to error detection in the main application areas covered by this thesis. As mentioned in Chapter 1, error detection, often referred to as *anomaly detection* [22], consists of diagnosing the state or behavior of a running system by analyzing certain system metrics that can indicate the presence of errors. The analyzed metrics range from hardware counters to network packets depending on the system under study, and the analysis is typically done by means of statistical or machine learning techniques [22]. Error detection has applications in numerous domains, such as distributed systems [23], IP networks [25], sensor networks [26], and security [27] among others.

Anomaly detection can be categorized into black, gray, and white box depending on the level of the analysis [23]. White box approaches analyze internal system information, such as execution paths. Gray box methods analyze other more external system data, such as the call stack, and black box approaches analyze the system from external variables, such as resource consumption. For example, Magpie [51] is a gray box request extraction and workload modeling tool that can be used to detect performance anomalies in multi-tier systems. Magpie employs instrumentation to track the execution path followed by requests and their resource consumption. Then, similar requests are clustered together using an incremental clustering algorithm.

Requests are tagged as anomalous when they do not belong to any cluster. Magpie requires additional efforts from developers to instrument the source code of the system. Moreover, it cannot be applied if access to this code is restricted.

Another gray box approach is Pip [52], a framework for the detection of unexpected behaviors in distributed systems. Pip is based on two main mechanisms. On the one hand, the programmer instruments the source code of applications to generate Pip execution traces at run time. On the other hand, the programmer writes a description of the expected behavior of applications in a defined language. Thus, when the system is executed, Pip compares the execution traces obtained with the written expectations to detect deviations. To manually define the expected behavior of a system in a textual description can be difficult and inefficient in systems with a high number of components and numerous complex interactions. Thus, Pip is not well suited for large-scale scenarios.

In this thesis we focus in black box analysis because it requires less computational resources and thus is better suited for large-scale systems [24]. Since we employ our approach to tackle the error detection problem in three domains, namely, distributed systems, large-scale systems, and malicious traffic detection, in the following, we review the existing black box anomaly detection approaches that have been applied to these three domains. Since distributed systems and large-scale systems are overlapping domains, in the large-scale section (Section 2.2), we review approaches that experiment with systems composed of more than one hundred computers.

## 2.1 Anomaly Detection in Distributed Systems

In this section, we summarize in chronological order the most relevant works in error detection in the area of distributed systems, where it has received significant attention often referred to as anomaly detection [23].

The work by Hood and Ji [53] contributed to the foundations of current black box anomaly detection in distributed systems and networks. Hood and Ji propose a method for the detection of anomalies in networks by analyzing data obtained through the Simple Network Management Protocol. The authors fit an autoregressive model [54] to the collected data, and model the normal behavior of the monitored network by means of a Bayesian model [55]. This model is then employed to detect deviations from normality, and to notify the network manager when appropriate.

More recent work by Williams et al. [32] demonstrates the feasibility of failure prediction through the black box analysis of performance metrics in distributed systems. Williams et al. propose a framework, called Tiresias, able to detect sudden changes in the system metrics by means of a dynamic template of normal behavior. Then, through a set of heuristic rules, Tiresias decides whether a failure is likely to occur in the near future. Williams et al. assume that the degradation in some performance metrics, such as CPU usage or network traffic, are likely to produce a global failure eventually. Even though the authors focus on failure prediction, Tiresias heavily relies on anomaly detection to generate its predictions. Tiresias can detect sudden changes in a particular system metric, however, it cannot detect changes in the correlation between multiple metrics, or anomalies characterized by a progressive change, such as a trend.

Ozonat [38] presents an anomaly detection technique for distributed web services, where system performance metrics are monitored over a time period and divided into segments, that is, sets of contiguous readings that are similar to each other. These segments are discovered by means of information theoretic techniques and dynamic programming. Then, segments are clustered based on statistical distances between them using an agglomerative hierarchical algorithm. Segments not belonging to any cluster are considered anomalous. Since Ozonat's method relies on the observed data, and does not make assumptions on the characteristics of anomalies, it can detect a wide variety of errors. However, choosing the sensitivity of the clustering algorithm is system dependent, and difficult to achieve without any knowledge about characteristics of the anomalies.

Cherkasova et al. [56] propose a method for the detection of performance anomalies and application changes in multi-tier architectures. The method consists of a regression based transaction model that correlates processed transactions and consumed CPU, and an application performance signature that provides a model of the behavior of the application. Performance anomalies are detected when the observed CPU utilization cannot be explained by the observed application workload, and application changes are detected when CPU consumption exhibits a change sustained over time. Cherkasova et al. provide an adaptive method that takes into account the evolution of the system metrics over time. However, the method relies on historical data, which might not be available in many cases, and limits the detection process to well-known anomalies.

Gu and Wang [33] present a method for the online prediction of bottlenecks in data stream applications running in a cluster. Online means that the method works at run time. The method analyzes system performance metrics, such as CPU and memory consumption, and employs a set of Bayesian classifiers to detect anomalies, and a discrete time Markov chain to compute the probability of future bottlenecks. The method relies on anomaly symptoms learned from historical failure data. Thus, when the symptoms are repeated, the probability of a future bottleneck is assumed to be high. Apart from that, the method assumes that system metrics are independent from each other, which is not necessarily true in many scenarios.

EbAT [41] is a lightweight online anomaly detection framework based on entropy calculation for cloud infrastructures. EbAT collects different virtual machine metrics, such as CPU utilization, over a time period and computes their entropy in a sliding window manner that generates an entropy time series. Then, EbAT employs a spike detection algorithm to locate abnormally high values in entropy that can be indicative of an anomaly. EbAT can detect previously unseen anomalies because it does not rely on historical failure data. Moreover, EbAT adapts to changes in behavior because it only analyzes entropy deviations with respect to recently observed data.

Wang et al. [40] propose two statistical techniques to detect anomalies in distributed systems: Tukey and relative entropy. The Tukey method [57] is employed to detect point anomalies, that is, anomalies characterized by a single abnormal reading in a performance metric, and is based on the interquartile range of a series of data points. The relative entropy method is employed to detect anomalies characterized by an abnormal sequence of readings in a performance metric. The entropy method is based on the multinomial goodness-of-fit test [58], and finds anomalies by looking at the distribution of values in the recent history. Wang et al. provide an adaptive method that does not require historical anomalous data, and that analyzes system behavior over time. However, Wang et al. do not take into account the correlations between the different system metrics, and can only detect anomalies characterized by a change in a single metric.

PREPARE [29] is a prediction-driven performance anomaly prevention system for virtualized cloud computing infrastructures. PREPARE employs a 2-dependent Markov chain to estimate the future value of system performance metrics, and a tree-augmented naive Bayesian network to decide if the predicted value corresponds to a normal or anomalous state. In addition, PREPARE can carry out prevention actions, such as increase the memory allocation of a particular virtual machine, to prevent the system from reaching the predicted anomalous state. PREPARE relies on historical failure data and thus it cannot detect previously unseen anomalies.

UBL [21] is an unsupervised anomaly detection framework for cloud systems able to detect anomalous virtual machines by analyzing performance metrics such as CPU and memory utilization. UBL employs a self-organizing map [59] to build a representation of the system behavior using normal data. Then, UBL raises an alert if the system deviates from what the self-organizing map considers normal. Since UBL relies on a definition of normality, it can detect previously unseen anomalies. However, UBL cannot adapt to changes in normal behavior as continuous updates degrade the quality of the the self-organized map.

CloudPD [60] is an anomaly detection and remediation framework devised for cloud infrastructures. CloudPD's anomaly detection engine has two stages. In the first stage, an anomaly detector is used to find deviations in virtual machines' performance metrics based on a model of normal recent behavior. Metrics with significant deviations are further analyzed in the second stage where the anomaly is confirmed if there is a significant change in the correlation of the candidate metric and other metrics of the same virtual machine, or other virtual machines running the same application. Since CloudPD employs a model of normal behavior based on recent history, it is able to detect previously unseen anomalies and can adapt to changes in normal behavior.

### 2.1.1 Key Issues

Even though there have been significant advances in anomaly detection in distributed systems over the years, key challenges remain. We identify the following main issues in existing works:

**Use of historical data:** Some of the existing works [29, 33, 56] rely on historical failure data to detect anomalies. However, historical failure data is unavailable in most real world scenarios [21, 48]. Moreover, obtaining samples of all the anomalies that can take place in a distributed system is impossible [19]. Thus, works that rely on historical failure data are limited to the detection of well-known anomalies, or anomalies that exhibit well-known patterns.

**Time awareness:** Many anomalies can only be detected when observed over a time period rather than at a time instant [61]. However, some works do not analyze the system over time [21, 32]. This means that these works can only detect anomalies characterized by a single anomalous data point (i.e., point anomalies), and not other more complex anomalies such as periodic patterns.

**Multi-metric:** Some anomalies are characterized by an anomalous combination of multiple performance metrics. For example, high CPU utilization might be normal if the system workload is also high and abnormal otherwise [62], where the system workload can be characterized by the number of requests received per time unit. Another example are anomalies where different metrics that should be independent from each other end up being correlated [20]. Existing works [29, 32, 33, 40, 41, 56, 60] that do not take into account metric correlations cannot detect these kind of anomalies.

**Adaptiveness:** The behavior of a running system changes over time. Therefore, an effective anomaly detection framework needs to be able to adapt to changes in behavior by dynamically incorporating new data into its statistical model. Some of the existing approaches do not implement this kind of adaptiveness [29, 21, 33].

**Progressive changes:** Some works [32, 41, 60] detect anomalies when a significant difference between current monitored data and recent past data is found. These approaches might find difficulties in detecting anomalies that exhibit a progressive change in the monitored metrics, such as a trend [23], since the difference between current readings and the recent past is never significantly high.

Table 2.1 shows a comparison of existing anomaly detection methods in the area of distributed systems. Previously unseen refers to approaches that do not rely on historical anomalous data and thus can detect previously unseen anomalies. As it can be seen, most approaches have issues that limit their anomaly detection capabilities. In fact, only one of the existing works meets all the considered criteria [38]. The most critical point is the inclusion of the correlations between different system metrics in the analysis, which is only accomplished by three of the reviewed works [21, 38, 41].

**Table 2.1.** Comparison of anomaly detection methods in the area of distributed systems.

| Approach | Previously Unseen | Time-Aware | Multi-Metric | Adaptiveness | Progressive Changes |
|---|---|---|---|---|---|
| Tiresias [32] | ✓ | | | | |
| Ozonat [38] | ✓ | ✓ | ✓ | ✓ | ✓ |
| Cherkasova et al. [56] | | ✓ | | ✓ | ✓ |
| Gu and Wang [33] | | ✓ | | | ✓ |
| EbAT [41] | ✓ | | ✓ | ✓ | |
| Wang et al. [40] | ✓ | ✓ | | ✓ | ✓ |
| PREPARE [29] | | ✓ | | | ✓ |
| UBL [21] | ✓ | | ✓ | | ✓ |
| CloudPD [60] | ✓ | ✓ | | ✓ | |

## 2.2 Anomaly Detection in Large-Scale Systems

Large-scale systems are a particular class of distributed systems, however, a precise definition of what constitutes a large-scale system does not exist. Nevertheless, the literature generally considers systems composed of hundreds to thousands of computers as large-scale [34, 35, 39]. Since large-scale systems are also distributed systems, some of the error detection approaches outlined in Section 2.1 can be applied to them. However, none of these approaches evaluate their methods in a specifically large scenario. In this section, we review the works in anomaly detection that experiment with systems of more than one hundred computers.

Tan and Wang [39] propose a context-aware anomaly detection framework for stream processing systems called ALERT. Using a Bayesian model, ALERT first determines the probability that a running system is in a particular execution context.

ALERT then makes use of a context-specific classifier to raise an alert if the system is in a state that can lead to a failure. Both context generation and anomaly detection are based on historical failure data. ALERT analyzes the system behavior in a time period, however, it does not take into account the correlations between the different system metrics, and relies on historical anomalous data. Tan and Wang evaluate ALERT in a system composed of 250 servers, where 20 metrics are monitored per server.

Lan et al. [35] present a method for detecting anomalous nodes in large-scale systems where all computers exhibit the same behavior. Their method consists of collecting a set of performance metrics from the computers in the system, reduce the dimensionality of the data, and label as anomalous the computers that deviate from the majority. Lan et al. explore principal component analysis (PCA) [63] and independent component analysis (ICA) [64] as dimensionality reduction techniques, and a cell-based algorithm to find computers that deviate from the majority. Since Lan et al.'s method relies on deviations from the majority, it does not require historical data, and can adapt to changes in normal behavior. Lan et al. evaluate their method in a cluster composed of 887 nodes, where they collect 19 metrics per node.

Fu et al. [43] present a framework for anomaly detection in cloud systems. This framework is composed of two adaptive support vector machine classifiers [65] that can learn new system information as it becomes available. One of the classifiers is one-class and the other is binary. Similar to Lan et al.'s approach, this framework employs ICA to reduce the dimensionality of the data. Anomaly detection is achieved by weighting the outputs of the two classifiers based on a credibility score and the proportion of abnormal data that has been observed. The credibility score is adjusted based on the number of errors that each classifier perform over time, and the more abnormal data, the more weight is given to the binary classifier.

Fu et al.'s framework improves its accuracy as new abnormal data is obtained at run time, however, thanks to the one-class classifier, the framework can theoretically detect previously unseen anomalies. Fu et al. run their experiments in a system composed of 362 servers, where 518 overall system metrics are collected.

Guan et al. [30] propose an anomaly detection method for cloud systems that is based on wavelet analysis of the system metrics in a sliding time window. The method first finds the subset of system metrics that better represent normal and anomalous behaviors. Then, a wavelet decomposition procedure and a neural network are employed on these metrics to find anomalies. The process can incorporate new data at run time and thus can adapt to changes in behavior. However, the method depends on historical failure data to select the most relevant metrics and to carry out the wavelet decomposition. Guan et al. experiment with 362 nodes and 518 metrics.

Guan and Fu [66] present a method for identifying anomalies in cloud infrastructures. This method employs PCA to select the system metrics that are more correlated with failures. A Kalman filter [67] is then employed to estimate the value of the selected metrics at a time point. An anomaly is detected if the difference between the estimation and the actual value is greater than a predefined threshold. Guan and Fu's method can adapt to changes in behavior by updating the selected metrics when new information is obtained. However, since metric selection depends on previously seen failures, the method cannot detect novel anomalies that affect other system metrics. Guan and Fu experiment with 362 nodes and 518 metrics.

Yu et al. [34] propose an anomaly detection framework for large-scale infrastructures. Yu et al. first group nodes together based on their geographical location and the network topology. Their framework then clusters together nodes of similar behavior within each group, and employs a two-phase majority voting algorithm to detect anomalous nodes in each cluster. The framework is able to detect previously unseen anomalies as long as these deviate from the behavior of the majority of the nodes. However, it is not clear if node clustering can be updated at run time if the behavior of the system changes. Yu et al. experiment in a cluster of 6,400 computers.

### 2.2.1 Key Issues

Some of the limitations that we encounter in black box anomaly detection for large-scale systems are the same as those outlined in Section 2.1. That is, some of the existing works do not consider the evolution of the system metrics over time [39, 43], others do not take into account the correlation between the system metrics [66], others are not adaptive [39], and others rely on historical failure data [30, 39, 66].

Moreover, other methods [34, 35] assume that most computers in the system exhibit similar behavior, and detect those that deviate from the majority. These works have a different objective to the work presented in this thesis in that they are more focused on high performance clusters, where nodes can be easily grouped by similarities in their behavior. Conversely, we provide a more universal approach, that does not rely on this assumption and is architecture agnostic.

**Table 2.2.** Comparison of anomaly detection methods for large-scale systems.

| Approach | Previously Unseen | Time-Aware | Multi-Metric | Adaptiveness | Heterogeneous |
|---|---|---|---|---|---|
| ALERT [39] | | | ✓ | | ✓ |
| Lan et al. [35] | ✓ | ✓ | ✓ | ✓ | |
| Fu et al. [43] | ✓ | | ✓ | ✓ | ✓ |
| Guan et al. [30] | | ✓ | ✓ | ✓ | ✓ |
| Guan and Fu [66] | | ✓ | | ✓ | ✓ |
| Yu et al. [34] | ✓ | ✓ | ✓ | | |

Table 2.2 compares the different works in anomaly detection for large-scale systems. Heterogeneous refers to works that do not assume homogeneity in the behavior of the different system components. We can see that none of the reviewed approaches meets all the considered criteria. However, in this case, most approaches take into

account metric correlations, and the major issue is the detection of previously unseen anomalies, which is only accomplished by three approaches [34, 35, 43].

## 2.3 Malicious Traffic Detection

In the area of security, error detection has been employed for the detection of different malicious activities, such as intrusions or viruses [22]. A portion of the research in security has also focused on the detection of botnets [27], one of the most serious threats due to their pervasiveness and harmful capabilities.

We tackle the anomaly detection problem in this particular area because botnet detection is affected by challenges similar to those in anomaly detection in large-scale distributed systems, such as the need to process large amounts of information, and the difficulty in obtaining historical data. Moreover, existing botnet detection works share many of the key issues outlined in Sections 2.1 and 2.2.

Anomaly detection has been applied to botnet detection from two perspectives: host-based and network-based detection [27]. Host-based detection methods focus on detecting malicious executables in individual computers, whereas network-based methods focus on detecting malicious computers by analyzing network traffic (i.e., network packets). In this section we review the most relevant network-based botnet detection approaches, as these are more similar to our work in that a network of computers is also a distributed system.

Strayer et al. [68] propose a botnet detection method to identify Internet Relay Chat (IRC) [69] based bots in a network. Their method first filters out well-known traffic, employs a classifier to identify chat communications, and then uses a correlation engine to find hosts with suspiciously similar behaviors.

Karasaridis et al. [42] present another method for detecting IRC bots from a set of suspicious hosts. Their method employs a series of statistical techniques on the network traffic to find activities that can be related to botnets, such as receiving many connections from suspicious hosts.

Strayer et al. and Karasaridis et al. provide botnet detection approaches that heavily rely on the IRC protocol. This is because IRC used to be the most popular protocol for botnet communication. However, current botnets employ other communication methods and architectures to make their detection more difficult.

BotHunter [70] is a bot detection system that relies on various intrusion detection techniques and a correlation engine. BotHunter first employs malware signatures, payload analysis and predefined rules to log different suspicious activities, such as port scanning or binary downloading. Then, BotHunter uses a correlation engine to flag hosts that carry out several of these suspicious activities in a specific order, or within a specific time frame. Since BotHunter is based on detecting malicious activities, it does not depend on a specific communication protocol or botnet architecture. However, since BotHunter employs signatures, it cannot detect previously unseen botnets.

BotSniffer [36] is a bot detection framework focused on the detection of IRC and HTTP botnets by finding hosts with similar behavior. BotSniffer is based on the fact that bots typically communicate with their bot master at the same time and in similar ways. BotSniffer employs two main detection algorithms. The first algorithm analyzes *response crowds*, that is, groups of hosts that reply to a message more or less at the same time, whereas the second algorithm analyzes the homogeneity of these crowds, that is, the similarity of the responses produced by the different hosts. BotSniffer does not rely on historical botnet data and thus can detect novel botnets as long as bots behave in a synchronized manner.

BotMiner [71] is a botnet detection framework that looks for hosts that perform suspicious activities, and with similar communication patterns. To achieve this, BotMiner carries out two clustering processes. The first process groups hosts performing suspicious activities, such as port scanning or binary downloading. The second clustering process groups hosts with similar communication patterns. BotMiner then computes a botnet score by correlating the different clusters obtained in the two processes. BotMiner is protocol and architecture agnostic, and can detect previously unseen botnets. However, BotMiner requires the presence of more than one bot of the same type in the monitored network, as it can only detect bots that exhibit synchronized behavior.

BotGAD [37] is a botnet detection framework that, like BotMiner, looks for group activities in a network (i.e., computers with similar communication patterns). BotGAD mainly focuses on the analysis of DNS traffic, and can differentiate between malcious and legitimate group activities.

Yen and Reiter [72] propose a method to differentiate legitimate peer-to-peer (P2P) traffic from botnet P2P traffic. Their method is based on the observations that P2P botnet traffic is less voluminous, more constant, and less random. Yen and Reiter's method detects malicious hosts by analyzing certain characteristics of the network traffic, such as the average number of bytes transmitted or the number of IP addresses contacted. The method heavily relies on specific characteristics of existing botnets.

BotGrep [28] is a P2P detection algorithm based on graph analysis. BotGrep finds P2P nodes by analyzing the mixing time of the network communication topology through random walks. BotGrep is a first step in the detection of P2P nodes that needs to be extended with another detection technique, such as the one proposed by Yen and Reiter [72], to differentiate legitimate from malicious P2P traffic.

Yu et al. [73] propose an online botnet detection method, which consists of finding similarities in network flows. Network flows are groups of packets with the same source and destination IP addresses and ports. Network flows are compared to each other by transforming them to the frequency domain by means of a discrete Fourier transform, and computing their Euclidean distance. After a set of suspicious hosts has been identified, a more localized network analysis is employed to detect hosts performing malicious activities such as port scanning. Since Yu et al.'s method analyzes network packets at the transport layer, it is protocol and architecture agnostic.

Moreover, it can detect previously unseen botnets, as long as bots exhibit synchronized behavior.

Zhao et al. [31] propose a botnet detection method based on the classification of network flows. Zhao et al. analyze certain flow characteristics, such as number of packets exchanged or average payload size, to categorize flows into normal or malicious by means of a decision tree classifier. Zhao et al.'s method is protocol and architecture agnostic, and does not require the presence of group activities to detect botnets. However, their method cannot detect previously unseen botnets as it relies on historical data.

Beigi et al. [1] present an extensive study on feature selection for the detection of botnets. In their study, Beigi et al. classify network flows using a decision tree classifier, and find that the best features to differentiate legitimate from malicious traffic are average packet length, flow duration, average bits per second, and percentage of small packets exchanged. Beigi et al.'s method is protocol and architecture independent but, similar to Zhao et al.'s approach, their method relies on historical botnet data and thus cannot detect previously unseen botnets.

Finally, PsyBoG [74] is a botnet detection framework based on DNS analysis. PsyBoG monitors DNS queries looking for extensive usage, periodic patterns and group activities. PsyBoG employs a significant peak detector to find periodic behaviors in DNS traffic, and a group activity analyzer to find similarities in DNS traffic. PsyBoG can detect previously unseen botnets as long as these employ the DNS protocol in a suspicious manner.

### 2.3.1 Key Issues

Existing network-based botnet detection approaches employ a wide range of tools and exploit different botnet characteristics to achieve their goal. However, there are still a series of limitations that are more or less common among most existing works. These limitations are:

**Use of historical data:** methods that rely on historical botnet data [1, 28, 31, 72] have difficulties in detecting previously unseen botnets and, similar to protocol specific approaches, are at risk of becoming obsolete as botnets evolve and modify their communication patterns.

**Protocol specific:** works that are protocol or architecture specific [28, 36, 37, 72, 74] are limited to the detection of botnets that use these protocols and architectures. This means that these approaches are at risk of becoming obsolete as new botnets that employ other communication strategies emerge.

**Group activities:** assuming group activities in a botnet is a strong assumption because the main purpose of these networks is to carry out synchronized malicious actions. Nevertheless, works that rely on group activities [36, 37, 71, 73, 74] require the presence of more than one bot of the same type in the monitored network, which may not be always the case.

Our framework overcomes these three main limitations by employing a protocol agnostic behavioral representation, not making any assumptions on the botnet behavioral characteristics, and not relying on historical botnet data. In this manner, our framework is able to detect previously unseen botnets with high accuracy, and with a reduced risk of becoming obsolete as new types of botnets appear.

Table 2.3 presents a comparison of methods in the area of malicious traffic detection. Heterogeneous refers to methods that do not rely on bots having similarities in their behavior to be able to detect them. The area of malicious traffic detection is slightly different to the area of anomaly detection in distributed systems because malicious traffic is easier to define than performance anomalies. In malicious traffic detection there are three main groups of approaches. The first group encompasses approaches that are heavily tailored for a particular type of botnet [42, 68]. These are the first botnet detection methods that were proposed, cannot typically detect previously unseen botnets, and are protocol specific.

The second group is formed by approaches that are protocol agnostic but heavily rely on group activities [70, 71, 73]. These approaches can usually detect previously unseen botnets because their analysis is based on finding behavioral similarities between hosts (assuming that novel botnets exhibit group activities).

The third group contains other more recent approaches that analyze network flows regardless of the protocol, architecture, and communication similarities [1, 31]. These approaches are more similar to existing works in the area of distributed systems in the sense that are purely focused on data analysis, and do not consider suspicious communication mechanisms, protocols, or architectures. These approaches also present similar issues to anomaly detection methods in distributed systems, such as relying on historical botnet data or the lack of adaptiveness.

**Table 2.3.** Comparison of anomaly detection methods in the area of botnet detection.

| Approach | Previously Unseen | Protocol Agnostic | Heterogeneous |
|---|---|---|---|
| Strayer et al. [68] | | | |
| Karasaridis et al. [42] | | | |
| BotHunter [70] | | ✓ | |
| BotSniffer [36] | ✓ | | |
| BotMiner [71] | ✓ | ✓ | |
| BotGAD [37] | ✓ | | |
| Yen and Reiter [72] | | | |
| BotGrep [28] | | | |
| Yu et al. [73] | ✓ | ✓ | |
| Zhao et al. [31] | | ✓ | ✓ |
| Beigi et al. [1] | | ✓ | ✓ |
| PsyBoG [74] | ✓ | | |

## 2.4 Analysis of Anomaly Detection Approaches

In this section we present a comprehensive comparison of the different anomaly detection approaches reviewed in this chapter. The comparison, which can be seen in Table 2.4, is based on the main challenges in anomaly detection identified in Chapter 1, that is, the definition of anomalies, the boundary of normality, and scalability. The main issues identified in Sections 2.1, 2.2, and 2.3 are a direct consequence of these three challenges. Therefore, we map each issue to the corresponding challenge and employ them as the comparison criteria. In the following, we detail each of these criteria.

### 2.4.1 Definition of Anomalies

As explained in Chapter 1, obtaining a precise definition of the anomalies in a system is difficult because its number is typically very large (and unknown), and because most anomalies have never been seen before they appear. This leads to anomaly detection approaches to target specific types of well-known anomalies, or anomalies that exhibit certain well-known patterns. We employ the following criteria to compare what types of anomalies different approaches are able to detect:

**Previously Unseen:** we consider that a method is able to detect previously unseen anomalies (or botnets) if it does not rely on historical failure data or anomalous signatures.

**Time-Aware:** time-aware applies to approaches in the area of distributed systems that analyze the system over a time period, as many anomalies can only be detected in this manner [61]. Time-awareness is not crucial for malicious traffic detection and thus we consider it not applicable in this domain.

**Multi-Metric:** multi-metric applies to approaches that take into account the correlation between multiple system metrics to detect anomalies. Like time-awareness, this is not essential for malicious traffic detection and thus we consider it not applicable in this domain.

**Progressive Changes:** some anomaly detection approaches can only detect anomalies that are significantly different from recent past data. These approaches cannot detect anomalies that exhibit a progressive change in the system metrics. This is also not applicable to malicious traffic detection.

**Heterogeneous:** this includes approaches that can detect anomalies in heterogeneous conditions as opposed to relying on some kind of homogeneity. For example, in distributed systems, approaches that assume that different computers behave in a similar manner are not heterogeneous. In malicious traffic detection, we do not consider as heterogeneous the approaches that require group activities to detect botnets.

**Protocol Agnostic:** this only applies to malicious traffic detection, and refers to approaches that are not tied to a specific network protocol.

**Architecture Agnostic:** we consider as architecture agnostic the approaches that can detect anomalies regardless of the architecture of the system under study or botnet.

In Table 2.4 we see that very few approaches [38] employ a universal definition of anomalies that can be used to detect previously unseen anomalies without making any assumption on their characteristics. These assumptions occur especially in the area of malicious traffic detection because botnets can be defined more easily than performance anomalies. However, even in botnet detection, making assumptions on the characteristics of the anomalies limits the detection capabilities of existing works, especially when dealing with previously unseen anomalies.

### 2.4.2 Normality Boundary

The second challenge identified in Chapter 1 is the definition of a boundary of normality. This is difficult because there can be multiple definitions of what is normal, these definitions can evolve over time, and may depend on a particular context. We employ two criteria related to the boundary of normality:

**Adaptive:** we analyze if existing works can incorporate new data into their definition of normality as it becomes available at run time. Incorporating new data is crucial to adapt to possible changes in the normal behavior of the system under study.

**Contextual:** we analyze if the different approaches take into account contextual information when defining what is normal in the system. This does not apply to malicious traffic detection approaches as context is not usually as relevant as in the distributed systems domain.

Table 2.4 shows that while some approaches are adaptive, very few take into account contextual information [38, 39, 40]. Moreover, none of the malicious traffic detection works are adaptive. This is because most of these works rely on a definition of anomalies (i.e., botnet traffic) more than on a definition of normal traffic. Nevertheless, incorporating new data to the definition of normality as it becomes available can help in reducing the amount of false positives in cases where normal and anomalous behaviors are very similar.

### 2.4.3 Scalability

The third challenge identified in Chapter 1 is scalability. We analyze the scalability of existing works from two perspectives:

**Discussed:** this includes approaches that discuss the problem of scalability, and that propose a design that can be considered scalable (e.g., a decentralized architecture).

**Experimental:** this includes approaches that have actually been tested in large-scale scenarios, that is, more than one hundred nodes for distributed systems, and millions of network packets for malicious traffic detection.

In Table 2.4 we can see that numerous approaches are concerned with scalability, and that several works employ large-scale datasets. However, very few of the reviewed works [34] have been tested in a real world large-scale scenario with thousands of computers. Moreover, none of the works presents a comprehensive scalability analysis on their methods.

Table 2.4 also shows that, even though anomaly detection is tackled in different ways depending on the area of application, there are key issues shared among the different areas. These issues are the result of the main challenges in anomaly detection outlined in Chapter 1, and overcoming them in one application domain can bring significant improvements in other domains as well. For example, Table 2.4 shows that a global definition of anomalies that is system independent can greatly improve the state-of-the-art in anomaly detection in multiple domains, as all the reviewed works rely on some specific characteristic of the system under study. The same occurs with scalability, which has become essential for anomaly detection due to the size of current computer systems and networks.

**Table 2.4.** Comparison of anomaly detection methods in the domains covered by this thesis. a–Strayer et al. [68], b–Tiresias [32], c–Karasaridis et al. [42], d–BotHunter [70], e–Ozonat et al. [38], f–Cherkasova et al. [56], g–BotSniffer [36], h–BotMiner [71], i–Gu and Wang [33], j–BotGAD [37], k–EbAT [41], l–ALERT [39], m–Lan et al. [35], n–Yen and Reiter [72], o–BotGrep [28], p–Yu et al. [73], q–Wang et al. [40], r–PREPARE [29], s–UBL [21], t–Fu et al. [43], u–CloudPD [60], v–Guan et al. [30], w–Guan and Fu [66], x–Zhao et al. [31], y–Yu et al. [34], z–Beigi et. al [1], $\alpha$–PsyBoG [74]

Legend: •–Fulfills, ○–Not applicable

| . | | Approaches | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Criteria | | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | $\alpha$ |
| Anomalies | Previously Unseen | | • | | | • | | • | • | | • | • | | • | | | • | • | | • | • | • | | | | • | | • |
| | Time-Aware | ○ | | ○ | ○ | • | • | ○ | ○ | • | ○ | • | | • | ○ | ○ | ○ | • | • | | | • | • | • | ○ | • | ○ | ○ |
| | Multi-Metric | ○ | | ○ | ○ | • | | ○ | ○ | | ○ | | • | • | ○ | ○ | ○ | | | • | • | | • | | ○ | • | ○ | ○ |
| | Progressive Changes | ○ | | ○ | ○ | • | • | ○ | ○ | • | ○ | | • | • | ○ | ○ | ○ | • | • | • | • | | • | • | ○ | • | ○ | ○ |
| | Heterogeneous | | • | | | • | • | | | • | | • | • | | | | | • | • | • | • | • | • | • | • | | • | |
| | Protocol Agnostic | | ○ | | • | ○ | ○ | | • | ○ | | ○ | ○ | ○ | | | • | ○ | ○ | ○ | ○ | ○ | ○ | ○ | • | ○ | • | |
| | Architecture Agnostic | | • | | • | • | | | • | • | | • | • | | | | • | • | | | • | | | • | • | | • | |
| Normality | Adaptive | | • | | | • | • | | | | | • | | • | | | | • | | | • | • | • | • | | | | |
| | Contextual | ○ | | ○ | ○ | • | | ○ | ○ | | ○ | | • | | ○ | ○ | ○ | • | | | | | | | ○ | | ○ | ○ |
| Scalability | Discussed | | | • | | | | • | | | • | • | • | • | | • | | • | • | • | • | • | • | • | | • | | • |
| | Experimental | • | | | | | | • | • | | • | | • | • | | • | | | | | • | | • | • | • | • | • | • |

## 2.5 Summary

In this section, we have presented an extensive review of existing works in the area of anomaly detection applied to distributed systems, large-scale infrastructures, and malicious traffic detection. We have analyzed the most common limitations in these three domains, and explained how our general framework overcomes these limitations.

Even though the three domains have been studied independently in the literature (especially malicious traffic detection), the anomaly detection problem is essentially the same, and is subject to similar key challenges. In fact, the three domains share some of the identified limitations, as shown in our literature comparison (Section 2.4).

This thesis provides a general framework for anomaly detection that creates a behavioral representation of a running system, and that can detect behavior deviations regardless of the system under study or the nature of the analyzed metrics. This contributes to the state-of-the-art in anomaly detection, including many of its areas of application.

# Chapter 3

# A Framework for Behavior Identification without Historical Data

The probability of failures or malicious attacks increases as software systems become larger, more complex, and pervasive. Nevertheless, detecting and identifying anomalous behaviors in large-scale distributed systems with high accuracy remains as an open problem as seen in Chapter 2. This is because of key challenges such as the difficulty in processing large amounts of information, the decentralized nature of these kind of systems, the huge variety of anomalies that can appear, and the difficulty in characterizing these anomalies.

In this chapter, we present our black box framework for anomaly detection in large-scale distributed systems. Our framework has been designed to overcome many of the issues identified in Chapter 2, such as the lack of historical data, the diversity of distributed system architectures and anomalies, and the need for scalability. This is achieved by means of machine learning methods, and by providing several deployment settings that can be tailored to numerous scenarios and application domains.

## 3.1 Overview

As seen in Chapter 1, dependability is crucial for current distributed systems, which go from large-scale web services composed of numerous computers that communicate and work together to provide a service to users, to P2P networks where users share their resources to achieve a common goal. Even though different distributed systems have different purposes and characteristics, all need to ensure correct behavior, maximize efficiency and availability, protect the privacy of their users, and avoid malicious activities among other requirements. Failing to fulfill one or more of these requirements can cause economic losses [15], exposure of sensitive information [13] or, in extreme cases, loss of life [12]. For example, reports show that the cost of downtime in industry ranges from $100K to $540K per hour on average [75].

Threats to dependability range from software flaws to physical conditions (e.g., room temperature), to malicious attacks [16]. These threats can produce errors in the system, and errors end up causing failures [16]. The negative effects of system errors could be ameliorated with better monitoring tools that can quickly detect anomalies in the system [76]. In addition, it is important not only to detect the presence of an anomaly, but also to identify the anomaly type. This can potentially improve the dependability of the system as the root cause of problems can be more easily identified, without the need for the system expert to analyze large amounts of information [77].

Unfortunately, anomaly detection in large-scale systems is an arduous task with low efficiency [35]. On the one hand, analyzing a great number of components and interactions is difficult in terms of complexity and magnitude [61]. On the other hand, most of the anomalies that appear in running systems have not been seen before, and thus it is difficult to detect them, as historical failure data is unavailable [78].

As seen in Chapter 2, black box analysis is a popular method for anomaly detection in large-scale systems and networks. However, existing black box approaches still have limitations that must be addressed in order to detect anomalies with increased accuracy in real world scenarios. Key issues include the need for historical failure data, detecting only a particular class of anomalies, being architecture specific, and the lack of adaptiveness and scalability.

In this chapter we present a general framework for identifying anomalous behaviors in large-scale distributed systems. Our framework is designed to overcome the challenges faced by existing works by using a combination of several classifiers in a two-step process that allows the detection of previously unseen anomalies, as well as the identification of their type. Our framework builds and maintains a model of the system behavior at run time, and obtains relevant properties from this model that are used to classify the system behavior into normal or anomalous. This is done by means of several *support vector machines* (SVM) [65]. Since the behavioral model is built and updated at run time, our framework does not require historical failure data and can adapt to changes in behavior. Moreover, our framework does not depend on the architecture of the system under study, or makes any assumption on the characteristics of the anomalies, which means that can be employed in numerous scenarios and domains.

The rest of this chapter is organized as follows. Section 3.2 presents the different behavioral perspectives that can be adopted when analyzing the behavior of distributed systems. Section 3.3 explains the considerations that have been taken in the design of our anomaly detection framework. Section 3.4 presents our framework in detail, and Section 3.5 concludes the chapter.

## 3.2 Characterizing System Behavior

A crucial step in black box anomaly detection is the characterization of system behavior, that is, how system behavior is represented using the available system information. The behavior of a system is defined as *“what the system does to implement its function and is described by a sequence of states”* [16], where the state of a system is the combination of its computation, communication, stored information, interconnection, and physical condition. The global behavior of a distributed system is the combined behavior of all of its components, where a system component refers to any hardware or software entity that is part of the system. For example, a campus network can be seen as a system of computers, switches, and routers that communicate with each other, whereas a multi-tier architecture can be seen as a system of software applications, such as databases and web servers. When a system component deviates from its intended behavior, we say that there is an error, an *anomaly*, or an *anomalous behavior* in the system. The level at which we observe system behavior depends on the characteristics of the system under study and on the type of anomalies that we are interested in detecting.

Existing works in anomaly detection employ various perspectives when modeling and analyzing the behavior of the system under study [29, 31, 21, 34]. Their choice of perspective is determined by the type of distributed system under analysis, its goals, as well as the type of anomalies to be detected. We classify these perspectives in three main categories: global behavior, system of systems and replicated behavior, as shown in Fig. 3.1.

A global behavior perspective can be adopted when we are interested in modeling the system behavior as a whole and in detecting deviations from this overall behavior [33, 40]. The system of systems perspective is typically employed when the system under study can be divided into subsystems from the point of view of its behavior [29, 21], and the replicated behavior perspective can be employed when we are interested in using a single model to characterize the behavior of various system components [34, 35].

**Global Behavior** In certain distributed systems there is a clear definition of global behavior, in the sense that system components cooperate with each other to achieve a common goal, such as providing an external service. In these scenarios, we are usually

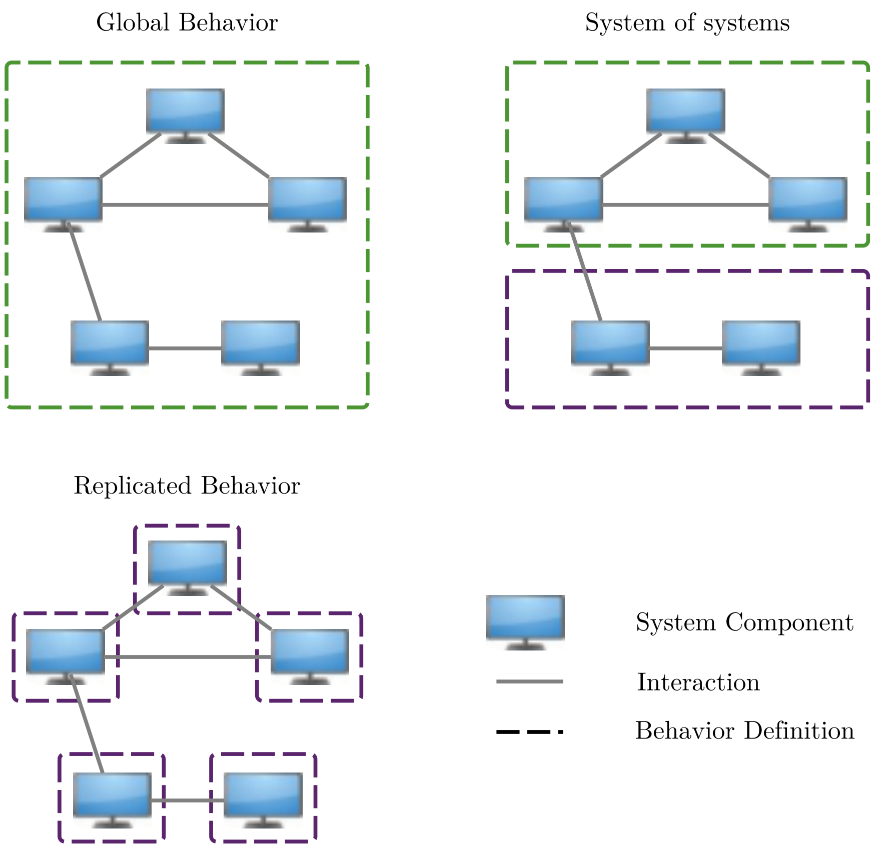


**Fig. 3.1.** Different perspectives that can be employed when modeling the behavior of distributed systems.

interested in ensuring that the system components work together in the expected way, and that the correct service is delivered. Thus, behavior is usually analyzed as the combined behavior of all the components in the system. Examples of these kind of systems are multi-tier architectures, where components such as databases, load balancers, web and application servers work together to deliver a service over the network. Global behavior is the most common perspective adopted by anomaly detection approaches in distributed systems [30, 32, 33, 39, 38, 41].

**System of Systems** A system of systems perspective is employed when we are interested in analyzing system behavior as a collection of subsystems. This perspective can be useful in cloud infrastructures, for example, where the behavior of each virtual machine can be modeled independently, and the system administrator can be alerted if any of the virtual machines deviates from the expected [29, 21]. The system of systems perspective can also be useful in large-scale scenarios such as HPC clusters, where the scalability of the anomaly detection process can be improved by independently analyzing groups of geographically close computers [34], instead of analyzing the behavior of the system as a whole.

**Replicated Behavior** A replicated behavior perspective means analyzing the behavior of the different system components using a single model. This is useful in cases where several components exhibit similarities in their behavior, as it enables the detection of components that deviate from the rest. This kind of behavioral perspective has been employed especially in HPC clusters, where different nodes can be assumed to behave in similar ways [34, 35].

## 3.3 Design Considerations

In Chapter 2 we present an analysis of current limitations in black box anomaly detection. These limitations include the use of historical failure data, focusing on specific types of anomalies, targeting specific architectures, and the lack of adaptiveness and scalability. Based on these limitations, we discuss below the main considerations that we have taken in the design of the framework proposed in this chapter.

**Historical Data** The basic idea behind most black box anomaly detection approaches is to build a statistical model that can separate normal and anomalous behaviors (see Chapter 2). This model is then employed to categorize the current system behavior into these two classes. This statistical model can be built using normal or anomalous historical data of the system. However, anomalous historical data is unavailable in most real world scenarios as mentioned in Chapter 1. Moreover, relying on anomalous historical data to build the model limits the ability to detect anomalies that are completely different from what has been observed in the past. For these reasons, we assume that historical data is unavailable, which means that our framework needs to build a system behavioral model at run time only using data available since the system started running.

**Time Awareness** Anomalies in distributed systems are characterized by an anomalous succession of states. Therefore, the statistical model employed in the detection of anomalous behaviors needs to include multiple metric readings over a time period to be able to detect all types of anomalies [61].

**Metric Correlations** Some anomalies are characterized by an anomalous correlation between several system metrics. For example, high CPU utilization might be normal if memory utilization is also high, and abnormal otherwise. Thus, our framework needs to model the behavior of the system while taking into account the correlation between the different system metrics.

**Architecture Independence** The system metrics that are relevant for anomaly detection depend on the characteristics of the system under study, and go from hardware counters to application metrics. For example, in the case of a P2P network, the analysis could include messages sent and received by each peer, number of peers contacted, number of bytes sent and received, etc. Conversely, in the case of an HPC cluster, the monitored metrics could be CPU, memory, and network utilization per node. The relevant metrics also depend on the type of anomalies that are to be detected. Thus, if the main goal of the analysis is the detection of malicious activities in a network, the most relevant metrics will be related to network usage. Our design must be independent from the choice of system metrics to allow our framework to be employed in a variety of scenarios, from large-scale networks to multi-tier architectures.

Additionally, as mentioned in Section 3.2, system behavior can be analyzed from several perspectives depending on the type of anomalies that are to be detected. Our framework also needs to be independent from the behavioral perspective, and needs to provide a flexible design that can be used to model the behavior of a variety of distributed systems regardless of their characteristics and architecture.

**Adaptiveness** The normal behavior of the system under study can change over time. Therefore, to adapt to these changes, our framework needs a mechanism to incorporate new data to the statistical model as it becomes available at run time. This mechanism is also necessary because we assume a lack of historical failure data.

**Context Awareness** Some anomalies depend on the context. For example, high CPU utilization can be normal in a high demand scenario, and abnormal otherwise. Thus, our framework needs a way of incorporating contextual information into the statistical model.

**Scalability** Scalability is essential as distributed systems become larger and more complex. Thus, our framework needs to be scalable to be employed in large-scale scenarios.

**Anomaly Identification** Identifying the type of the anomalies can help debugging and finding their root causes. Thus, apart from detecting anomalies, our framework needs to provide means for the identification of known anomalies.

## 3.4 Behavior Identification Framework

Fig. 3.2 shows our framework's design, including its main components and how they interact with each other. The framework is designed to detect previously unseen anomalies, and to identify the type of known anomalies while satisfying the

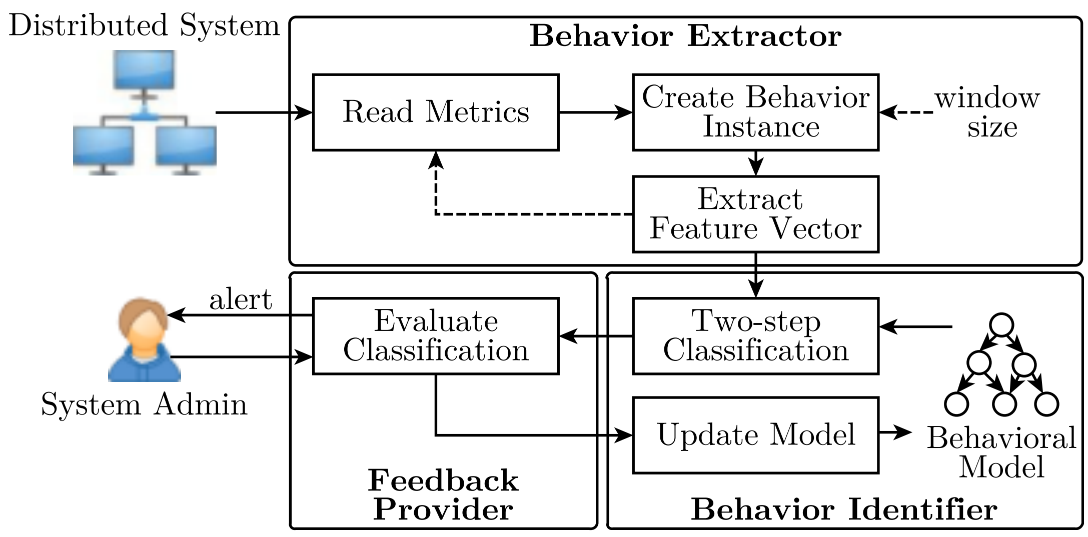


**Fig. 3.2.** Framework's design.

design considerations outlined in Section 3.3. Our framework is composed of three main components: the *Behavior Extractor*, the *Behavior Identifier*, and the *Feedback Provider*.

The *Behavior Extractor* periodically collects system performance metrics from a running distributed system and generates a *Behavior Instance* (BI) every time new data is available. BIs are representations of the system behavior in a time period. The *Behavior Identifier* classifies each BI as normal or anomalous using a statistical model called the *Behavioral Model* (BM). When a BI is classified as anomalous, the *Behavior Identifier* informs the *Feedback Provider*. The *Feedback Provider* then decides whether to alert the system administrator. When alerted, the system administrator informs the *Feedback Provider* about the actual state of the system (i.e., normal or anomalous). The *Feedback Provider* transmits this information to the *Behavior Identifier*, which can then update the BM to incorporate the new data. The internals of each component are detailed in the following sections.

### 3.4.1 Behavior Extractor

The *Behavior Extractor* periodically collects system performance metrics from a running distributed system. These metrics may be *hardware metrics*, such as the CPU, memory, or the network utilization of the system nodes, *operating system metrics*, such as the number of running processes or threads in each node, or *software metrics*, such as the number of exchanged messages between applications. These metrics can be represented as time series, where each data point is the value of a certain metric at a particular time instant. Thus, given a time frame, the behavior of the system can be defined by a set of time series. This set of time series is what we call a *Behavior Instance* (BI).

Formally, given the set of all the available system metrics

$$M = \{m_i \,|\, 1 \leq i \leq n\}, \tag{3.1}$$

we denote $r_{ij}$ as the reading for metric $m_i$ at time instant $j$, and

$$s_{ij} = r_{ij}, r_{i(j+1)}, \ldots, r_{i(j+z)} \tag{3.2}$$

as the sequence of $z$ readings for metric $m_i$ starting with $r_{ij}$. We then denote the BI at time instant $j$ as the set

$$B_j = \{s_{ij} \,|\, 1 \leq i \leq n\}, \tag{3.3}$$

a collection of $n$ time series of size $z$. Behavior Instances are thus created in a *sliding window* of constant size $z$, where $z$ determines the granularity of the analysis. Every time a new set of readings $\{r_{i(j+z+1)} \,|\, 1 \leq i \leq n\}$ is obtained, a new BI is generated as

$$B_{j+1} = \{s_{i(j+1)} \,|\, 1 \leq i \leq n\}, \tag{3.4}$$

where

$$s_{i(j+1)} = r_{i(j+1)}, r_{i(j+2)}, \ldots, r_{i(j+z+1)} \,. \tag{3.5}$$

The *Behavior Extractor* applies a feature extraction process to BIs after generating them. A feature is a function that derives some relevant property from a dataset. In this case, features extract relevant properties from the time series data such as mean, standard deviation, skewness, or kurtosis [79]. Computing these features serves two purposes. On the one hand, it provides a more robust way of comparing time series than straightforward differences between readings such as Euclidean distances [79]. On the other hand, it reduces the dimensionality of the data. After the feature extraction process, each BI is transformed into a *feature vector*. Feature vectors represent the behavior of the system during a time period in an $N$-dimensional space called the feature space. Detection of anomalous behaviors can then be done by detecting deviations in this feature space.

Formally, given a set of features

$$S_F = \{f_k \mid 1 \leq i \leq u\}, \tag{3.6}$$

where $f_k : \mathbb{R}^z \to \mathbb{R}^{d_k}$ is a feature that reduces the dimension of the data from $z$ to $d_k$, we denote

$$v_{ij} = f_1(s_{ij}), f_2(s_{ij}), \ldots, f_k(s_{ij}) \tag{3.7}$$

as the sequence of all the features in $S_F$ applied to the time series $s_{ij}$. We then define $F(B_j)$ as the concatenation of all the features applied to all the time series in $B_j$. Thus, the feature vector of $B_j$ is

$$F(B_j) = v_{1j}, v_{2j}, \ldots, v_{nj} = x_j \,, \tag{3.8}$$

$$
\begin{array}{lcccl}
B_j & & & & x_j \\
s_{1j} : r_{1j}, \dots, r_{1(j+z)} & \rightarrow & f_1(s_{1j}), f_2(s_{1j}), \dots, f_u(s_{1j}) & \rightarrow & v_{1j} \\
s_{2j} : r_{2j}, \dots, r_{2(j+z)} & \rightarrow & f_1(s_{2j}), f_2(s_{2j}), \dots, f_u(s_{2j}) & \rightarrow & v_{2j} \\
\vdots & & \vdots & & \vdots \\
s_{nj} : r_{nj}, \dots, r_{n(j+z)} & \rightarrow & f_1(s_{nj}), f_2(s_{nj}), \dots, f_u(s_{nj}) & \rightarrow & v_{nj}
\end{array}
$$

**Fig. 3.3.** Derivation process of feature vector $x_j$ from Behavior Instance $B_j$ and a set of features $S_F$. Each feature is applied to each sequence of readings in $B_j$, and the results are concatenated to form $x_j$.

and $F : \mathbb{R}^{n \times z} \rightarrow \mathbb{R}^N$, where $N = \sum_{k=1}^{u} d_k$. The feature extraction process is depicted in Fig. 3.3.

Feature vectors capture the evolution of the system metrics over time and the relationships between these metrics. Therefore, deviations in the feature space can be caused by a change in one of the metrics, as well as by changes in the correlation of multiple metrics. Moreover, contextual information can be easily included in the feature vector by representing context as a system metric. For example, a high demand context could be represented as the number of client requests received. Since the feature vector takes into account metric correlations, different contexts are automatically represented as different regions in the feature space, and anomalies characterized by deviations in a specific context can be detected.

### 3.4.2 Behavior Identifier

The *Behavior Identifier* is in charge of building and maintaining the *Behavioral Model* (BM), of deciding whether a feature vector is anomalous, and of identifying the type of detected anomalies. This is done by means of several classifiers. A classifier is a supervised learning method to categorize sets of data [80]. Typically, a classifier is first trained with a set of labeled data, that is, data whose category is known, and then used to categorize unlabeled data afterwards. The process of categorizing unlabeled data is also known as label prediction. Building the training dataset requires a system expert to label historical data belonging to the considered categories. However, as mentioned before, in our case we assume that historical data is unavailable, and make use of online classifiers that can build and update the BM at run time. The accuracy of an online classifier increases with the number of predictions, as the statistical model converges to an optimal representation of the data distribution [81]. The disadvantage of using online classifiers is that, for a period of time at the beginning, the predictions can be less accurate. The advantages are that no previous labeled data is needed, and that the BM can easily adapt to changes in the definition of normal behavior. This is relevant as a precise definition of normal and anomalous behavior of a system is hard or even impossible to obtain before the actual execution of the system. In the following we describe the technical details of the *Behavior Identifier*.

### Support Vector Machines

The *Behavior Identifier* employs a type of classifier called *Support Vector Machines* (SVMs), which are a widely used mechanism for classifying data [65]. Having a set of labeled vectors $L = \{(x_i, y_i) \mid 1 \leq i \leq l\}$, where $x_i \in \mathbb{R}^n$, and $y_i \in \{-1, +1\}$ represents two distinct categories, the training phase in an SVM consists of finding the optimal hyperplane that separates the two categories in an $n$-dimensional space.

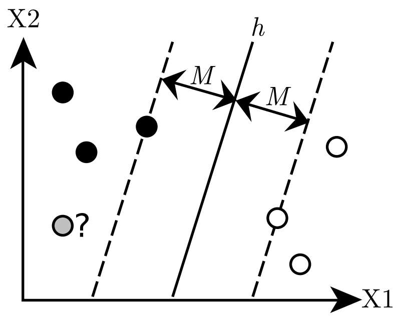


**Fig. 3.4.** Construction of a SVM in a 2-dimensional space.

Fig. 3.4 shows and example of an SVM in a 2-dimensional space, where X1 and X2 are the dimensions of the vectors, black and white circles represent labeled vectors belonging to two distinct categories, and the gray circle represents an unlabeled vector. The $h$ line represents the optimal separating hyperplane, which is optimal because is found by maximizing the distance $M$. The two dotted lines are called the margins, and the vectors lying on them are called the support vectors. After finding $h$, this SVM would classify the unlabeled vector as "black" according to its position with respect to $h$. Finding $h$ requires minimizing $||w||^2$, where $w$ is a vector normal to $h$. This minimization can be expressed as the optimization problem

$$\begin{gathered}\sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j x_i \cdot x_j \\ \text{subject to} \quad \alpha_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{l} \alpha_i y_i = 0 \,,\end{gathered} \tag{3.9}$$

where $\alpha_i$ and $\alpha_j$ are Lagrange multipliers and

$$w = \sum_{i=1}^{l} \alpha_i y_i x_i \,. \tag{3.10}$$

After solving the optimization problem, unlabeled vectors can be categorized according to their position in the $n$-dimensional space with respect to the hyperplane. The decision function for an unlabeled vector $x_j$ becomes

$$f(x_j) = \operatorname{sgn}\left( \sum_{i=1}^{l} \alpha_i y_i x_i \cdot x_j - b \right) \,, \tag{3.11}$$

where $b$ is a bias term that represents the distance from the hyperplane to the origin of the coordinate space.

For not linearly separable categories in the $n$-dimensional space, a kernel trick can be used by replacing the product $x_i \cdot x_j$ in Eq. (3.9) by a kernel function $K(x_i, x_j) \equiv \phi(x_i) \cdot \phi(x_j)$, where $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^m$ represents a transformation from the $n$-dimensional space to an $m$-dimensional space. This kernel function enables building the hyperplane in the $m$-dimensional space, where the two categories can be separated. A widely used kernel function is the *radial basis function* (RBF) kernel, which is defined as [82]

$$K(x,\, x') = \mathrm{e}^{-\gamma \|x-x'\|^2}, \tag{3.12}$$

where $\gamma$ is a free parameter and $\|x-x'\|^2$ is the *squared Euclidean distance* between $x$ and $x'$.

### Gradient Descent

Solving Eq. (3.9) given a training dataset yields the optimal hyperplane or decision function. In our case, we are interested in building and updating this decision function at run time. To build an online support vector machine that modifies its decision function whenever new observations become available, the *gradient descent* method can be used. Gradient descent is an algorithm for finding the minimum of a function in an iterative manner. We employ a modified version of the SVM proposed by Kivinen et al. [83]. The SVM optimization problem can be formulated as the loss minimization problem

$$\min \sum_{x \in S} \ell(f(x),\, y), \tag{3.13}$$

where $\ell(f(x), y)$ is the loss incurred when misclassifying $x$. Then, we can apply the gradient descent method to iteratively update the decision function $f$. Being $x_j$ a new observation, we have that

$$f_{j+1} = f_j - \eta \frac{\partial}{\partial f} \ell(f(x_j),\, y_j), \tag{3.14}$$

where $\eta$ is the *learning rate*, that is, the "speed" in which we approach the $f$ that minimizes the amount of classification errors. To apply this method, we can set

$\alpha_i = y_i \alpha_i$ in Eq. (3.11) and write $f_j$ as

$$f(x_j) = \operatorname{sgn}\left(\sum_{i=1}^{j-1} \alpha_i x_i \cdot x_j - b\right). \tag{3.15}$$

Then we can update the $\alpha$ coefficients at each iteration as

$$\alpha_i = \begin{cases} -\eta \ell'(f_j(x_i),\, y_i) & \text{for} \quad i = j \\ (1-\eta\lambda)\alpha_i & \text{for} \quad i < j\,, \end{cases} \tag{3.16}$$

where $\lambda$ is used decrease the weight of past observations, giving more importance to new data. As loss function, we can use the soft margin loss:

$$\ell_\rho(f(x),\, y) = \max(0,\, \rho - yf(x))\,, \tag{3.17}$$

where $\rho \geq 0$ is a margin parameter that controls the amount of margin errors permitted. This loss function then can be used in Eq. (3.16) with

$$\ell'_\rho(f(x), y) = \begin{cases} -y & \text{if } yf(x) \leq \rho \\ 0 & \text{otherwise}\,. \end{cases} \tag{3.18}$$

These equations build a classifier able to differentiate between two classes, that is, a binary classifier. With some modifications, we can also define a one-class classifier, which is a classifier that requires only labeled observations from one category to build the decision function. One-class classifiers build a sphere around the labeled observations, and categorize any observation that lies outside this sphere as anomalous. To build a one-class classifier we use the loss function

$$\ell_\rho(f(x)) = \max(0,\, \rho - f(x)) - \nu\rho\,, \tag{3.19}$$

where $\rho$ defines the size of the sphere, $y$ is dropped as there is only one class, and $\nu$ controls the frequency of anomalous classifications. Since

$$\frac{\partial \ell_\rho}{\partial f} = \begin{cases} -1 & \text{if } f(x) < \rho \\ 0 & \text{otherwise} \end{cases} \tag{3.20}$$

and

$$\frac{\partial \ell_\rho}{\partial \rho} = \begin{cases} 1-\nu & \text{if } f(x) < \rho \\ -\nu & \text{otherwise}\,, \end{cases} \tag{3.21}$$

we have that the updates for the $\alpha$ terms and $\rho$ are [1]

$$(\alpha_i, \alpha_j, \rho) = \begin{cases} ((1-\eta)\alpha_i,\ \eta\ ,\ \rho - \eta(1-\nu)) & \text{if } f(x) < \rho \\ ((1-\eta)\alpha_i,\ 0\ ,\ \rho + \eta\nu) & \text{otherwise}\,. \end{cases} \tag{3.22}$$

The decision function is then

$$f(x_j) = \operatorname{sgn}\left(\rho - \sum_{i=1}^{j-1} \alpha_i x_i \cdot x_j\right)\,. \tag{3.23}$$

**Data Normalization**

Normalizing the vectors before classification is crucial to avoid bias towards a particular dimension [84] because SVMs rely on dot products between vectors. For example, the dot product between $x = [0.3,\ 100]$ and $x' = [0.1,\ 200]$ is biased towards the second dimension as it takes much larger values. For this reason, the *Behavior Identifier* normalizes each vector component to the $[0,1]$ interval using a sigmoidal function. This type of function has the advantage that it is robust to the presence of extreme values in the data. The function is defined as

$$x'_i = \frac{1}{1 + \mathrm{e}^{\frac{x_i - \mu_i}{\sigma_i}}} \quad \text{for} \quad 1 \le i \le n\,, \tag{3.24}$$

where subindex $i$ in this case denotes the $i$th component of a vector, $x$ is the original vector, $x'$ is the scaled vector, and $\mu$ and $\sigma$ are the mean and standard deviation of

[1]In [83] there is a typo and the plus and minus signs of the $\rho$ update are swapped.

the set of seen vectors. Since the *Behavior Identifier* receives vectors one at a time, $\mu$ and $\sigma$ are computed also online. Initially, $\mu$ is set to the first received vector and $\sigma = [1, 1, ..., 1]$. Then, updates can be performed using

$$\begin{aligned} \mu_i' &= (1 - \frac{1}{t})\mu_i + \frac{1}{t}x_i \\ \sigma_i' &= (1 - \frac{1}{t})\sigma_i + \frac{1}{t}(x_i - \mu_i')^2 , \end{aligned} \tag{3.25}$$

where $x$ is the last received vector and $t$ is the number of vectors processed so far.

### Two-step Classification

The *Behavior Identifier* makes use of a combination of a one-class classifier and multiple binary classifiers to achieve two main goals: the one-class classifier is used to detect previously unseen anomalies without historical failure data, and a set of binary classifiers is used to identify the type of known anomalies. Algorithm 1 presents the pseudocode of the one-class classifier. Classes normal and anomalous are represented with $-1$ and $+1$ respectively.

The classifier contains two main methods: CLASSIFY and UPDATE_MODEL. CLASSIFY returns the predicted label (lines 7 and 9) of an unlabeled vector $x_j$ using the decision function in Eq. (3.23) (line 6). UPDATE_MODEL modifies the current decision function given a vector $x_j$ of the normal class ($-1$). If the kernel evaluation is less than $\rho$ (line 14), meaning that the current function misclassifies $x_j$, the vector is incorporated into the set of support vectors $V$ (line 15) with $\alpha_j = \eta$ (line 12), and $\rho$ is decreased (line 17). Here, instead of using $\rho = \rho - \eta(1 - \nu)$ as defined in Eq. (3.22), we use $\rho = \rho - \rho\eta(1 - \nu)$ to enforce $\rho > 0$, given that $\eta < 1$ and $\nu < 1$. Alternatively, $\rho$ is increased if the current function correctly classifies $x_j$ (line 19). This makes $\rho$ to approach its optimal value close to the kernel evaluation, ensuring that the size of the sphere around the normal data is minimal. Finally, the $\alpha$ values are also updated (line 22).

**Algorithm 1** Online One-class Classifier

```
1: function INIT
2:     $V \leftarrow \emptyset$          // Set of support vectors
3: end function
4:
5: function CLASSIFY($x_j$: Vector)
6:     if $\rho < \sum_{x_i \in V} \alpha_i K(x_i, x_j)$ then
7:         return -1
8:     else
9:         return +1
10:    end if
11: end function
12:
13: function UPDATE_MODEL($x_j$: Vector)
14:    if $\sum_{x_i \in V} \alpha_i K(x_i, x_j) < \rho$ then
15:        $V \leftarrow V \cup \{x_j\}$
16:        $\alpha_j \leftarrow \eta$
17:        $\rho \leftarrow \rho - \rho\eta(1-\nu)$
18:    else
19:        $\rho \leftarrow \rho + \rho\eta\nu$
20:    end if
21:    for all $\alpha_i \neq \alpha_j$ do
22:        $\alpha_i \leftarrow (1-\eta)\alpha_i$
23:    end for
24: end function
```

Algorithm 2 shows the pseudocode of the binary classifier, which is very similar to the one-class classifier. The CLASSIFY method uses the decision function in Eq. (3.15) (line 6), and the UPDATE_MODEL now makes use of $y_j$ to decide whether to update the decision function based on the soft margin loss in Eq. (3.17) (line 14). If the current function misclassifies $x_j$, the vector is added to $V$ (line 16) and $\alpha_j = \eta y_j$ according to Eq. (3.16) (line 15). Finally, the rest of the $\alpha$ coefficients are updated (line 19).

The binary classifier is designed to receive a similar amount of positive and negative vectors over time. Given an unbalanced dataset, the classifier is biased towards the most common class. This is obvious when $|V| = 1$, since the kernel evaluation in the CLASSIFY method (line 6) will always yield the sign of the only support vector in

**Algorithm 2** Online Binary Classifier

```
 1: function INIT
 2:     $V \leftarrow \emptyset$            // Set of support vectors
 3: end function
 4:
 5: function CLASSIFY($x_j$: Vector)
 6:     if $\sum_{x_i \in V} \alpha_i K(x_i, x_j) < 0$ then
 7:         return -1
 8:     else
 9:         return +1
10:     end if
11: end function
12:
13: function UPDATE_MODEL($x_j$: Vector, $y_j$: int)
14:     if $y_j \sum_{x_i \in V} \alpha_i K(x_i, x_j) \leq \rho$ then
15:         $\alpha_j \leftarrow \eta y_j$
16:         $V \leftarrow V \cup \{(x_j, y_j)\}$
17:     end if
18:     for all $\alpha_i \neq \alpha_j$ do
19:         $\alpha_i \leftarrow (1 - \lambda\eta)\alpha_i$
20:     end for
21: end function
```

the set. Moreover, successive calls to the UPDATE_MODEL method with vectors of the same type will produce no actual changes in the model, as the update condition (line 14) will be false with high probability after a few updates, and new vectors of the same type will not be included into $V$. To avoid a biased model in a scenario where most of the vectors are negative because distributed systems run most of the time in normal conditions, we update the binary classifier in a balanced manner. Algorithm 3 shows the pseudocode of the balanced update process.

If the set of support vectors is empty (line 7), the model is updated normally (line 8). Otherwise, if $x_j$ is negative and there are positive vectors stored in $P$ (lines 9 and 10), the model is updated with $x_j$ (line 11) and a random vector from $P$ (lines 12 and 13). If $x_j$ is negative but $P$ is empty (line 14), instead of updating the model, $x_j$ is stored in $N$ (line 15). The process when $x_j$ is positive is the same but swapping $N$ and $P$ (lines 18 to 24). This avoids successive updates with vectors of the same type

**Algorithm 3** Balanced Update Process

```
function INIT
    $P \leftarrow \emptyset$        // Set of positive vectors
    $N \leftarrow \emptyset$        // Set of negative vectors
end function

function BALANCED_UPDATE($x_j$: Vector, $y_j$: int)
    if $|V| = 0$ then
        UPDATE_MODEL($x_j$, $y_j$)
    else if $y_j < 0$ then
        if $|P| > 0$ then
            UPDATE_MODEL($x_j$, $y_j$)
            $x_p \leftarrow$ GET_RANDOM($P$)
            UPDATE_MODEL($x_p$, $+1$)
        else
            $N \leftarrow N \cup x_j$
        end if
    else
        if $|N| > 0$ then
            UPDATE_MODEL($x_j$, $y_j$)
            $x_n \leftarrow$ GET_RANDOM($N$)
            UPDATE_MODEL($x_n$, $-1$)
        else
            $P \leftarrow P \cup x_j$
        end if
    end if
end function
```

by storing them to be used in the future, and ensures that the model is updated in a balanced manner.

As mentioned before, the *Behavior Identifier* combines several classifiers to detect previously unseen anomalies and to identify the type of known anomalies. We call this two-step classification because *Behavior Identifier* decides whether a vector is anomalous in a first step and, in case it is, the *Behavior Identifier* identifies its type in a second step. This two-step process is done by means of a one-class classifier (OC), a binary classifier (BC), and a multi-class classifier (MC). The MC is in turn composed of multiple binary classifiers organized in a tree-like *directed acyclic graph* (DAG) [85].

Fig. 3.5 shows an MC able to identify four types of anomalies: deadlock, livelock, thrashing, and memory leak. Each node in the graph represents a binary classifier, and each of these binary classifiers discards one type of anomaly until the final result is obtained. For example, the node at the top of the graph decides if a vector is a memory leak or a deadlock. The classification process continues to the left if the vector *is not* a memory leak, and to the right if the vector *is not* a deadlock. When a new type of anomaly is found, the DAG is extended with a new level. There are $c-1$ levels given $c$ types of anomalies, and each level has one extra node than the previous level. Thus, the total number of nodes for $c$ types of anomalies is $\frac{c(c-1)}{2}$, and the number of classifications needed to obtain the final result is $c-1$. Each binary classifier in the MC maintains its own decision function, updating it online when new data becomes available. Given an anomalous vector of type $y_a$, the MC updates the decision functions of the nodes that compare $y_a$ against other types. This means a total of $c-1$ updates.

Fig. 3.6 depicts the two-step process that combines the OC, BC, and the MC. In the first step, the OC and the BC classify vectors as normal or anomalous. The OC serves to detect previously unseen anomalies while the BC is used to reinforce the detection

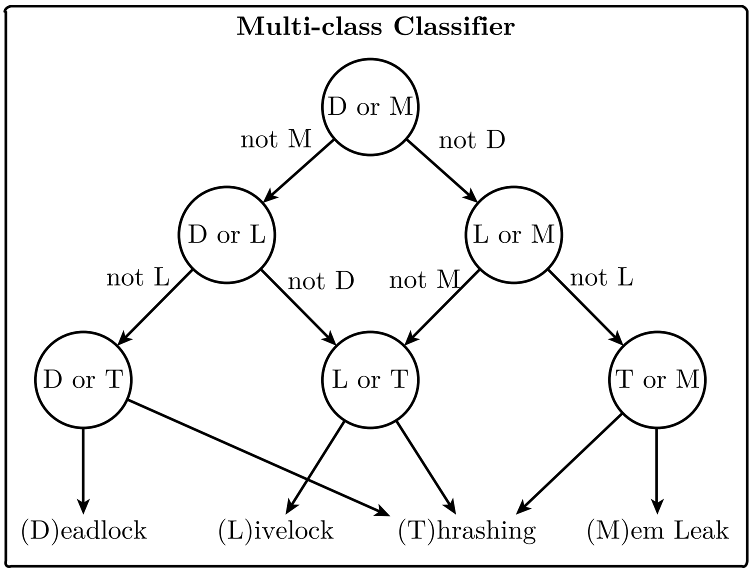


**Fig. 3.5.** DAG of binary classifiers to form a multi-class classifier.

of already known anomalies. The outputs of the OC and the BC are combined to produce a final prediction for step one. A simple approach is to combine the outputs using a logical *or*, and more advanced techniques could use weights based on the reliability of each classifier. The process finishes if the vector is classified as normal in this first step. If the vector is anomalous, the process moves on to step two, where the MC identifies the type of the anomaly. The total number of evaluations is therefore 2 for normal vectors and $c+1$ for anomalous vectors.

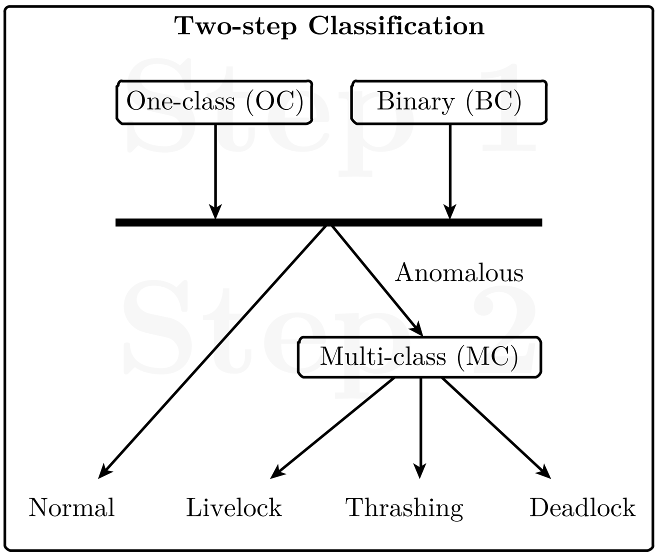


**Fig. 3.6.** Two-step classification process. Multi-class classification is provided by means of a directed acyclic graph of binary classifiers.

## Model Update

The *Behavior Identifier* transmits the result of the two-step classification to the *Feedback Provider*. The *Feedback Provider* then decides whether to alert the system administrator. If the system administrator is alerted and feedback is received, the *Behavior Identifier* updates the *Behavioral Model*. The BM is defined by the decision functions of the multiple classifiers that are involved in the two-step classification process. Given a vector $x_j$ with known type $y_j$, the OC's decision function is updated if $y_j = -1$, (i.e., $x_j$ is normal), and the MC is updated if $x_j$ is anomalous. The BC is updated in any case as it is used to differentiate between normal vectors and known anomalies. Each of the classifiers normalizes $x_j$ before updating the decision function, and also updates the normalizing function using Eq. (3.25). Note that each of the classifiers maintains its own normalizing function.

### 3.4.3 Feedback Provider

The *Feedback Provider* decides when to alert the system administrator based on the result of the two-step classification process. The *Feedback Provider* behavior is modeled as a *finite state machine* (FSM), which is shown in Fig. 3.7.

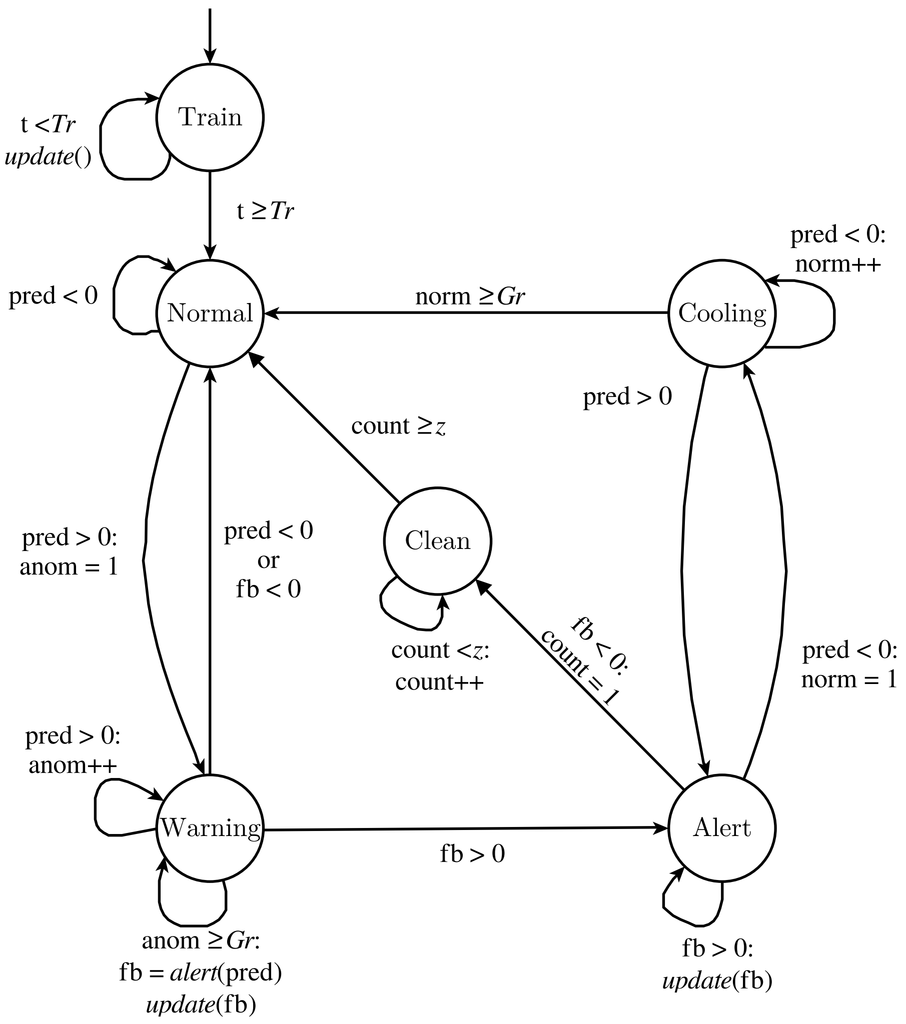


**Fig. 3.7.** *Feedback Provider* behavior modeled as a FSM.

In this FSM, the *pred* variable represents predictions received from the *Behavior Identifier*, and the $fb$ variable represents feedback received from the system administrator. A negative value in these variables means normal behavior, whereas positive values represent different types of anomalies. The *Feedback Provider* starts in the *Train* state. This state is completely optional and is used to train our framework if some labeled data is available, or if it can be guaranteed that the system will run normally for a certain amount of time at the beginning. The parameter $Tr$ controls the duration of this initial training phase, and the state is avoided if $Tr = 0$. After this, the *Feedback Provider* moves on to the *Normal* state, and remains in it as long as the predictions received from the *Behavior Identifier* are normal. The *Feedback Provider* moves to the *Warning* state if it receives an anomalous prediction. The *Feedback Provider* remains in the *Warning* state as long as the number of consecutive anomalous predictions (variable *anom*) is less than a predefined number, represented by the parameter *Gr*. If a normal prediction is received, the *Feedback Provider* goes back to the *Normal* state. The *Feedback Provider* raises an alert if it receives more than *Gr* consecutive anomalous predictions while in the *Warning* state. In this manner, *Gr* helps to reduce the number of false alerts caused by noisy readings. *Gr* defaults to 1, but it can be increased to tailor our framework to different scenarios.

When an alert is raised, the *Feedback Provider* requests feedback to the system administrator. The $update()$ method transmits this feedback to the *Behavior Identifier*, which can then update the *Behavioral Model* accordingly. The *Feedback Provider* moves back to the *Normal* state if the actual behavior of the system is normal (i.e., $fb < 0$). Otherwise, the *Feedback Provider* moves to the the *Alert* state. The *Feedback Provider* remains in the *Alert* state as long as the system administrator confirms the presence of the anomaly. If no feedback is given, the *Feedback Provider* returns to the *Normal* state through the *Cooling* state if *Gr* consecutive normal predictions are received. Conversely, if the system administrator confirms the end of the anomaly, the *Feedback Provider* moves to the *Clean* state. The *Feedback Provider* remains in this state for $z$ observations (i.e., the sliding window size), to ensure that no anomalous readings remain in the BIs once the *Feedback Provider* returns to the *Normal* state.

The feedback provided by the system administrator is crucial for our framework's performance. In the case of receiving incorrect feedback, our framework might include incorrect data into the *Behavioral Model*. Errors in the BM can be dynamically corrected with new data and thus are not permanent in any case. However, an inaccurate BM can produce false positives and false negatives. False positives are easy to correct because the system administrator is alerted, and has the opportunity to provide new feedback. Conversely, false negatives might remain undetected until a failure is observed, and positive samples can then be used to correct the BM.

### 3.4.4 Addressing Multiple Behavioral Perspectives

As mentioned in Section 3.2, distributed systems can be analyzed from several behavioral perspectives depending on their characteristics and the type of anomalies that we are interested in detecting. Our framework is designed to be independent from the architecture of the system under study and the system metrics being analyzed. We achieve this by building each framework component as a black box from the point of view of the other components, allowing for a highly flexible deployment. In the following, we describe how our framework can be employed to tackle the different behavioral perspectives detailed in Section 3.2.

#### Global Behavior

This perspective is employed when we are interested in analyzing a global behavior definition. Thus, we set our framework to collect performance metrics from all the system components and use these to build the BM, alerting the system administrator when the global behavior of the system deviates from what is expected. This is our framework's default setting, and is depicted in Fig. 3.8.

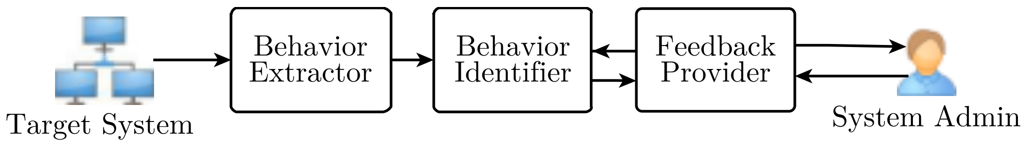


**Fig. 3.8.** Default setting used in global distributed system behavioral analysis.

### System of Systems

Our framework can be set to analyze the behavior of a system in a system of systems perspective as shown in Fig. 3.9. In this case, we deploy several instances of our framework's components. Each set of instances models the behavior of one of the subsystems. Before alerting the system administrator, feedback is aggregated using a *Feedback Aggregator*. The most basic *Feedback Aggregator* alerts the system administrator when any of the subsystems deviates from what is considered normal, however, other more complex policies can be implemented such as correlation analysis or weighted alerts. This approach can also be applied to large-scale systems where analyzing the behavior of the whole system is unfeasible due to computational or storage limitations.

The stars in Fig. 3.9 denote the fact that each set of framework component works independently from the others and builds a separate BM. Each of these BMs can employ different configuration settings and system metrics.

### Replicated Behavior

We can employ the setting depicted in Fig. 3.10 in scenarios where we are not interested in analyzing the global behavior of the system, but the behavior of the individual components from a shared perspective. In this setting, each system component is monitored independently, but the BM is shared among the different components. That is, several *Behavior Extractors* generate BIs corresponding to each system component. Then, each BI is classified using the same behavior definition, and the decision on whether to alert the system administrator is taken based on each component's trajectory.

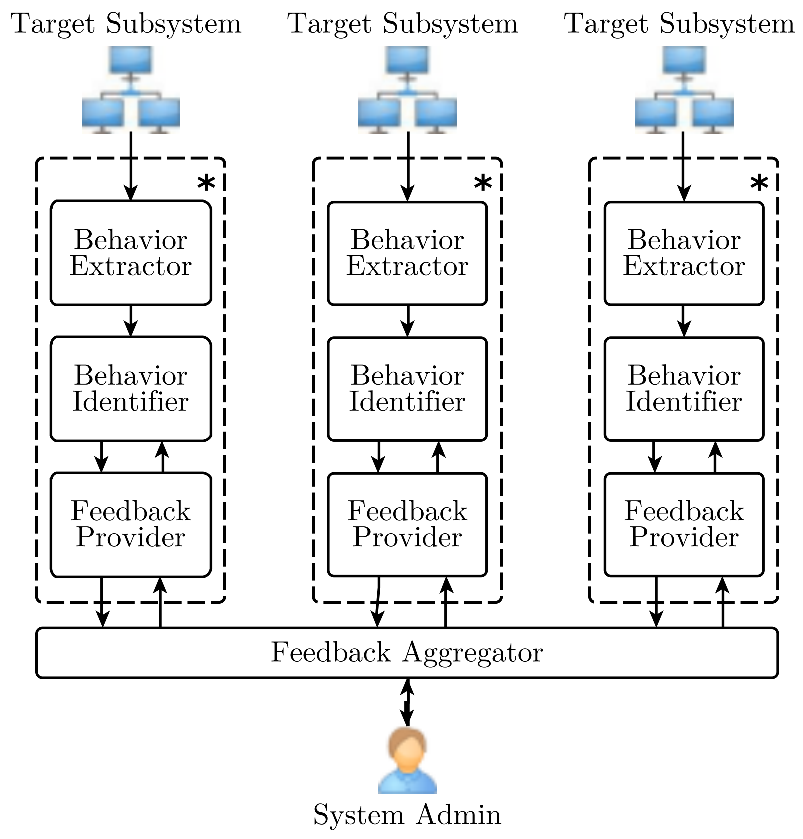


**Fig. 3.9.** Setting employed to analyze a distributed system from a system of systems perspective.

This architecture can be employed to detect misbehaving nodes in a network, or in scenarios where several computers are expected to behave in a similar manner, such as in an HPC cluster.

## 3.5 Summary

In this chapter, we presented our behavior identification framework for anomaly detection in distributed systems. Our framework analyzes the behavior of a running distributed system by periodically collecting system performance metrics. From these metrics, our framework builds a representation of the system behavior in a time frame taking into account the correlations between the different metrics. Then, using a combination of several SVM classifiers, our framework is able to detect anomalous

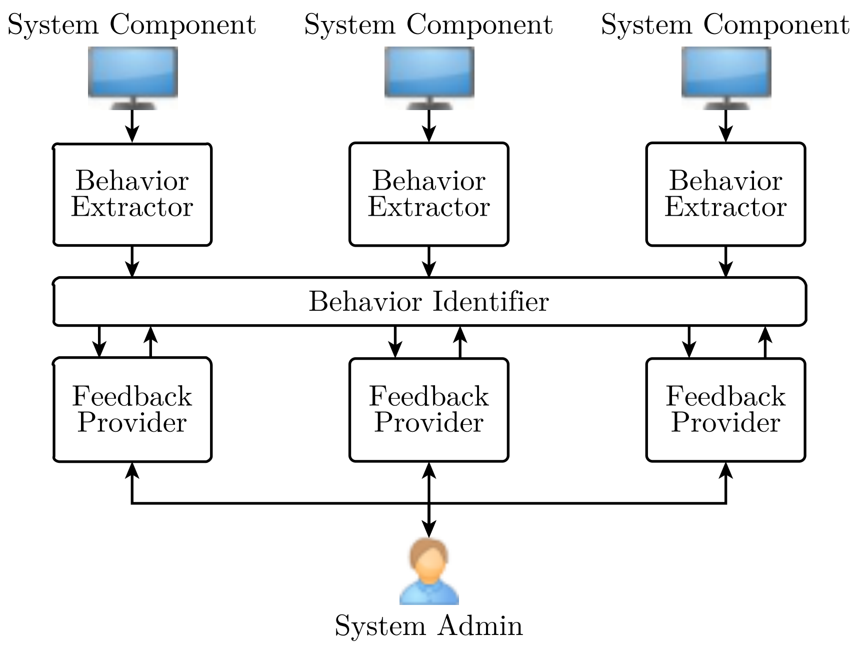


**Fig. 3.10.** Shared BM setting employed to analyze distributed systems from a replicated behavior perspective.

behaviors and to identify their type. Unlike similar existing approaches, our framework does not rely on historical data, does not make any assumptions on the characteristics of the anomalies, can be employed in a variety of scenarios regardless of the system architecture, can adapt to changes in the normal behavior of the system, and is scalable. To our knowledge, our framework is the only existing work that enables the detection and identification of anomalous behaviors in large-scale distributed systems that can be applied to most scenarios without system modifications or redeployment.

# Chapter 4

# A Synthetic Distributed System to Model Complex Anomalous Behaviors

In Chapter 3, we present our black box framework for anomaly identification in distributed systems. Our framework is architecture agnostic and has been designed to overcome many of the issues in the area of anomaly detection, such as the lack of historical data or the need for scalability. The three main components that form our framework, namely, *Behavior Identifier*, the *Behavior Extractor*, and *Feedback System*, can be configured with several parameters to adapt our framework to different scenarios and application domains. These configuration parameters are crucial for obtaining an increased detection accuracy. Thus, to be able to successfully employ our framework in the various scenarios where it can be applied, we first need to obtain a deep understanding on how these parameters affect our framework performance.

Towards this, we need to test our framework in a controlled scenario. Thus, in this chapter, we present the design of a synthetic distributed system that can reproduce numerous complex anomalous behaviors. We then employ a series of high level design and deployment tools to build, execute, and monitor our synthetic system in a controlled manner. Using monitoring data obtained from our synthetic system, we then perform a comprehensive parametric study on our framework to better understand how the different configuration parameters impact on our framework's detection accuracy.

## 4.1 Overview

Distributed systems often manifest nonlinear or even chaotic behaviors in the sense that small variations in the system state at a particular instant can result in a completely different outcome. This is caused by the vast number of system interactions and their complexity, and by the non-deterministic nature of concurrent systems, where timing and the order of events play a key role in the system outcome. For example, an unexpected delay in a particular transaction can trigger a race condition that leads to a deadlock.

Errors that arise from the system interactions or the timing of events are caused by faults introduced during the development stage. However, system interactions and timing are conditions that are difficult to study before system execution, making these types of errors extremely difficult to predict at the development stage (see Chapter 1). These types of errors have also been referred to as *emergent behaviors* [20], that is, complex anomalous behaviors that emerge from the interactions of the system components, and that cannot be identified by just observing the individual components. Emergent behaviors in software systems have been grouped into six categories [20]:

- **Livelock:** This happens when the progress of the system stalls because several entities are changing in response to each other. A special case of this is *receive livelock*, when the progress of the system stalls because an increasing amount of high priority tasks needs to be processed before actual work can be performed.
- **Deadlock:** A situation in which the progress of the system stalls because several entities are waiting for each other.

- **Thrashing:** This occurs when the competition over a shared resource causes the system to spend more time switching contexts than doing actual work. A typical example is the one caused by too much paging in a system.
- **Unwanted synchronization:** When multiple system activities that should be independent from each other end up being correlated. For example, a series of messages that different components should transmit at random times end up being transmitted simultaneously due to some inherent coupling between the components.
- **Unwanted oscillation:** Occurs when the system oscillates between a fixed set of states.
- **Phase change:** This happens when an incremental change in some variable ends up causing a drastic change in the whole system. For example, in the presence of a memory leak, a system may keep correct functioning until a tipping point is reached and the whole system crashes.

Despite the harmful potential of these complex anomalous behaviors, very few works have addressed them [21, 35, 86]. A possible reason for this is the lack of publicly available datasets [23], or the difficulty in reproducing them in controlled and realistic conditions. The few publicly available datasets of system monitoring data [50, 87] do not contain cases of complex anomalous behaviors in a real world deployment.

In this chapter we build a synthetic distributed system able to reproduce several complex anomalies, and analyze these behaviors from the perspective of their impact on the system performance metrics. Then, through an extensive parametric study, we evaluate our framework's performance when dealing with these complex anomalous behaviors.

The rest of this chapter is organized as follows: Section 4.2 describes our synthetic distributed system and the different complex anomalous behaviors that it can reproduce. Section 4.3 analyzes how these anomalous behaviors affect the performance metrics of our synthetic distributed system. In Section 4.4 we present a comprehensive parametric study on our framework employing monitoring data from our synthetic distributed system, and Section 4.5 concludes the chapter.

## 4.2 Synthetic Distributed System

We developed our own synthetic distributed system (SDS) due to the lack of publicly available datasets containing complex anomalous behaviors, and to provide a controlled testing scenario for our framework. The SDS allows us to model several types of anomalous behaviors and to analyze our framework's detection capabilities. The development and deployment of the SDS is done by means of MEDEA [88], a tool for modeling and analyzing the performance of distributed systems. MEDEA allows to graphically model system components and their interactions, and then automates their deployment and execution by generating code using the Component workload emulator Utilization Test Suite (CUTS) [89]. During system execution, MEDEA also allows for the collection of several performance metrics, such as CPU and memory utilization, by means of a Data Distribution Service [90] logging system.

### 4.2.1 System Design

The SDS consists of a distributed system where multiple Clients share access to several Databases. Clients can freely read information from the Databases, but write access is granted using a distributed mutual exclusion protocol. This protocol is an adaptation of the algorithm by Ricart and Agrawala [91] that achieves mutual exclusion using $2(N-1)$ messages, being $N$ the number of processes. In the SDS, access to each

Database is controlled by a subset of Clients. Clients wanting to write to a Database must request votes to the Database controllers, and can only perform the write after achieving a majority of their votes.

Components in MEDEA are defined in terms of their interface and behavior. The Client's interface has two input ports and two output ports, namely, `commIn`, `txIn`, `commOut` and `txOut`. Communication ("comm") ports are used to interact with other clients, whereas transfer ("tx") ports are used to access Databases. The Database's interface has two ports: `txIn` and `txOut` to interact with Clients. Clients exchange five types of communication messages between them: `voteRequest` to ask for votes to obtain write access to a Database; `vote` to issue a vote; `rescind` to rescind an issued vote; `rescindAck` to acknowledge a rescission; and `release` to inform voters that the write action has been performed. The Client component also exchanges four types of transfer messages with the Database component: `read`, `write`, `readAck`, and `writeAck`. To ensure a correct ordering of actions, all exchanged messages are tagged on emission with a Lamport clock timestamp [92]. Fig. 4.1 shows the component interfaces, their ports, and the connections between them. Client B `txIn` and `txOut` connections are not depicted for clarity. In the case of having more components, each Client is connected to all the other Clients and to all the Databases.

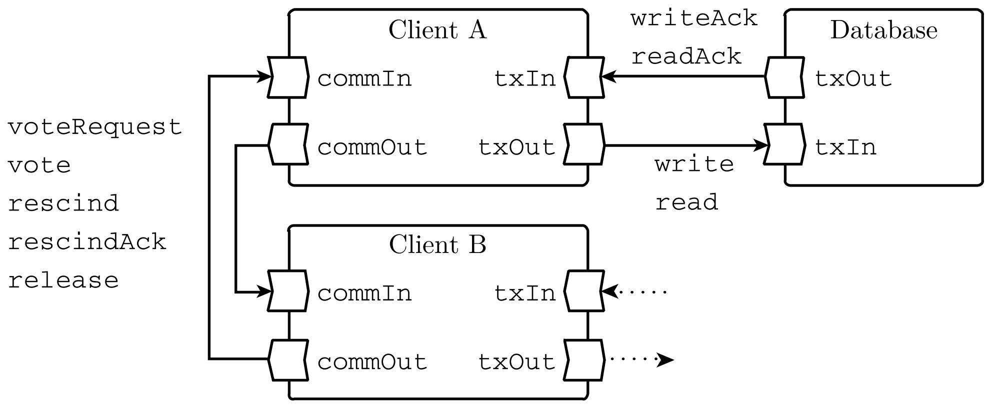


**Fig. 4.1.** Representation of the components interfaces, and how these are connected.

Clients can be in three different states: `working`, `writing`, and `rescinding`. The `working` state means that a Client is carrying out some activity, and it is used to prevent Clients from doing more than one thing at a time, otherwise the whole system could become overloaded with requests. The `writing` and `rescinding` states are reached when a Client is writing to a Database or rescinding a vote respectively. Clients can be in multiple states at the same time, or not be in any of them. Additionally, each Client maintains a Lamport clock, a priority queue to store voting requests, information about the current issued vote, a counter for its own votes, and the ID of the Database to manage. Database management is assigned uniformly among all Clients when the SDS is initialized. The Databases behavior is much simpler than the Client behavior as they just reply to write and read requests with acknowledgements.

Component behavior in MEDEA is modeled as a set of workflows that execute when certain events trigger. Fig. 4.2 shows how Clients decide which action to carry out. Clients continuously perform one of three actions: write and read to a Database, or idle for a random amount of time. Clients have a 0.5 probability of choosing the write action, a 0.35 probability of choosing the read action, and a 0.15 probability of choosing the idle action. The workflow in Fig. 4.2 begins with a periodic event. Upon triggering, the Client checks its `working` state. If the Client is not working, it enters the `working` state and randomly chooses an action to perform. If the Client is already working, it checks if it is being idle, and the amount of remaining idle time. If the chosen action is to write, the Client selects one of the available Databases at random, and broadcasts a `voteRequest` to the other Clients. A `voteRequest` message expresses the desire of the sender Client to obtain mutual exclusion to access a Database. In these requests, Clients include their ID, a timestamp, and the target Database. If the chosen action is to read, the Client directly accesses a random Database by sending a `read` message. As said before, reads do not require mutual exclusion. If the chosen action is to remain idle, the Client sets a random amount of time to do so. Apart from selecting an action, every time the periodic event triggers,

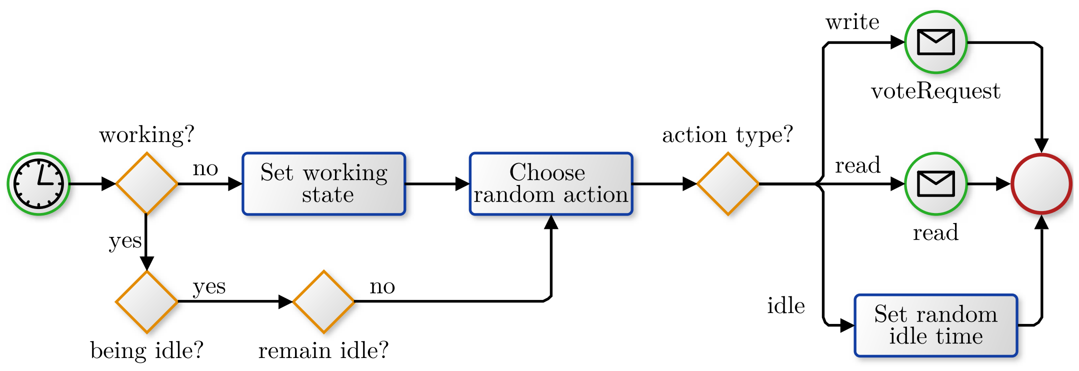


**Fig. 4.2.** Process that models how Clients periodically perform different actions.

Clients also perform a random number of iterations in an empty loop to simulate CPU work. In Fig. 4.2, missing arrows coming out of some of the evaluation nodes (e.g., remain idle? $\rightarrow$ yes) go to the ending state, and are not depicted for clarity.

A voting process runs concurrently in addition to the action selection process. This voting process, depicted in Fig. 4.3, is in charge of issuing and rescinding votes. The voting process also runs periodically, and begins by checking whether the Client has already issued a vote. If the Client has not issued a vote and the priority queue that stores voting requests is not empty, the Client removes and votes for the head of the queue. The priority queue sorts voting requests by timestamp and Client ID, ensuring that the oldest request is always at the head of the queue. If the Client has already issued a vote but it is not the oldest one in the queue, the Client enters the `rescinding` state and sends a `rescind` message to the Client that was voted. It can be seen how the `rescinding` state is used to avoid carrying out more than one rescission concurrently.

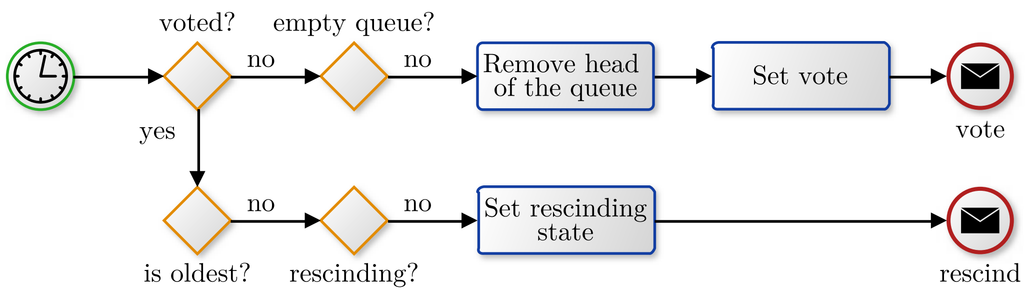


**Fig. 4.3.** Process that models how Clients issue and rescind votes.

A third periodic process manages writes to the Database. This process is shown in Fig. 4.4. If the Client is not already writing and it has the majority of the votes, it sends a `write` message to the Database and enters the `writing` state. To represent the information being written, `write` messages carry a random number of bytes.

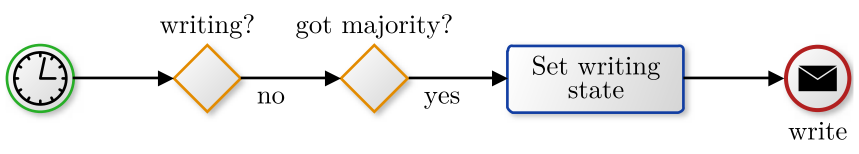


**Fig. 4.4.** Process that models how Clients write to a Database after achieving the majority of the votes.

Client interactions through the communication ports are modeled as a separate process that can be seen in Fig. 4.5. Upon message arrival at the `commIn` port, the Client checks the message type and carries out the appropriate actions, which are summarized in the following:

- `voteRequest:` The Client first checks if the request is to access the Database it manages. If that is the case, the Client simply inserts the request into its priority queue. The voting process will serve the request when appropriate.
- `vote:` The Client increases its number of votes if the vote is valid. Valid votes are those corresponding to the Client's current voting request, which is identified by its timestamp. If the vote is outdated, the Client replies with a `release`, as the writing action has already been completed.
- `release:` If the message is valid (i.e., corresponds to the current issued vote), the Client sets its issued vote to nil and resets the `rescinding` state, as a rescind is no longer possible. This allows the voting process to serve another request from the queue.
- `rescind:` If the message is valid and the Client is not in the `writing` state, the Client decreases its number of votes and replies with a `rescindAck` to

acknowledge the rescission. If an outdated `rescind` is received, the Client simply replies with a `release`. Note that if the Client is already writing, the `rescind` message is ignored as a `release` will be sent after writing is finished.

- `rescindAck`: The Client inserts the vote request again in the priority queue (remember that it was removed when the Client issued its vote), sets its vote to nil and exits the `rescind` state. At this point the rescission has been successful, and the voting process will vote for the oldest Client in the queue.

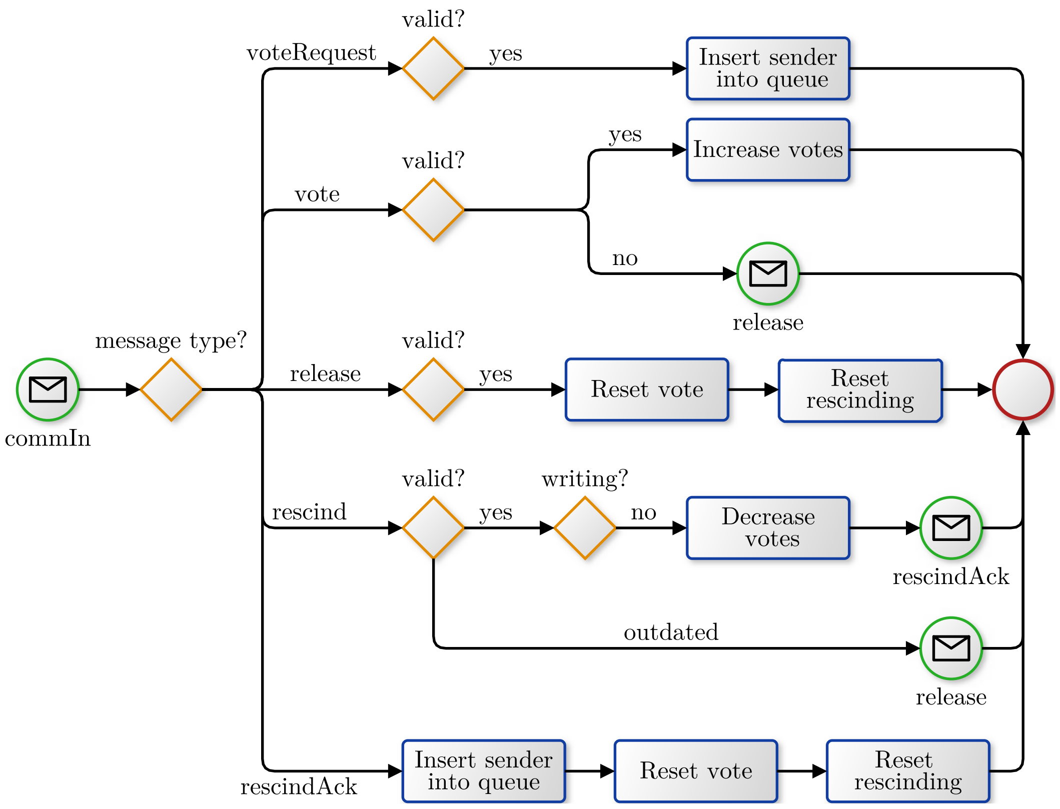


**Fig. 4.5.** Process that models how Clients treat messages received at the `commIn` port.

The behavior of the Database component is only composed of one process, shown in Fig. 4.6. The Database carries out a random amount of iterations in an empty loop every time it receives a message to simulate CPU work, and then replies with the corresponding acknowledgement. The `readAck` message contains a random number of bytes that simulate the retrieval of some information from the Database.

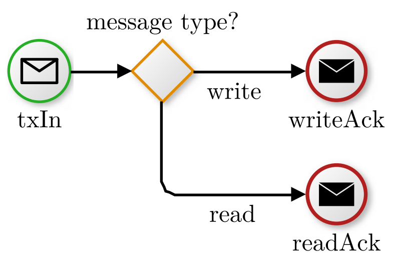


**Fig. 4.6.** Process modeling the Database behavior.

Finally, the process that manages Clients' `txIn` port is depicted in Fig. 4.7. When a `writeAck` message is received the Client broadcasts a `release` message allowing new votes to be issued to another Client, sets its number of votes to zero, and resets the `writing` and `working` states to end the writing action. If a `readAck` is received, then the Client resets the `working` state to end the read action.

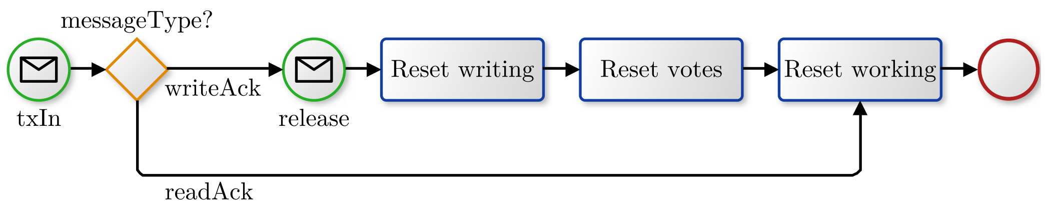


**Fig. 4.7.** Process that models how Clients treat messages received at the `txIn` port.

All processes run concurrently within each system component, and all communication is asynchronous. However, MEDEA guarantees that messages received at the same port are processed one at a time. Additionally, we force some actions to be performed in a mutually exclusive manner. For example, in the process in Fig. 4.7, the sending of the `release` message, and the state updates are performed atomically to avoid race conditions such as processing a rescission in between. Note how the liveness of the SDS is guaranteed since, eventually, the oldest vote request will achieve a majority of the votes, and outdated `vote` and `rescind` messages are always replied with a `release` message that frees the vote issuer.

### 4.2.2 Modeling Anomalous Behaviors

The behavior of the SDS is complex enough to allow the introduction of several anomalous behaviors. The SDS behavior follows the global behavior scheme described in Chapter 3, that is, a distributed system with a well-defined global behavior. More precisely, we model five types of anomalies: deadlock, livelock, unwanted synchronization, memory leak, and starvation. We introduce a new Client state per each of the anomalies, and modify certain processes to generate the desired behavior. Anomalies can then be activated and deactivated by setting Clients in the desired state. The way in which the SDS reproduces the anomalous behaviors is described in the following.

**Deadlock** This anomaly is achieved by forcing the priority queue to return a random vote request instead of the oldest one, and not allowing rescissions. This ensures that votes are distributed among all requesting Clients, preventing any of them to reach the majority. Deadlock is implemented to only affect access to one of the Databases. This means that, since Clients perform actions at random, the SDS could keep progressing under the right conditions. However, as soon as two Clients try to access the affected Database at the same time, the probability of deadlock occurring is high. Moreover, once the affected Database enters a deadlocked state, the whole system rapidly shuts down as Clients progressively try to access it.

**Livelock** This anomaly is achieved by forcing the priority queue to return a random vote request instead of the oldest one, and letting Clients to rescind their vote. This creates an endless loop of rescissions since Clients are unable to vote for the correct requester. Like deadlock, livelock only affects one of the Databases, and due to the randomness of the voting process, Clients are able to obtain write access in rare occasions. Even so, like in deadlock, the whole system rapidly degrades when all Clients end up trying to access the livelocked Database.

**Unwanted synchronization** This anomaly is recreated by forcing all Clients to request write access to the same Databases, one after the other. In addition to this, a limit is added to the number of requests that can be stored. Clients start by requesting write access to one of the Databases. This causes manager Clients for this Database to receive a number of vote requests. When the limit of requests is reached, manager Clients simulate an overloading by sending a new type of message to vote requesters, the `reject` message. When a requester Client receives a `reject` message, it broadcasts a `release` message, and requests access to another Database, which is the same for all Clients. This produces periodic bursts of requests from one Database to another as long as the synchronization is active. Contrary to deadlock and livelock, this anomaly affects the whole system since the beginning.

**Starvation** This anomalous behavior is achieved by forcing Clients to ignore vote requests from one of the Clients. In this manner, the affected Client is unable to achieve the majority and starves.

**Memory leak** This anomaly is created by making one of the Clients to periodically allocate an increasing amount of memory until a maximum value close to the total available system memory. This anomaly does not fit into the definition of complex anomalous behavior. Nevertheless, we include it due to its prevalence in many software systems.

### 4.2.3 Testbed

We run the SDS in a testbed consisting of six nodes, each composed of two Intel Xeon Dual Core 2.33GHz processors, 4GB of memory, and 73GB of disk space. The nodes run CentOS 6.5 as operating system, and the system deployment and execution is driven by Jenkins [93]. One of the nodes runs the logging system and the remaining five run the system components. In our experiments, we deploy 20 of these components:

17 Clients and 3 Databases. Each Database is deployed in a different node and Clients are uniformly distributed, resulting in each node running four components.

As said before, MEDEA allows the collection of several performance metrics per node. These are: total CPU utilization, utilization of each core (four per node), disk utilization per each partition, and memory utilization. In addition to these metrics, the SDS logs other kinds of information, such as messages sent and received by every component. Using this data, we extract network utilization (in and out) per node, and the number of messages sent per Client. Network utilization is essential to fully model the behavior of the SDS as it implements a distributed mutual exclusion protocol, and thus relies heavily on component interactions. We also include messages sent per Client to be able to capture the starvation anomaly. In the same manner, other application metrics, such as the number of Clients contacted or the number of writes performed, can be extracted. In total we monitor 60 hardware metrics (including network utilization), and 17 application metrics. The monitoring frequency is once per second for all metrics.

## 4.3 Behavior Characterization

As detailed in Chapter 3, in order to detect anomalies, our framework builds a behavioral model from the monitored performance metrics. Understanding how system metrics change in response to anomalies is useful to improve the construction of this model. Fig. 4.8 shows two hundred seconds of the hardware metrics of one of the nodes when the SDS is in normal operation. The behavior of a node is the combination of the behaviors of all the components running in that node, which is four per node in our case. This makes the behavior of the nodes to be very similar to each other. It can be seen that CPU and network utilization are the metrics that provide more information. However, it is important to include as many metrics as possible in the analysis since anomalous behaviors are unpredictable, and can have an impact on

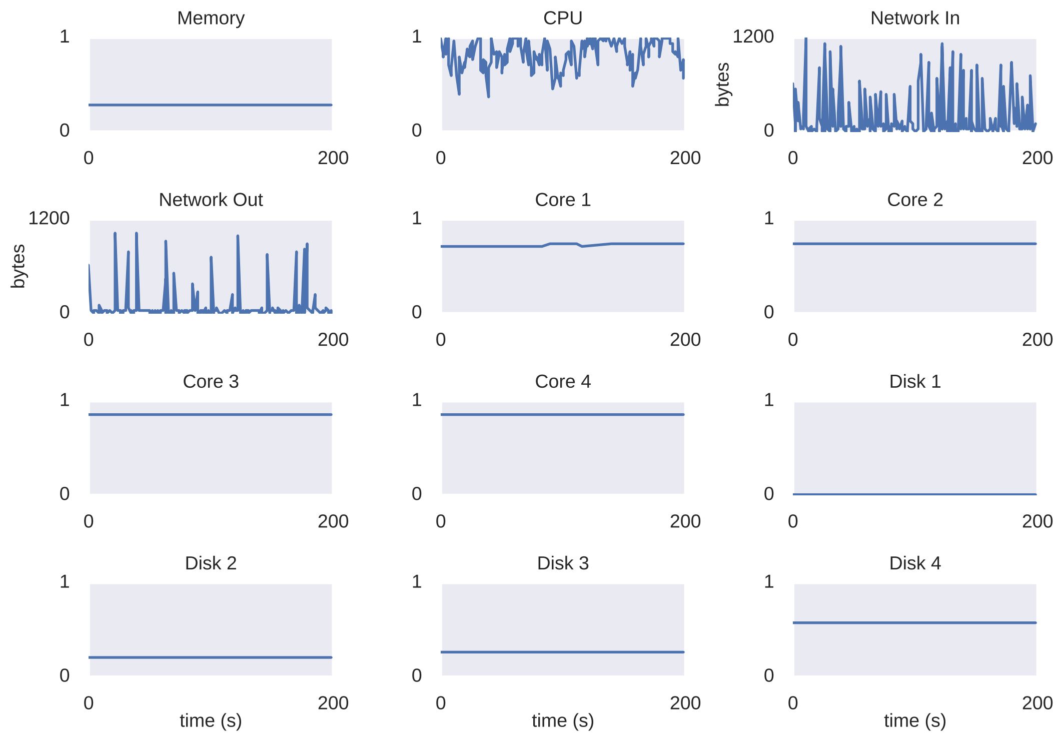


**Fig. 4.8.** Hardware metrics collected from one of the nodes in a time span of two hundred seconds with the SDS running in normal mode.

any of the metrics. For example, even though memory utilization does not seem very informative, it becomes crucial in the presence of a memory leak.

Fig. 4.9 shows two hundred seconds of memory, CPU and network utilization of one of the nodes right after the SDS deadlocks. At the beginning the system seems to behave normally, however, around one hundred seconds after deadlock is activated, both network input and output suffer a complete drop in activity. This is because Clients enter deadlock progressively when they try to write to the affected Database, as explained in Section 4.2.2.

Fig. 4.10 depicts the same metrics immediately after a livelock is activated. Similar to deadlock, we see a significant drop in network utilization after around 100 seconds. Network input seems to remain active for a longer period, possibly due to Clients still writing or reading to a Database not affected by the anomaly. In this case, Clients keep exchanging votes and rescissions between them, and some network activity remains.

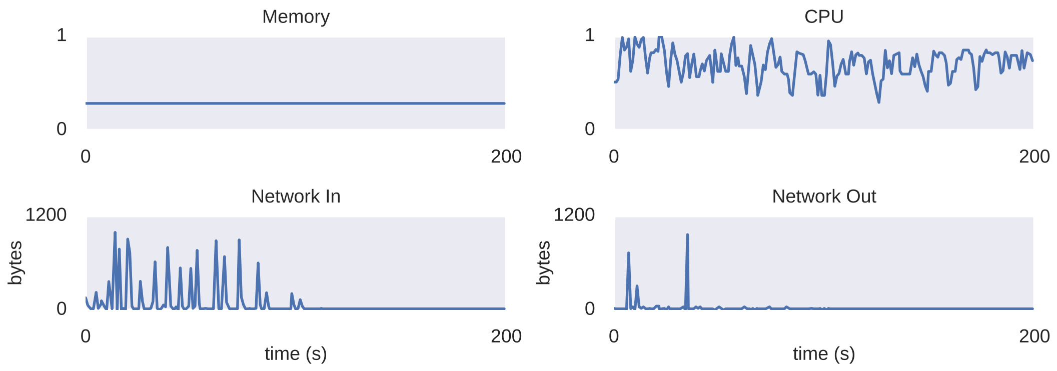


**Fig. 4.9.** Hardware metrics collected from one of the nodes in a time span of two hundred seconds with the SDS running in deadlock mode.

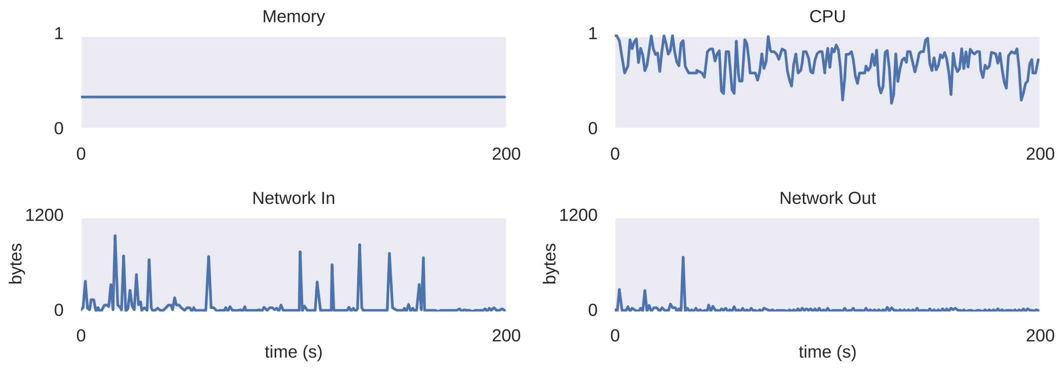


**Fig. 4.10.** Hardware metrics collected from one of the nodes in a time span of two hundred seconds with the SDS running in livelock mode.

Fig. 4.11 shows the hardware metrics of a node during unwanted synchronization. Again, the most affected metrics are network input and output. In this case, network input is characterized by short periods of high activity followed by low activity regions. This is because the depicted node is running one of the Databases, and Clients have their write requests synchronized.

Fig. 4.12 shows network input of the five nodes running the SDS when unwanted synchronization is taking place. The nodes running the three Databases are one, four, and five. This clearly shows how Clients move from one Database to another in groups, as high activity a nodes matches low activity in the rest. Nodes not running Databases do not receive as much traffic as the others because Clients are continuously doing writes.

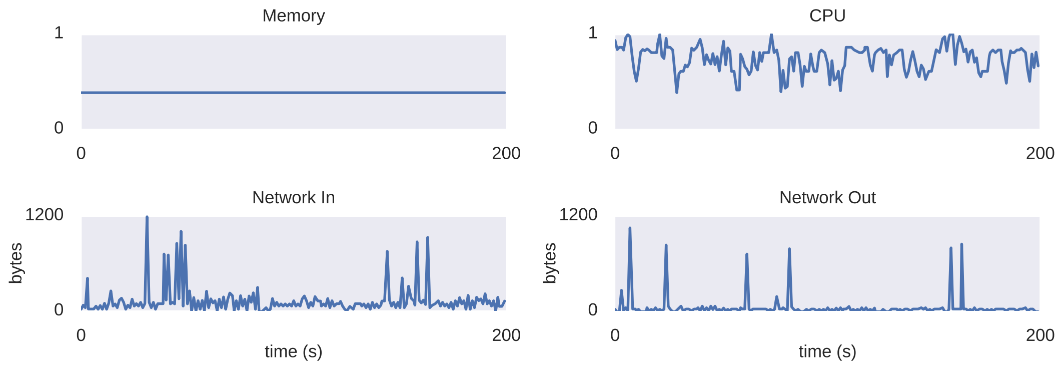


**Fig. 4.11.** Hardware metrics collected from one of the nodes in a time span of two hundred seconds with the SDS running in synchronization mode.

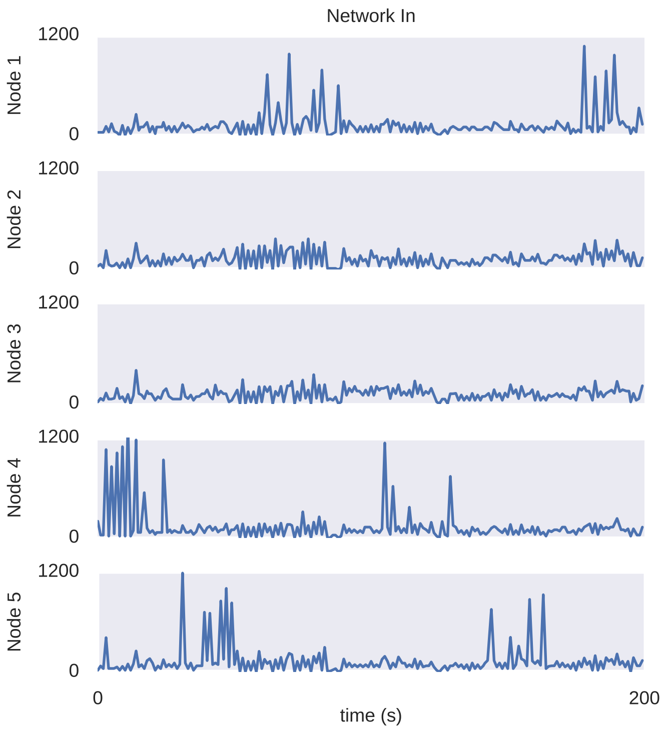


**Fig. 4.12.** Network input of the five nodes running the SDS during an unwanted synchronization.

Fig. 4.13 shows the metrics of a node affected by a memory leak. This anomaly has a significant impact on memory utilization, where an increasing trend is generated. The rest of the metrics remain normal.

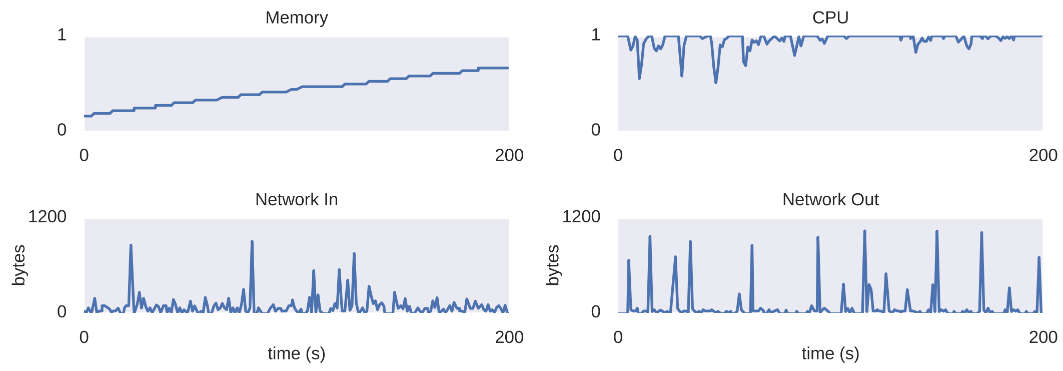


**Fig. 4.13.** Hardware metrics collected from one of the nodes in a time span of two hundred seconds during a memory leak.

Starvation does not have a significant impact on the hardware metrics because there are four components executing concurrently in each node. This is the reason we also collect the number of messages sent per Client.

Fig. 4.14 shows this metric for four Clients running in the same node right after the system activates the starvation of Client 3.

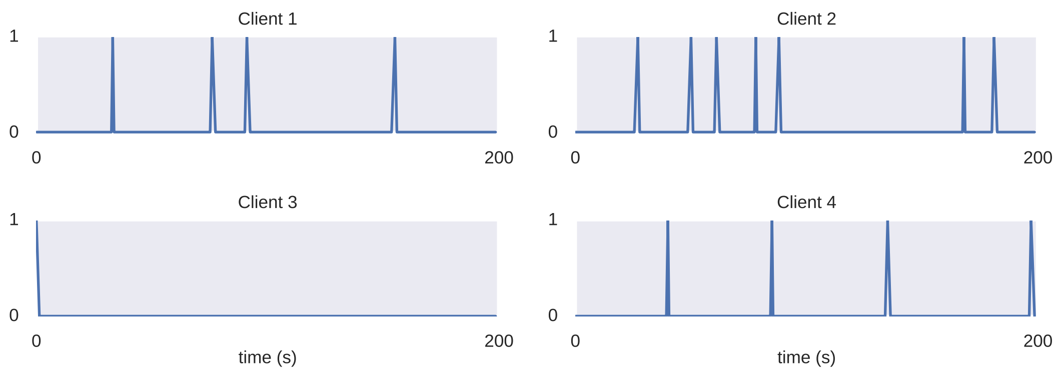


**Fig. 4.14.** Messages sent per Client right after the SDS enters starvation mode.

Observing how different anomalies impact on the system metrics provides insight on what kind of information a behavior representation needs to include in order to maximize the chances of detecting anomalies. The observed patterns support the ideas outlined in Chapter 3 in regards to representing system behavior. On the one hand, it is clear that most anomalies are characterized by an anomalous sequence of values. For example, deadlock (Fig. 4.9) is characterized by a downward shift in network output (among other conditions). This shift differs from the fluctuations in network output observed during normal operation (Fig. 4.8) in that the metric remains low for a certain amount of time. Observing network output at a single time instant does not provide sufficient information to differentiate deadlock from normal behavior, as network output can occasionally take low values during normal operation as well. Instead, we require to analyze the system in a time frame to decide whether the metric has been low for enough time to be considered an anomaly. On the other hand, some anomalies are characterized by anomalous correlations between several metrics. For example, during unwanted synchronization there is a significant coupling between the network input in several nodes. Thus, in order to detect these anomalies, it is necessary to analyze the metrics as a whole instead of independently.

In addition, the way in which the system metrics deviate from normality is similar across different types of anomalies. This is because there is a finite number of patterns that a time series can produce, and these have been extensively studied [94]. For example, deadlock and starvation produce a similar drop in the metric that each anomaly affects. A better understanding of these deviations is crucial for building a behavioral model that enables the detection of previously unseen anomalies, as these are likely to reproduce the same patterns in the data. These common patterns are [94]:

- **Shifts:** an abrupt change in the probability distribution of a metric. That is, an upward or downward shift in mean and standard deviation. Shifts can be indicative of sudden changes in the system internal operation such as deadlocks, errors in configuration, malicious activities, or software and hardware failures.
- **Trends:** or more precisely, changes in trend. Since resources in a computer system are finite, upward or downward trends cannot reproduce indefinitely, and must change at some point. Changes in trend can signify resource exhaustion or system degradation.

- **Seasonality:** patterns in the data that repeat at uniform time intervals. The appearance of a seasonal pattern may imply the presence of an unwanted synchronization or oscillation, whereas changes in seasonality can mean a variation in the system's normal behavior, for example, due to network congestion or a malicious activity.

Shifts, trends and seasonality changes can be produced by anomalies, but can also be the result of a normal change in the system. For example, upward shifts or trends can appear due to an increase in user activity. In these cases, it is necessary to analyze the correlation between multiple metrics.

## 4.4 Parametric Study

The framework presented in Chapter 3 can be configured with various parameters to tune the detection accuracy in different scenarios. The *Behavior Extractor* uses a window size (denoted $z$) to control the amount of consecutive readings that form the feature vectors. The SVM classifiers can be tuned with various variables (or hyperparameters): $\gamma$ to control the kernel function, $\rho$ to define the area of the decision function, $\eta$ to define the learning rate, $\lambda$ to give more importance to new vectors, and $\nu$ to control the frequency of anomalous classifications. The *Feedback Provider* uses a parameter called $Tr$ to force a training phase at the beginning of the execution, and another parameter called $Gr$ to define the amount of consecutive anomalous predictions before raising an alert. Table 4.1 summarizes the different parameters and their ranges. In this section, we present a parametric study to obtain a better understanding on how the different configuration parameters affect our framework's performance.

The most important parameters are the window size ($z$), the kernel tunning hyperparameter ($\gamma$), and the hyperparameter that controls the frequency of alerts ($\nu$). These three parameters directly affect the behavioral representation of the system

**Table 4.1.** Summary of the different parameters used to tune our framework.

| Component | Parameter | Purpose | Typical Range |
|---|---|---|---|
| *Behavior Extractor* | $z$ | Window size | $\geq 1$ |
| *Behavior Identifier* | $\gamma$ | Kernel tunning | $> 0$ |
| | $\nu$ | Frequency of alerts | $(0,1]$ |
| | $\eta$ | Learning rate | $(0,1]$ |
| | $\rho$ | Decision function size | $(0,1]$ |
| | $\lambda$ | Decay rate | $[0,1]$ |
| *Feedback Provider* | $Tr$ | Training size | $\geq 0$ |
| | $Gr$ | Alert grouping | $\geq 1$ |

and the decision function that separates the normal and anomalous classes. The $\rho$ parameter is also important, however, since this parameter is adjusted at run time (see Chapter 3), it has a lower impact on the final detection accuracy. The other classifier hyperparameters ($\eta$ and $\lambda$) control the speed at which new information is incorporated into the *Behavioral Model*, and the rate at which old data is "forgotten". These hyperparameters can be highly useful in particular scenarios, but their impact on the detection accuracy is in general lower than other hyperparameters.

$Tr$ and $Gr$ are useful to reduce the number of false positives. $Tr$ allows to incorporate normal data to the model at the beginning of the execution, reducing the number of false positives that can naturally appear when the BM does not contain any information, and $Gr$ can help to reduce the amount of false alerts in scenarios with noisy data or where the normal and anomalous classes are very similar.

We generate a set of monitoring data by running the SDS with different settings. We run the SDS for 50 minutes, and inject one of the anomalous behaviors described in Section 4.2.2 at minute 20. The anomalous behavior is active for a certain amount of time, and then the SDS resumes normal operation. We experiment with anomalous behaviors of 20, 50, 100, 300, and 600 seconds long, and run the SDS 10 times per anomaly type and duration, which means a total of 500 executions for the study. In the rest of this chapter we will refer to this dataset as SDS-Dataset-1.

### 4.4.1 Framework Deployment

As detailed in Chapter 3, our framework can be deployed in various manners to adapt to different system requirements. Since the different complex anomalies that the SDS can reproduce are system-wide behaviors, we employ a global perspective for the study of the SDS behavior. The deployment setting that our framework employs to model global system behavior is depicted in Fig. 4.15.

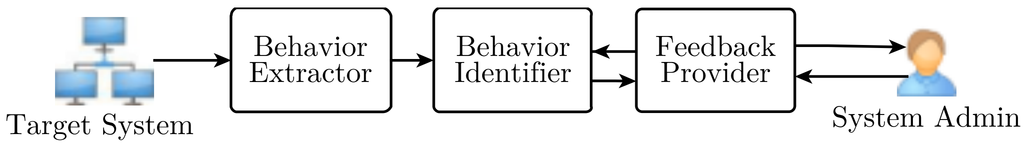


**Fig. 4.15.** Setting used in global distributed system behavioral analysis.

### 4.4.2 Experimental Setup

In our experimental analysis we take certain considerations that are worth mentioning. These are detailed in the following.

**Hyperparameters** Our framework's performance highly depends on the parameters used in the classification process (i.e., $\gamma$ , $\nu$, $\rho$, $\eta$, and $\lambda$). Choosing the right parameters in a classifier is a well-known limitation in supervised learning [95] that typically involves performing a grid search using the training set [84]. This procedure cannot be applied in our case, as our framework operates in a completely online manner, and it does not rely on historical data. Instead, we reduce the size of the search space by setting $\rho$, $\eta$, and $\lambda$ to predefined values. In doing this we might be using suboptimal values and the results obtained become a lower bound on our framework's performance. In the case of the one-class classifier, we set $\rho = 0.5$ and $\eta = \frac{1}{\sqrt{t}}$, where

$t$ is the number of classified vectors. In the case of the binary classifier we use the same function for $\eta$, and set $\rho = 0$ and $\lambda = 1$. Regarding $\gamma$ and $\nu$, we experiment with several values and present the best results obtained.

**Evaluation Time** We configure our framework to ignore the first $z_{max} - z + (Tr_{max} - Tr)$ seconds of execution, where $z_{max}$ and $Tr_{max}$ are the maximum window size and $Tr$ values used across all the parametric study. This is done to be able to compare the results when using different parameter combinations, since using a higher $Tr$ or $z$ results in less vectors being classified, and therefore, less probability of getting false alarms. In this manner, the amount of execution time used to evaluate our framework is the same in all experiments regardless of the parameters used.

**Feedback** Feedback is given automatically every time that alerts are raised, and as long as the *Feedback Provider* remains in the *Alert* state (see Chapter 3).

**Performance Metrics** Network utilization exhibits a lot of variance, and can mask anomalies that have an impact on other metrics, such as memory leak and starvation. In the experiments, we remove network utilization from the analysis when detecting these two anomalies. This permits an unbiased comparison between the performance of our framework when dealing with different anomalous patterns in the system metrics.

**Evaluation Metrics** Our framework's objective is to alert the system administrator as soon as possible after anomalies appear, but also to produce the minimum possible number of false alarms, since inspecting them is very time consuming. The typical evaluation metrics employed in anomaly detection [39], which are true positive rate (TPR) and false positive rate (FPR) with respect to the vector classifications are not very informative in our case. For example, in a run of 50 minutes long from which 20 seconds are anomalous and 2980 seconds are normal, generating 10 false positives yields a FPR of $\frac{2970}{2980} \approx 0.003$, which could be interpreted as a good result. However, investigating 10 false alarms in 50 minutes is unacceptable in a real world scenario. For this reason we employ TPR (or *Recall*) and *Precision*, and define them with respect to the number of anomalies instead of the classified vectors, that is,

$$\begin{aligned} Recall &= \frac{\text{detected anomalies}}{\text{total number of anomalies}} \\ Precision &= \frac{\text{detected anomalies}}{\text{total number of alerts}}, \end{aligned} \tag{4.1}$$

where we consider an anomaly as detected if at least one alert is raised while the anomaly is active. *Recall* evaluates the ability of our framework to detect anomalies, whereas *Precision* evaluates the absence of false positives. Both metrics range from 0 to 1, being 1 the optimal value. Note that multiple alerts during the same anomaly affect *Precision* negatively. In this manner we penalize random alerts and ensure that the anomaly is being truly detected. Additionally, in some experiments we also provide the detection time (DT), which is the time from the appearance of an anomaly until it is detected, and the F-score ($F_1$), which is the harmonic mean of *Recall* and *Precision*:

$$F_1 = 2 \cdot \frac{Recall \cdot Precision}{Recall + Precision}. \tag{4.2}$$

## 4.4.3 Features

We employ four features to represent behavior in the parametric study: mean (*ME*), standard deviation (*SD*), skewness (*SK*), and kurtosis (*KU*). Given a sequence of readings $\{r_i | 1 \leq i \leq z\}$ where $z$ is the sliding window size, the features are defined as [79]

$$ME = \frac{\sum_{i=1}^{z} r_i}{z} \tag{4.3}$$

$$SD = \sqrt{\frac{\sum_{i=1}^{z} (r_i - ME)^2}{z}} \tag{4.4}$$

$$SK = \frac{\sum_{i=1}^{z}(r_i - ME)^3}{z \cdot SD^3} \tag{4.5}$$

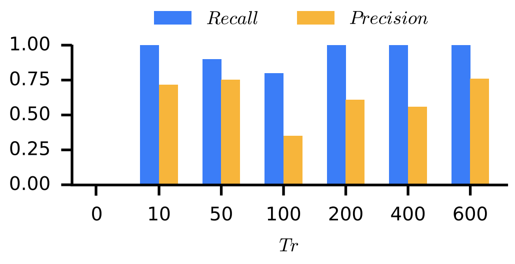
$$KU = \frac{\sum_{i=1}^{z}(r_i - ME)^4}{z \cdot SD^4} - 3\,. \tag{4.6}$$

These features provide a good representation of the probability distribution of the values in a time series. Mean gives the peak of the distribution, standard deviation the spread of values, and skewness and kurtosis measure the asymmetry and tail of the distribution respectively.

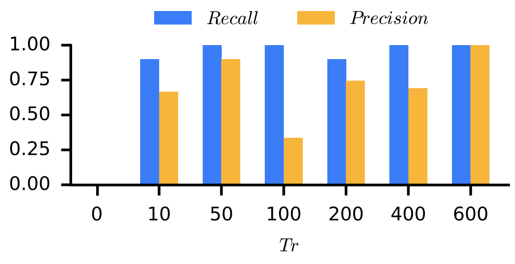
### 4.4.4 Initial Training

We want to know how beneficial is including a short training phase at the beginning of the execution. In some scenarios we can assume that a system runs normally for a short period of time at the beginning or, alternatively, we can closely monitor the system to ensure that no anomalies occur for a time period. Our framework can be set in train mode to avoid getting the false alarms that naturally occur at the beginning of the execution due to the behavioral model being empty. In these experiments, we use the 50 minutes runs containing an anomalous behavior between minutes 20 and 30, and employ different $Tr$ values. The window size is set to 128 seconds, and $Gr$ is set to 1.

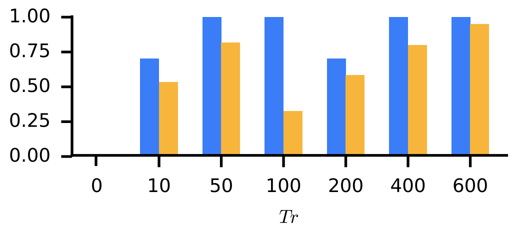
Fig. 4.16 shows mean *Recall* and *Precision* over 10 runs per anomaly type using different $Tr$ values. When $Tr$ is 0 none of the anomalies is detected. This is because we set $\rho$ to an arbitrary value, and the one-class classifier does not receive enough vectors to converge to a meaningful decision function. Even so, with $Tr = 10$, the one-class classifier has enough vectors to build the decision function, and is able to detect deadlock, livelock, and unwanted synchronization. For these three anomalies, results are improved slightly as $Tr$ increases, reaching 1.00 *Recall* with *Precision* of 0.76, 1.00 and 0.95 respectively at $Tr = 600$. In the case of the memory leak and starvation anomalies, *Recall* remains high in several cases, however, low *Precision* indicates that the anomalies are detected just because a high number of alerts increases the probability of one of them being raised during the anomalous part of the execution. In other words, the detection is accidental. In the next sections we will analyze the reasons behind the poor results obtained with these two anomalies.



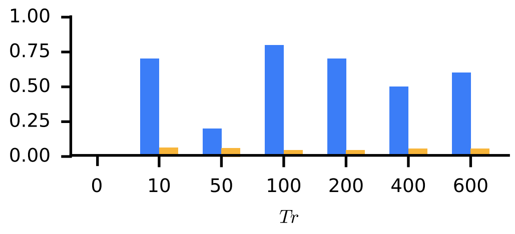
**(a)** Deadlock



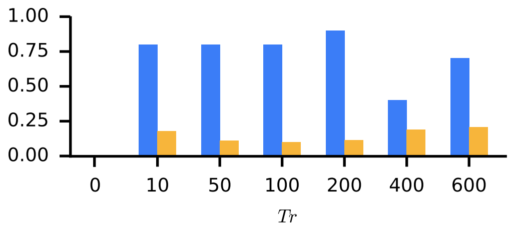
**(b)** Livelock



**(c)** Synchronization



**(d)** Memory Leak



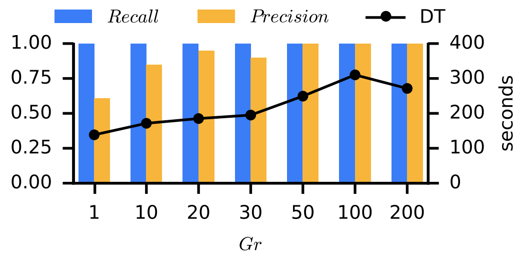
**(e)** Starvation

**Fig. 4.16.** Mean *Recall* and *Precision* obtained using different $Tr$ values. Anomaly duration is 10 minutes, $z = 128$, and $Gr = 1$.

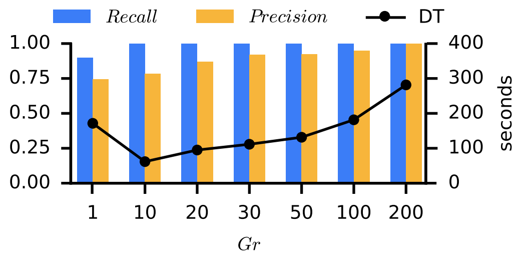
It is clear that the more information provided in the training phase, the better the results obtained. If the initial training period is shorter, the classifier obtains new information through false alarms, thus reducing *Precision*. However, as the behavioral model converges to an optimal representation of the system behavior, false alarms become less common, and our framework is able to achieve a good detection accuracy.

### 4.4.5 Alert Grouping

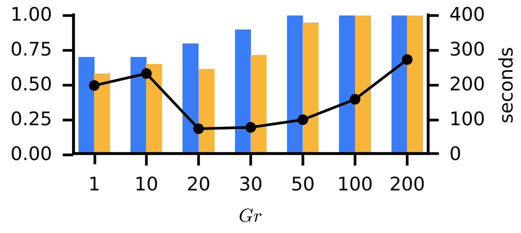
Through a second set of experiments, we want to know if false positives can be reduced by only raising alerts after a certain number of consecutive anomalous predictions have been found, and how this alert grouping affects our framework's detection time. We use a window of 128 seconds long, $Tr = 200$, and experiment with several $Gr$ values.



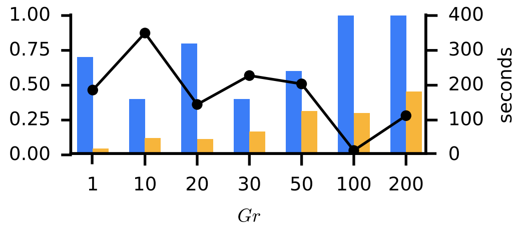
**(a)** Deadlock



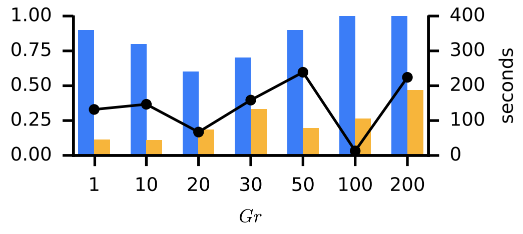
**(b)** Livelock



**(c)** Synchronization



**(d)** Memory Leak



**(e)** Starvation

**Fig. 4.17.** Mean *Recall*, *Precision* and DT obtained using different $Gr$ values. Anomaly duration is 10 minutes, $z = 128$, and $Tr = 200$.

Fig. 4.17 shows mean *Recall*, *Precision*, and detection time (DT) obtained in the monitoring data with 10 minute anomalies. Increasing $Gr$ clearly decreases the number of false alarms, resulting in higher *Precision*. In the synchronization case, higher $Gr$ values also increase *Recall*. This could be due to less noisy data being incorporated into the behavioral model, making the detection of the anomalous behavior easier.

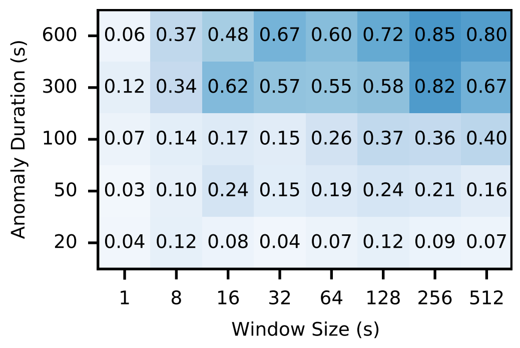
From $Gr = 1$ to $Gr = 200$, *Precision* improves by 64% in the deadlock case; by 34% in the livelock case; and by 71% in the synchronization case. Even though there is an improvement in memory leak and starvation as well, it is not significant as these anomalies are not being correctly detected. The drawback of high $Gr$ values is an increase in detection time that goes from less than 100 seconds with $Gr = 1$ to more than 300 seconds in some cases with $Gr = 200$. Considering these results, the optimal $Gr$ value is likely to be between 10 and 50 in most cases. However, higher values could be used in very noisy scenarios or when detection time is not critical.

### 4.4.6 Window Size

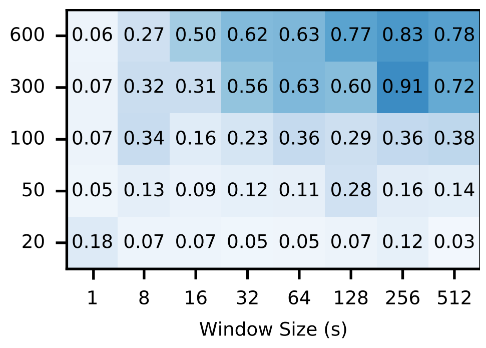
In a final set of experiments we analyze how window size affects our framework's performance. We set $Tr = 200$ and $Gr = 1$, and experiment with different window sizes and anomaly durations. Fig. 4.18 and Fig. 4.19 present mean F-score and DT respectively obtained with different combinations of window size and anomaly durations.

Fig. 4.18 shows that longer anomalies are detected more easily. In the case of deadlock and livelock (Fig. 4.18a and Fig. 4.18b), this is because it takes around 100 seconds for the anomalies to have a noticeable impact on the system metrics (see Section 4.3). Deadlocks and livelocks of 100 seconds or less do not differ enough from normal behavior when looking at the metrics. Something similar happens in the case of unwanted synchronization (Fig. 4.18c). It takes more than 50 seconds to be able to completely observe the anomalous pattern in the system metrics. This prevents our classifier to correctly detect the anomalies as these cannot be differentiated from normal behavior.

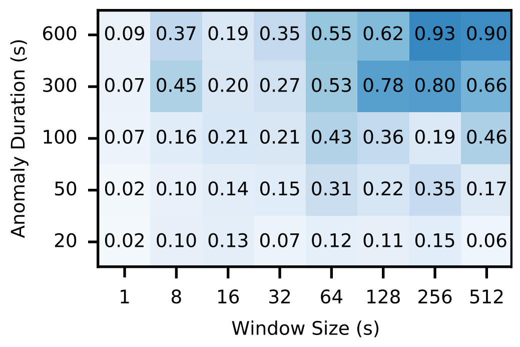
Longer windows achieve better detection rates overall. This is because of the combination of the sliding window and the feature extraction processes. Vectors generated in this way are equivalent to computing a rolling function that acts as



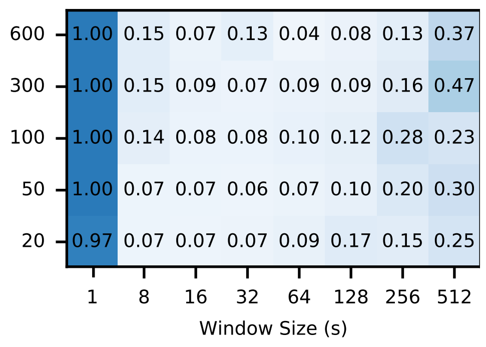
**(a)** Deadlock



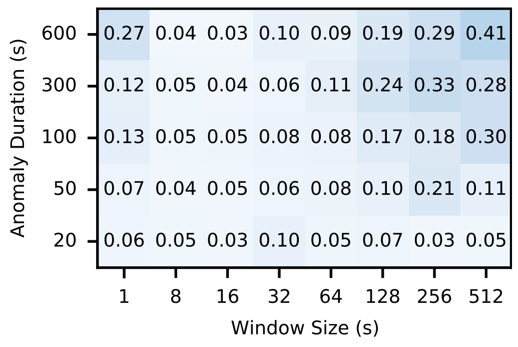
**(b)** Livelock



**(c)** Synchronization



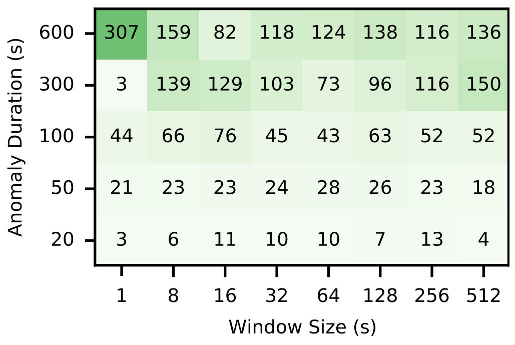
**(d)** Memory Leak



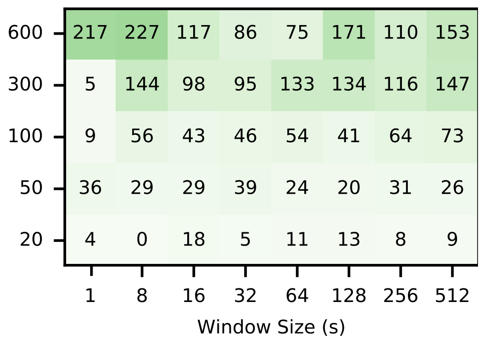
**(e)** Starvation

**Fig. 4.18.** Mean F-score for different combinations of window size and anomaly duration ($Tr = 200$ and $Gr = 1$).

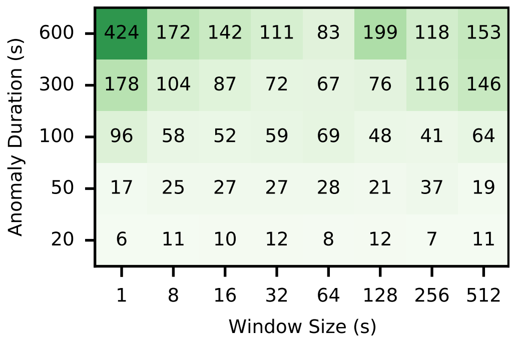
a noise filter. The longer the window, the smoother the resulting function. This is relevant when dealing with system metrics with high variance such as network utilization, as local changes are filtered out while preserving the global trends of the data. Moreover, longer windows do not prevent our framework from detecting short anomalies. For example, in the 100 seconds deadlock (Fig. 4.18a), a window of 256 seconds obtains better results than windows of 64 seconds or less, even though using a 256 second window means that fewer than 50% of the feature vector contains anomalous readings at best. This suggests that filtering local changes with larger windows is more beneficial than a higher proportion of anomalous readings in the feature vector. In addition, Fig. 4.19 reveals that detection time is not highly affected by the window size. Therefore, the optimal window size is between 128 and 512 seconds.



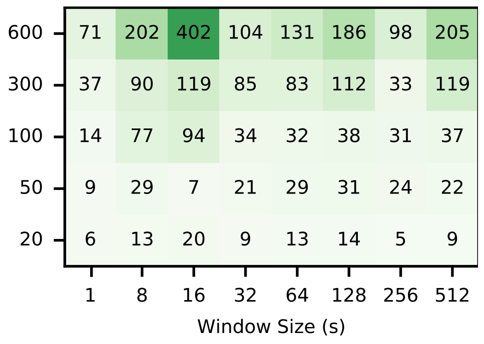
**(a)** Deadlock



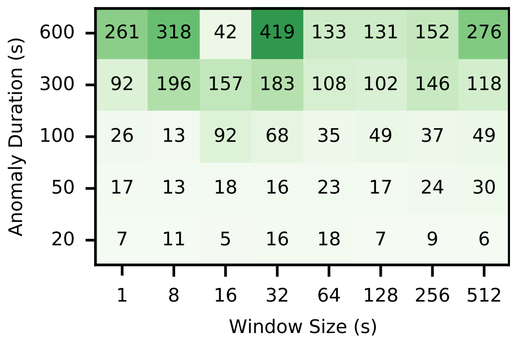
**(b)** Livelock



**(c)** Synchronization



**(d)** Memory Leak



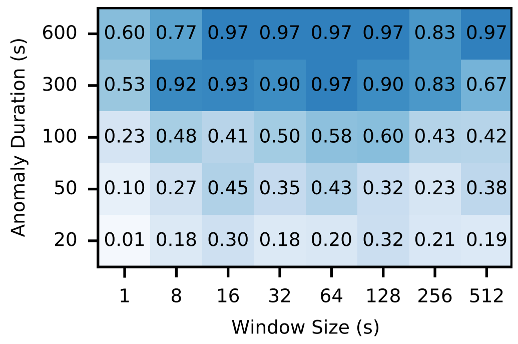
**(e)** Starvation

**Fig. 4.19.** Mean detection time in seconds for different combinations of window size and anomaly duration ($Tr = 200$ and $Gr = 1$).

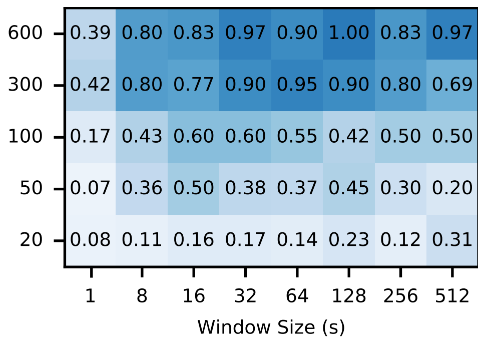
Windows of 1 second long perform poorly in most cases. This is consistent with the idea that analyzing the system over a time period is essential to detect most anomalous behaviors. The exception to this is the memory leak case (Fig. 4.18d), where using a window of size 1 seems to work much better than larger windows. This is because memory leak generates a smooth trend in memory utilization, and a window of size 1 is equivalent to obtaining the utilized memory at a time instant. Since in the memory leak case we remove network utilization from the analysis, our framework is able to detect the memory leak by setting a simple threshold. This, however, would not be useful in a real scenario where we are interested in detecting multiple types of anomalies, and where memory usage can increase due to other reasons (e.g., an increase in user requests).

The results obtained in the memory leak case suggest that the anomaly is not being detected when using windows of more than 1 second long because of how the behavior is represented. Probably, a different combination of features would yield much better results. The reason behind the bad results obtained in the starvation case (Fig. 4.18e) can be an improper choice of features, or the characteristics of the application metrics. It is possible that the difference between normal and anomalous behaviors when looking at messages sent per Client is not significant enough to trigger the anomaly detection. The use of other features to improve the detection of certain anomalies will be explored in the next sections.

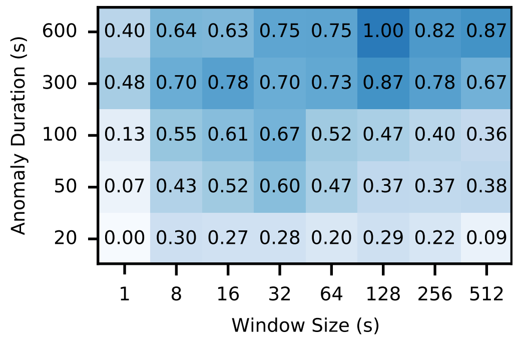
Finally, Fig. 4.20 shows the F-score obtained when using $Tr = 600$ and $Gr = 20$. Performance improves with respect to the previous setting, but the general trends remain. That is, longer windows are able to capture the behavior of the system better because are less sensitive to local changes.



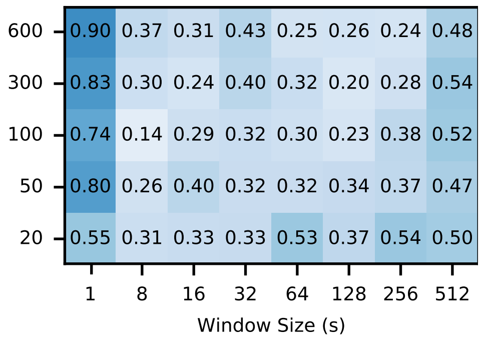
**(a)** Deadlock



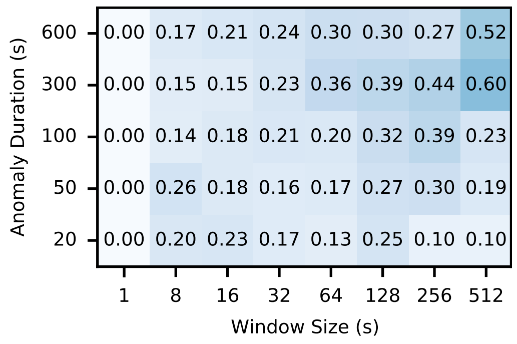
**(b)** Livelock



**(c)** Synchronization



**(d)** Memory Leak



**(e)** Starvation

**Fig. 4.20.** Mean F-score for different combinations of window size and anomaly duration ($Tr = 600$ and $Gr = 20$).

### 4.4.7 Optimal Parameters

Table 4.2 summarizes the results obtained in the parametric study by presenting the parameters that maximize our framework's anomaly detection performance when analyzing data from the SDS. These optimal parameters are system dependent (especially $\gamma$ and $\nu$). However, we have seen that large window sizes work better than short window sizes in general. This is likely to hold for most systems because the window

**Table 4.2.** Optimal parameter values.

| Parameter | Optimal Range |
|---|---|
| $\gamma$ | $(10^{-3}, 10^{-2})$ |
| $\nu$ | $(0.1, 0.4)$ |
| $Tr$ | $[100, 600]$ |
| $Gr$ | $[10, 50]$ |
| $z$ | $[64, 512]$ |

size affects the way our framework computes the similarity between time series, and it is independent from the system under study, the monitored metrics or the monitoring rate.

Additionally, the parametric study has shown that $Gr$ values between 10 and 50 can improve our framework's performance without increasing the detection time. This is also independent from the system being analyzed because $Gr$ affects the classification process but not the way system behavior is characterized.

Finally, $Tr$ should be set to the maximum possible value, since more training data typically produces better results. Nevertheless, the parametric study has shown that between 100 and 600 *Behavior Instances* are enough for our framework to produce acceptable results.

The best overall results are obtained with $Tr = 100$, $Gr = 50$, and $z = 512$. These results are presented in Table 4.3. We can see that deadlock, livelock and synchronization are detected with high accuracy, while starvation and memory leak obtain worse results. This is due to the proportion of performance metrics that the different anomalies affect. Deadlock, livelock and synchronization affect two metrics per node, whereas starvation and memory leak only affect one metric in one node.

**Table 4.3.** Results obtained when using the best overall parameters ($Tr = 100$, $Gr = 50$, and $z = 512$).

| Anomaly | *Recall* | *Precision* | DT (s) |
|---|---|---|---|
| Deadlock | 1.00 | 0.90 | 76 |
| Livelock | 1.00 | 0.95 | 83 |
| Synchronization | 1.00 | 0.93 | 92 |
| Memory Leak | 1.00 | 0.54 | 108 |
| Starvation | 1.00 | 0.42 | 13 |

## 4.5 Summary

In this chapter, we have employed a synthetic distributed system to perform an extensive study on complex anomalous behaviors, and how these affect the system performance metrics. In addition to this, we have employed monitoring data extracted from this synthetic system to carry out a parametric study on the framework presented in Chapter 3. By means of this study, we have gained insight on how various parameters affect our framework's ability to detect several complex anomalous behaviors.

By knowing the range of parameter values that maximize detection performance, we can employ our framework in different scenarios in a more effective manner. We have found that larger $Tr$ values result in less false positives, but a $Tr$ of 10 is enough to obtain a meaningful decision function that can detect many of the anomalies. Larger $Gr$ values produce better detection results in general, at the cost of increasing the detection time. A $Gr$ value between 10 and 50 results in the best balance between detection accuracy and time. Finally, we have found that longer window sizes produce better detection results because of the smoothing effect that this has on the monitoring data. Moreover, longer window sizes do not seem to have a negative impact on the detection time.

The results obtained prove that our framework is able to detect previously unseen complex anomalous behaviors with high accuracy. More precisely, when using the optimal parameters, our framework achieves 1.00 mean *Recall*, 0.75 mean *Precision*, and detection time below 100 observations.

In Chapter 5, we present a feature selection process that further explores how different behavior representations impact on our framework's performance.

# Chapter 5

# Feature Selection for Anomaly Detection and Identification

Black box anomaly detection is based on the detection of anomalous patterns in system metrics represented as time series. As mentioned in Chapter 4, there are three main patterns that can appear in the system performance metrics in the presence of anomalies: shifts, trends, and seasonality. Pattern changes in time series are also referred to as change points. Therefore, black box anomaly detection is closely related to change point detection in time series.

So far, we have employed four first order features to model the system metrics and to detect the different anomalous patterns that can appear. These features are mean, standard deviation, skewness and kurtosis (see Chapter 4). However, there are numerous other features that can be employed in the feature vector generation process [96], which might better to represent certain patterns, thus improving our framework's detection accuracy.

Additionally, anomaly identification is highly relevant for fault tolerance, as it allows system administrators to find the root causes of errors easily. Thus, in this chapter, we employ a publicly available dataset by Yahoo! [50] to evaluate the performance of our framework in the detection of change point anomalies. We analyze our framework's ability to identify the type of the different complex anomalous behaviors generated by our synthetic distributed system, and we present a comprehensive feature selection analysis to better understand how the feature extraction process influences our framework detection and identification performances. In this analysis we experiment with a variety of features that can capture different anomalous patterns in the system performance metrics.

## 5.1 Detecting Change Point Anomalies

In this section, we evaluate our framework's performance when detecting change point anomalies in time series. Change point anomalies are sudden changes in the distribution of values in a time series, which typically means a change in trend or seasonality. Being able to detect change point anomalies is relevant for our study, as most anomalous behaviors in distributed systems produce change points in the system metrics (see Chapter 4).

In these experiments we employ the Yahoo! Webscope labeled anomaly detection dataset [50]. This dataset is a publicly available dataset composed of real and synthetic time series. The real time series come from several Yahoo! services, whereas the synthetic data consists of 100 time series with varying trend, noise, and seasonality. We run our experiments using the synthetic data because the real time series do not contain change point anomalies.

Each of the 100 synthetic time series is composed of 1,681 data points, and have anomalies at random timestamps. We split these time series in smaller sections, and group these sections together to simulate the behavior of a distributed system from which several performance metrics are collected. We generate 1,320 groups of 65 time series of 150 data points long. Within each group, a random number (from 1 to 25) of time series contain an anomaly from timestamp 100 to the end of the time series. Each group of time series can be seen as the behavior of a system from which 65 performance metrics are collected. The first 100 data points are considered the normal behavior of the system, and the remaining 50 data points are considered anomalous. In the rest of this chapter, we will refer to this dataset as Yahoo-Dataset.

Fig. 5.1 depicts nine time series from one of the groups. The first three time series contain a change point anomaly at timestamp 100, and the rest of the time series are normal. Some of the time series also contain point anomalies, that is, an anomalous reading at a single data point. An example is the circled value at timestamp 43 in the last of the time series. However, we do not focus on detecting these kind of anomalies, as we are more interested in complex anomalous behaviors. Even though only three time series contain a change point anomaly, we consider the whole set of 65 time series as an anomalous behavior from timestamp 100 until the end.

In these experiments, we employ the same set of features and classifier parameters as in the parametric study presented in Chapter 4. Additionally, we use $Gr = 10$, $Tr = 50$, and three window sizes: 8, 16, and 32 data points.

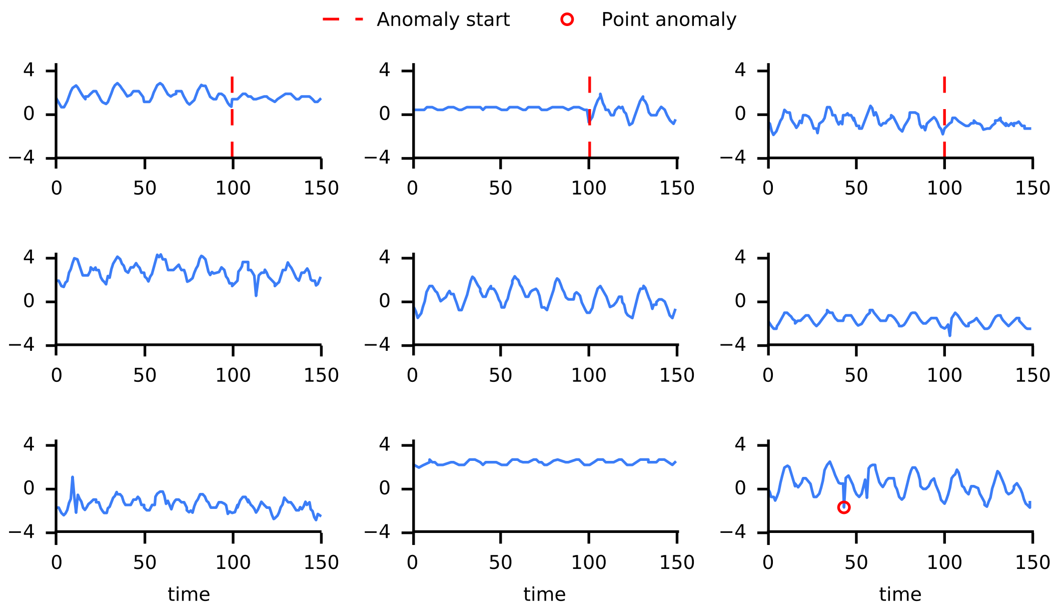
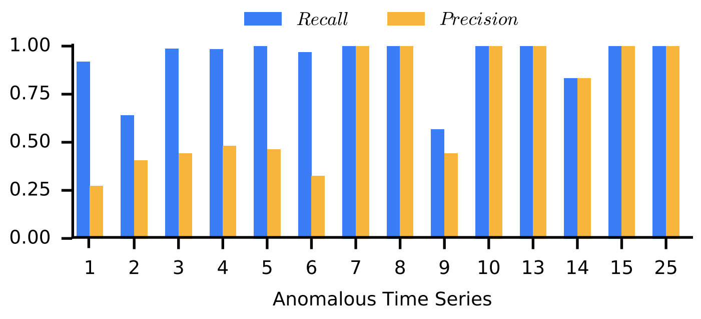


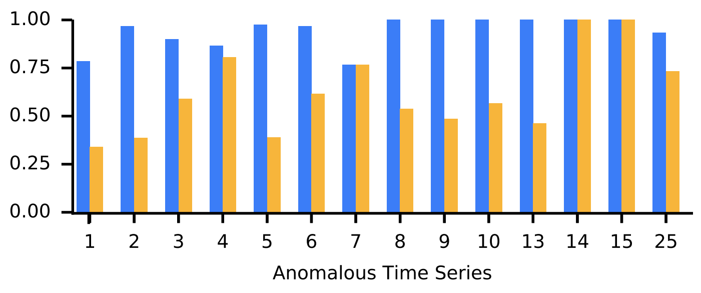
**Fig. 5.1.** Subgroup of time series from one of the sets generated from the Yahoo! data. The first three time series contain a change point anomaly at timestamp 100.

### 5.1.1 Results

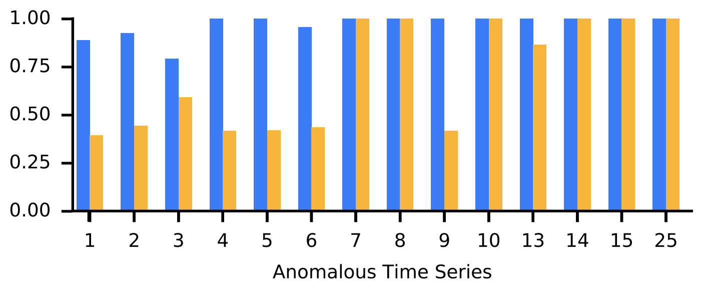
Fig. 5.2 shows mean *Recall* and *Precision* obtained in the 1,320 groups of time series, plotted by number of anomalous time series in the group. Performance improves as the number of anomalous time series increases. The minimum number of anomalous time series to achieve an acceptable performance is 7, which represents around 10% of the data. Even so, in some cases good performance is achieved with only 3 or 4 time series containing a change point (Fig. 5.2b and Fig. 5.2c).



**(a)** Window Size = 8



**(b)** Window Size = 16



**(c)** Window Size = 32

**Fig. 5.2.** Mean *Recall* and *Precision* obtained in the Yahoo! dataset for a varying number of anomalous time series ($Gr = 10$ and $Tr = 50$).

The best window size in this case is 32, however, window size does not seem to be as important as in the experiments presented in Chapter 4 with the SDS data. This can be due to the fact that a window of 8 data points is enough to capture the characteristics of the time series in this case, and to differentiate from normal and anomalous behaviors.

## 5.2 Identifying Anomaly Types

In this section we evaluate our framework's ability to identify the type of the different anomalous behaviors. This can help debugging, as it allows to identify the root cause of the anomalies easier.

In these experiments we employ a set of SDS monitoring data with four anomalies per run. We run the SDS for 200 minutes and inject four 10 minute anomalies at minutes 20, 70, 120, and 170. Let $t1$, $t2$, $t3$, and $t4$ be the type of the four anomalies, the first two anomalies are of different types in all runs (i.e., $t1 \neq t2$). Then, we perform 10 runs where $t1 = t3$ and $t2 = t4$, and 10 runs where $t1 = t4$ and $t2 = t3$. We do this for every combination of anomalies, obtaining 200 runs in total. For example, given deadlock and livelock, we perform 10 runs where the order of the anomalies is deadlock-livelock-deadlock-livelock; and 10 runs where the order of the anomalies is deadlock-livelock-livelock-deadlock. In the rest of this chapter, we will refer to this dataset as SDS-Dataset-2.

### 5.2.1 Experimental Setup

As said before, the SDS runs for 200 minutes with anomalies at minutes 20, 70, 120 and 170. We train our framework with the first 100 minutes of execution, including the two first anomalies (i.e., $Tr = 6000$), and evaluate the identification performance in the second half of the execution.

We use $Gr = 20$, and the same features and classifier parameter values used in Chapter 4. Feedback is also provided automatically after alerts and when the framework is in the *Alert* state. Finally, in these experiments, we include all the system metrics in the analysis, regardless of the anomaly being detected.

### 5.2.2 Evaluation Metrics

We employ a confusion matrix to evaluate our framework's performance of identifying anomaly types. Given two anomalies of types $t$ and $t'$, the element $C_{tt'}$ of the confusion matrix $C$ is the ratio between the number of times that $t$ has been classified as $t'$ and the number of times that $t$ has been detected. In a perfect scenario, $C_{tt'}$ would be 1 for $t = t'$ and 0 for $t \neq t'$.

### 5.2.3 Results

Tables 5.1, 5.2, and 5.3 show the confusion matrix obtained when using windows of 64, 128 and 256 seconds respectively. The scores are the mean over the 20 runs per anomaly pair.

All the anomalies except starvation are correctly identified in the majority of cases. This can be seen in the diagonal of the matrices. We also see that window size does not have a significant impact on the identification of the anomalies, as long as the window is long enough to allow detection.

**Table 5.1.** Confusion matrix using a window of 64 seconds.

| | Deadlock | Livelock | Synch. | Mem. Leak | Starvation |
|---|---|---|---|---|---|
| Deadlock | **0.68** | 0.24 | 0.08 | 0.00 | 0.00 |
| Livelock | 0.05 | **0.66** | 0.25 | 0.04 | 0.00 |
| Synch. | 0.13 | 0.02 | **0.84** | 0.00 | 0.01 |
| Mem. Leak | 0.03 | 0.03 | 0.21 | **0.66** | 0.07 |
| Starvation | 0.00 | 0.00 | 0.00 | 1.00 | **0.00** |

**Table 5.2.** Confusion matrix using a window of 128 seconds.

| | Deadlock | Livelock | Synch. | Mem. Leak | Starvation |
|---|---|---|---|---|---|
| Deadlock | **0.63** | 0.23 | 0.09 | 0.00 | 0.05 |
| Livelock | 0.07 | **0.57** | 0.26 | 0.10 | 0.00 |
| Synch. | 0.20 | 0.05 | **0.71** | 0.02 | 0.02 |
| Mem. Leak | 0.20 | 0.04 | 0.16 | **0.52** | 0.08 |
| Starvation | 0.00 | 0.00 | 0.00 | 0.50 | **0.50** |

**Table 5.3.** Confusion matrix using a window of 256 seconds.

| | Deadlock | Livelock | Synch. | Mem. Leak | Starvation |
|---|---|---|---|---|---|
| Deadlock | **0.70** | 0.23 | 0.03 | 0.02 | 0.03 |
| Livelock | 0.07 | **0.53** | 0.28 | 0.10 | 0.02 |
| Synch. | 0.14 | 0.08 | **0.75** | 0.02 | 0.01 |
| Mem. Leak | 0.11 | 0.00 | 0.05 | **0.68** | 0.16 |
| Starvation | 0.00 | 0.00 | 0.00 | 0.67 | **0.33** |

Deadlock is mostly confused with livelock, however, livelock is mostly confused with unwanted synchronization, and unwanted synchronization is confused with deadlock. This suggests that our classifier is slightly biased towards the last anomaly observed. In the runs containing deadlock and livelock, the anomaly injected at minute 50 is always livelock even when the anomalies at minutes 70 and 170 are swapped. The anomaly at minute 50 is the second anomaly used to train the classifier (the first being at minute 20), and the type of this anomaly obtains higher scores in the confusion tables. Despite of this bias the identification performance is over 0.6 in most cases.

Starvation is misclassified as memory leak in many cases because both anomalies are very similar from the point of view of network and CPU utilization, which are the system metrics that have a higher influence in the classification process. Moreover, our framework is unable to detect starvation in many cases due to the low impact that the anomaly has on the system performance metrics, and in the runs containing both anomalies, memory leak is injected at minute 50, producing a bias towards this anomalous behavior.

## 5.3 A Comprehensive Feature Selection Analysis

As seen in Chapter 4, especially in the case of memory leak, the way system behavior is represented is crucial for the detection and identification of anomalies. Our framework represents behavior as feature vectors (see Chapter 3), which are defined by the window size and the set of features $S_F$ extracted from the system metrics. In this section we explore how different features affect our framework's performance.

In these experiments we employ the SDS-Dataset-2, the Yahoo-Dataset, and another set of monitoring data consisting of 10 runs per anomaly type of 100 minutes long, where the same 10 minute anomaly is injected twice (at minutes 30 and 80). We experiment with deadlock, livelock, unwanted synchronization, and memory leak. We will refer to this dataset as SDS-Dataset-3. In this case, we do not use the starvation anomaly because the impact that it has on the system metrics is not significant enough to allow its detection.

### 5.3.1 Experimental Setup

With the anomaly detection dataset we use the same setup as in the parametric study presented in Chapter 4, whereas with the anomaly identification dataset we use the same setup as in the anomaly identification section (Section 5.2).

### 5.3.2 Evaluation Metrics

Apart from the metrics used in previous experiments, we redefine *Recall* and *Precision* in respect to the number of correct identifications to evaluate the performance of different features in identifying the anomaly types. That is, given an anomaly of type $t$, we define

$$
\begin{aligned}
Recall_t &= \frac{\text{number of times } t \text{ is correctly identified}}{\text{number of times } t \text{ is detected}} \\
Precision_t &= \frac{\text{number of times } t \text{ is correctly identified}}{\text{number of times an anomaly is identified as } t} .
\end{aligned}
\tag{5.1}
$$

These two metrics and the corresponding F-score (i.e., their harmonic mean) give an idea of how well an anomaly type is identified, and how often an anomaly type is confused with others.

### 5.3.3 Features

A feature is any relevant information that represents a characteristic of a time series. There is a large amount of features that can be extracted from time series [96]. For example, statistical features such as mean or standard deviation represent certain properties of the probability distribution of the values in the series. Applying certain transformations to a time series can also be used to obtain new knowledge. For example, the autocorrelation function [54] reveals periodic patterns in the data. Other techniques that can be used to extract features from time series are frequency domain transformations such as the discrete Fourier transform (DFT) or the discrete wavelet transform (DWT) [97]; regression models [98]; time series analysis models such as the autoregressive integrated moving average model (ARIMA) [53, 54]; and chaotic analysis functions such as the Lyapunov exponent [99].

In our experiments, we employ 17 features to capture the different patterns that anomalous behaviors produce in the system metrics, that is, shifts, trends and seasonality (see Chapter 4). The employed features are summarized in the following.

### First Order Features

First order features are the statistical features employed in the parametric study in Chapter 4, that is, mean (*ME*), standard deviation (*SD*), skewness (*SK*), and kurtosis (*KU*) [79]. These features, defined in Chapter 4, are essential to represent the underlying distribution of values in a time series, and can help especially in detecting shifts.

### Second Order Features

Second order features are co-occurence features typically employed in image processing that have been adapted to find similarities between time series [79]. Second order features include energy (*EG*), entropy (*EP*), correlation (*CO*), inertia (*IN*), and local homogeneity (*LH*). To compute these features, the data is first quantized into $q$ levels, that is, data points in the time series are mapped to a finite number of predefined values. Then, a matrix $D$ is computed where $D_{ij}$, for $1 \leq i, j \leq q$, contains the number of times a data point with level $i$ in the quantized series is followed, at a distance $d$, by a data point with level $j$. Using this matrix, second order features are defined as

$$EG = \sum_{i=1}^{q} \sum_{j=1}^{q} (D_{ij})^2 , \tag{5.2}$$

$$EP = \sum_{i=1}^{q} \sum_{j=1}^{q} D_{ij} \cdot log(D_{ij}) , \tag{5.3}$$

$$IN = \sum_{i=1}^{q} \sum_{j=1}^{q} (i-j)^2 D_{ij} , \tag{5.4}$$

$$LH = \sum_{i=1}^{q} \sum_{j=1}^{q} \frac{1}{1+(i-j)^2} D_{ij} , \tag{5.5}$$

and

$$CO = \sum_{i=1}^{q} \sum_{j=1}^{q} \frac{(i-\mu_x)(j-\mu_y)D_{ij}}{\sigma_x \sigma_y} , \tag{5.6}$$

where

$$\mu_x = \frac{\sum_{i=1}^{q} i \sum_{j=1}^{q} D_{ij}}{q}, \tag{5.7}$$

$$\mu_x = \frac{\sum_{j=1}^{q} j \sum_{i=1}^{q} D_{ij}}{q}, \tag{5.8}$$

$$\sigma_x = \sqrt{\frac{\sum_{i=1}^{q} (i-\mu_x)^2 \sum_{j=1}^{q} D_{ij}}{q}}, \tag{5.9}$$

and

$$\sigma_x = \sqrt{\frac{\sum_{j=1}^{q} (j-\mu_y)^2 \sum_{i=1}^{q} D_{ij}}{q}}. \tag{5.10}$$

Second order features give information about the distribution of close values in a time series. Thus, they can be used in detecting shifts and periodic patterns. In the experiments, we employ $d = 1$ and $q = 10$.

### Linear Regression

One of the simplest regression models that can be applied to a time series is the linear regression model [98]. This model consists in fitting a line to the data typically by minimizing the difference between the values in the series and a value in the fitted line.

The fitted line takes the form $y = \beta_0 + \beta_1 x$, where $\beta_0$ is the y-intercept, and $\beta_1$ is the slope. We fit a linear model to the time series and extract five features: $\beta_0$ ($B0$), $\beta_1$ ($B1$), the standard error of $\beta_0$ ($S0$), the standard error of $\beta_1$ ($S1$), and the *coefficient of determination* ($R2$), defined as

$$R2 = \frac{\sum_{i=1}^{n} (y_i - \mu)^2}{\sum_{i=1}^{n} (x_i - \mu)^2}, \tag{5.11}$$

for a sequence of values $x_1, ..., x_n$, where

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i . \tag{5.12}$$

The coefficient of determination is a measure of how well the model fits the actual data. These regression features are useful to reveal trend changes in the data.

**Discrete Fourier Transform**

The discrete fourier transform (DFT) converts a sequence of values to the frequency domain. The DFT is widely used in signal processing, but has also been employed to find similarities in time series [100]. Given a sequence of values $x_0, ..., x_n$, the DFT produces a transformed sequence of complex numbers $c_0, ..., c_n$. From this sequence, we can obtain a sequence of real numbers by computing the absolute value as

$$|c| = \sqrt{[\mathrm{Re}(c)]^2 + [\mathrm{Im}(c)]^2}, \tag{5.13}$$

where Re($c$) and Im($c$) are the real and imaginary parts of $c$ respectively.

When values in the original sequence are real numbers, the transformed sequence is symmetric, that is, $|c_i| = |c_{n-i}|$ for $0 \leq i < \frac{n}{2}$. Thus, we can use only the first $\frac{n}{2}$ transformed values.

The initial values of the transformed sequence represent low frequencies, whereas the last values of the sequence represent high frequencies. There has been some discussion in the literature as to which frequencies of the transform provide better information in order to find similarities in time series [100]. In our case, we experiment with both by extracting two features from the transformed sequence:

$$FH = |c_i| \quad \text{for} \quad 0 \leq i \leq \frac{k}{4} \tag{5.14}$$

and

$$FL = |c_i| \quad \text{for} \quad \frac{3k}{4} \leq i \leq k, \tag{5.15}$$

where $k = \frac{n}{2}$. In other words, $FH$ and $FL$ are the 25% initial and last values respectively of the first half of the DFT.

Since the DFT transforms the data to the frequency domain, it is extremely suited to discover periodic patterns.

#### Lyapunov Exponent

The Lyapunov exponent (*LE*) measures the rate of separation of infinitesimally close trajectories in dynamical systems [101]. The Lyapunov exponent can be used to measure how chaotic a time series is. Given a time series $x_i,...,x_n$, *LE* is defined as [99]

$$LE = \frac{1}{n}\sum_{i=1}^{n}\lambda(x_i,x_j^*), \tag{5.16}$$

where $x_j^*$ is the closest value to $x_i$ in the series such that $i \neq j$, and

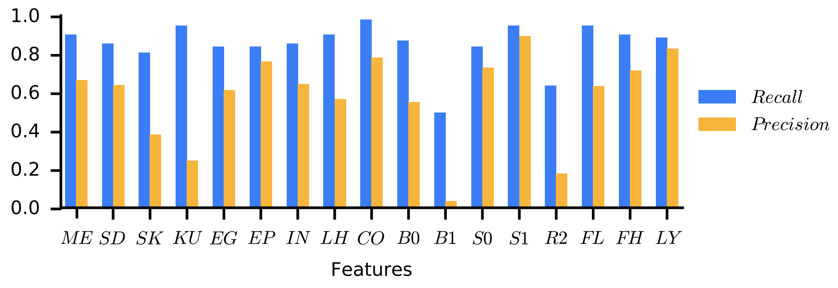
$$\lambda(x_i,x_j) = \frac{1}{n}log\frac{|x_{j+d}-x_{i+d}|}{|x_j-x_i|}, \tag{5.17}$$

where $d$ is a predefined distance.

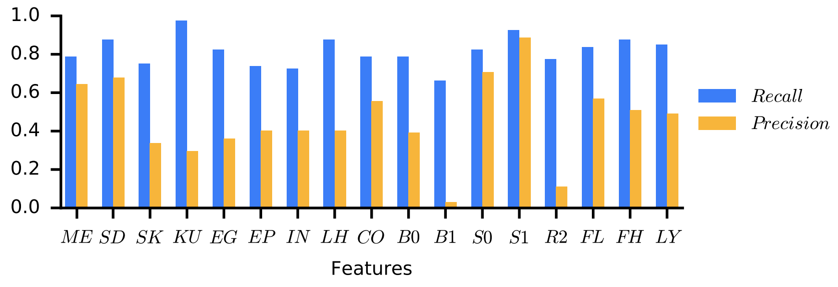
The Lyapunov exponent therefore measures how much close values in the series deviate from each other over time. In the experiments we employ $d = 0.1z$, where $z$ is the window size.

### 5.3.4 Anomaly Detection

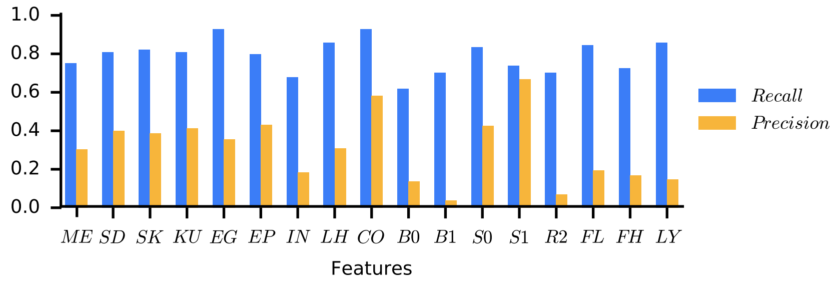
In a first experiment, we evaluate our framework's detection performance when using different features individually. Fig. 5.3 shows mean *Recall* and *Precision* obtained in the SDS-Dataset-3 and the Yahoo-Dataset. With the SDS data we use $Gr = 20$, $Tr = 400$, and windows of 32, 64, 128 and 256 seconds. With the Yahoo! data we use $Gr = 10$, $Tr = 50$, and windows of 8, 16 and 32 data points. Results are averaged over all window sizes to get a better idea on how well different features perform regardless of the window employed.



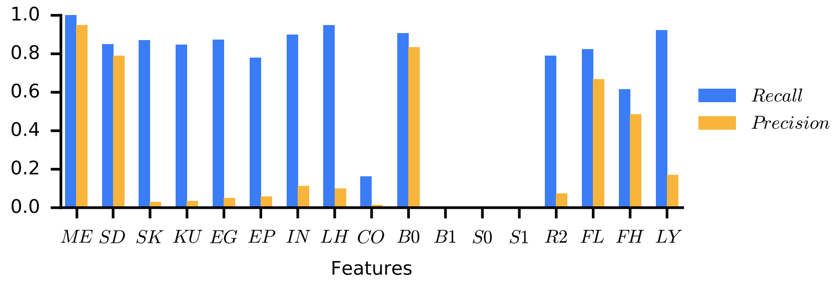
**(a)** Deadlock



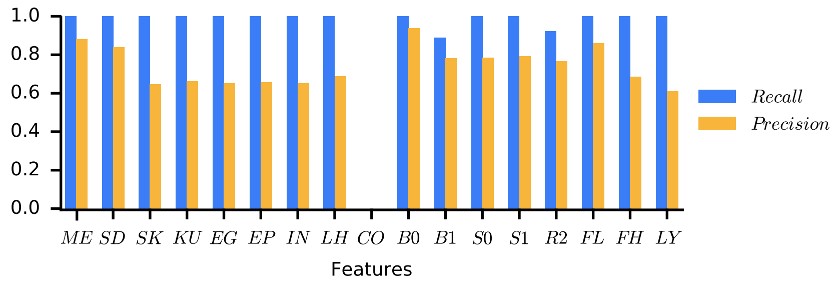
**(b)** Livelock



**(c)** Synchronization



**(d)** Memory Leak



**(e)** Yahoo!

**Fig. 5.3.** Mean *Recall* and *Precision* obtained with the different individual features.

Deadlock is best detected using $S1$ and $LY$. $S1$ obtains a *Recall* and *Precision* of 0.95 and 0.90, whereas $LY$ obtains 0.89 and 0.83. In the case of livelock, also $S1$ obtains the best results with 0.92 *Recall* and 0.89 *Precision*. For synchronization, $CO$ works best with 0.93 *Recall* and 0.58 *Precision*. The memory leak is best detected with $ME$ and $B0$. $ME$ obtains 1.00 *Recall* and 0.95 *Precision*, and $B0$ obtains 0.91 *Recall* and 0.83 *Precision*. In the case of the Yahoo! anomalies, also $ME$ and $B0$ obtain the highest performance. $ME$ achieves 1.00 *Recall* and 0.88 *Precision*, and $B0$ achieves 1.00 *Recall* and 0.94 *Precision*. $S1$ and $CO$ are in general better at dealing with high variance in the system metrics, like network utilization in the deadlock, livelock, and synchronization cases. Conversely, $B0$ and $ME$ work better in the cases where there is not such a great variance in the metrics, like in the memory leak and Yahoo! anomalies. The features that obtain the highest overall performance are $ME$ and $SD$. This is because these two features represent the underlying distribution of values in the system metrics very well. Regarding $FL$ and $FH$, both features obtain similar results, with $FL$ being slightly superior in some cases. Apart from $CO$, second order features like $EG$, $IN$ or $LH$ do not achieve good performance, and the worst overall features are $B1$ and $R2$. It is clear that there is a trade-off between detection accuracy and the number of anomalies that can be detected. For example, livelock can be detected with very high accuracy using $S1$, but $ME$ and $SD$ provide better detection across different types of anomalies.

We are also interested in knowing if a combination of multiple features can improve detection accuracy across different types of anomalies. Since evaluating every feature subset is unfeasible, we perform a simulated annealing search [102], and find that the set of features with the best overall performance is $\{CO, ME, B0, S1\}$. Table 5.4 shows the mean F-score obtained for this set compared to the F-score of the best individual feature ($ME$). The set $\{CO, ME, B0, S1\}$ sacrifices some performance in detecting the memory leak to achieve an overall mean improvement of 15% over $ME$. While

**Table 5.4.** Mean detection F-score of the set of features $\{CO, ME, B0, S1\}$ versus $ME$.

| Anomaly | $ME$ | $\{CO, ME, B0, S1\}$ | Improvement |
|---|---|---|---|
| Deadlock | 0.74 | 0.91 | 22% |
| Livelock | 0.69 | 0.79 | 14% |
| Synchronization | 0.41 | 0.65 | 58% |
| Mem. Leak | 0.97 | 0.81 | -16% |
| Yahoo! | 0.92 | 0.90 | -2% |
| Mean | 0.75 | 0.81 | 15% |

features $S1$ and $CO$ reinforce each other in the detection of deadlock, livelock and synchronization, $ME$ and $B0$ enable the detection of the memory leak.

Finally, we test the set $\{CO, ME, B0, S1\}$ in SDS-Dataset-2. Table 5.5 shows *Recall*, *Precision* and detection time averaged over 120 runs with different anomalies. We use $Tr = 400$, $Gr = 20$, and a window of 128 seconds.

**Table 5.5.** Average *Recall*, *Precision* and DT using the set of features $\{CO, ME, B0, S1\}$ in the SDS-Dataset-2 ($Tr = 400$, $Gr = 20$, $z = 128$).

| | *Recall* | | *Precision* | | DT (s) | |
|---|---|---|---|---|---|---|
| | Mean | Std. Dev. | Mean | Std. Dev. | Mean | Std. Dev. |
| Deadlock | 0.99 | 0.11 | 0.86 | 0.19 | 208 | 54 |
| Livelock | 0.92 | 0.27 | | | 215 | 59 |
| Synchronization | 0.62 | 0.48 | | | 158 | 43 |
| Memory Leak | 0.02 | 0.15 | | | 291 | 3 |

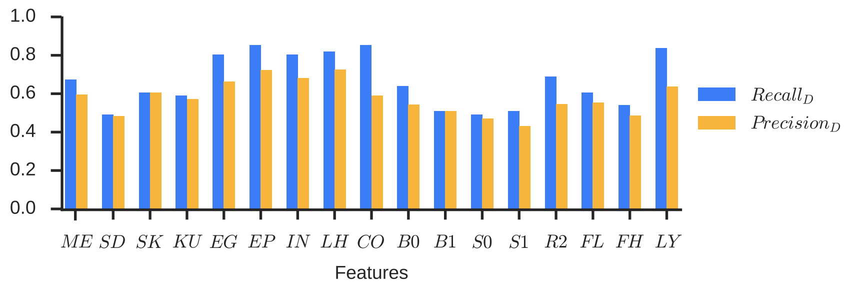
It can be seen how deadlock, livelock and synchronization are correctly detected with high accuracy. The drop in the detection of the memory leak is due to the introduction of network utilization in the analysis. This system metric has so much variance that it masks the trend in memory usage generated by the memory leak. *Precision* is computed overall since false positives cannot be associated with a particular anomaly in a run containing anomalies of different types. The slight drop in *Recall* in synchronization when compared with previous experiments, can be explained by the fact that the classifier parameters that enable the detection of one anomaly with high accuracy might not allow the detection of other anomalies. In other words, being able to detect multiple anomalies in the same run comes at the cost of overall accuracy. Even so, most anomalies are detected in less than 200 seconds (or obsevations) with high *Precision* (0.86). A solution to the memory leak problem would be to have dedicated classifiers for certain system metrics, the use of different features with different metrics, or the use of a combination of multiple classifiers with different settings (e.g., several window sizes).

### 5.3.5 Anomaly Identification

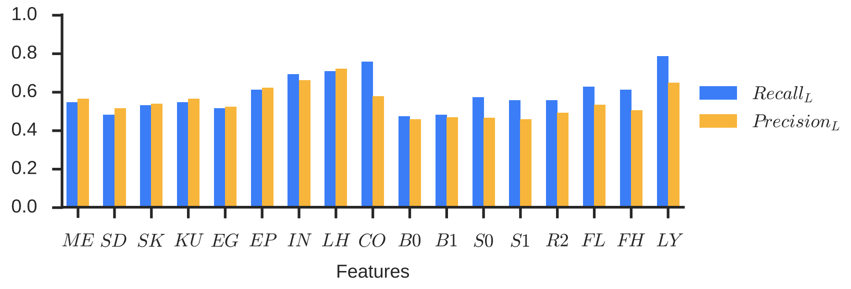
In a second experiment, we evaluate how different features perform when identifying anomaly types. We employ SDS-Dataset-2 using the first 100 minutes of execution to train our framework. We set $Gr = 20$, $Tr = 6000$, and $z = 128$. Fig. 5.4 shows mean $Precision_t$ and $Recall_t$ per anomaly type using different features.

In this case, the results are much more consistent across the different types of behavior, with *LH* outperforming the rest of the features in all cases. It seems that second order features are more useful for identifying the type of the anomalies than for detecting them, whereas the opposite happens to the linear regression features. This can be caused by similarities across the different anomalous behaviors, such as low $S0$ and $S1$, that make them indistinguishable when using these features.

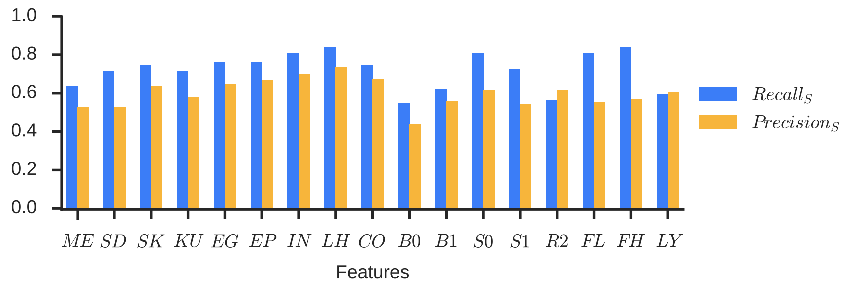
Next, we run a simulated annealing search to know if the identification performance can be improved by combining multiple features. We find that the optimal set is $\{LH, IN\}$, with a slightly better performance than *LH*. Other sets found in the simulated annealing process, such as $\{LH, IN, EP\}$, can also outperform *LH* in certain cases, but using more than these three features always reduces performance. This is because *LH* outperforms the rest of the features in all cases. Including *IN* or *EP* can reinforce the identification of anomalies to a certain degree, but adding more features beyond that only hinders performance. Table 5.6 shows the a comparison of the the F-score obtained by $LH$, $\{LH, IN\}$, and $\{LH, IN, EP\}$.



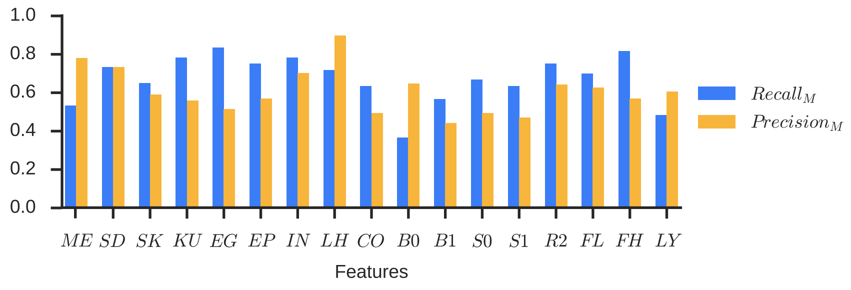
**(a)** Deadlock



**(b)** Livelock



**(c)** Synchronization



**(d)** Memory Leak

**Fig. 5.4.** Mean $Recall_t$ and $Precision_t$ obtained with the different individual features.

**Table 5.6.** Mean identification F-score of the set of features $\{LH, IN\}$ versus $\{LH, IN, EP\}$ and $LH$.

| Anomaly | $LH$ | $\{LH, IN, EP\}$ | $\{LH, IN\}$ | Improvement over $LH$ |
|---|---|---|---|---|
| Deadlock | 0.77 | 0.80 | 0.82 | 6.1% |
| Livelock | 0.71 | 0.75 | 0.75 | 4.4% |
| Synchronization | 0.78 | 0.76 | 0.76 | -3.7% |
| Mem. Leak | 0.80 | 0.76 | 0.79 | -1.3% |
| Mean | 0.77 | 0.77 | 0.78 | 1.4% |

Finally, Table 5.7 shows the confusion matrix obtained when using $\{LH, IN\}$ as the feature set. Comparing these results with Table 5.2 in Section 5.2 we see that there is a significant improvement in the identification performance when using $\{LH, IN\}$. More precisely, deadlock, livelock and unwanted synchronization identification improves in more than 20%, and memory leak identification improves in 68%, going from 0.52 to 0.88.

**Table 5.7.** Confusion matrix obtained using the feature set $\{LH, IN\}$ and a window of 128 seconds.

| | Deadlock | Livelock | Synch. | Mem. Leak |
|---|---|---|---|---|
| Deadlock | **0.77** | 0.09 | 0.09 | 0.06 |
| Livelock | 0.07 | **0.69** | 0.22 | 0.02 |
| Synch. | 0.04 | 0.02 | **0.88** | 0.05 |
| Mem. Leak | 0.06 | 0.00 | 0.06 | **0.88** |

## 5.4 Discussion

Experimenting with the Yahoo! data (Section 5.1) showed that, in some cases, the proportion of anomalous data needs to be greater than $\sim$10% to allow detection. In other cases, like memory leak in the SDS-Dataset-3, the anomaly is detected despite of representing as little as $\sim$2% of the feature vector. The major factor that hinders the detection of certain anomalies is the variance of the system metrics. Metrics with high variance tend to bias the classification process and can mask anomalies that mainly affect other metrics with lower variance. Solutions to this problem can be the use of other types of classifiers, or applying some noise reduction technique to smooth certain metrics.

In addition, we have seen that an adequate selection of features is essential. One of the best features for detection in many cases is $B0$. A reason for this can be that the y-intercept serves a as measure of trend or the distribution of values (similar to $ME$) depending on the case. Nevertheless, different features capture different anomalies, and even though combining multiple features improves overall performance, it comes at the cost of some loss in the detection of certain anomalies. Moreover, features that are useful to detect anomalies differ from features that are useful to identify their type. The use of multiple behavior representations, employing different sets of features (and maybe different window sizes), might improve overall detection and identification rates in scenarios with many different types of anomalies.

In any case, this chapter provides valuable insight on which features to employ in black box anomaly detection to model system behavior, as the patterns that the system metrics exhibit under anomalous conditions (i.e., trends, seasonality changes, and shifts) are likely to be similar even for different systems and architectures. In this sense, we have found that $CO$, $ME$, $B0$, and $S1$ are the best features for anomaly detection, and that $LH$, $IN$ and other second order features work better for anomaly identification.

## 5.5 Summary

In this chapter we have analyzed the performance of the framework presented in Chapter 3 when detecting change point anomalies, and identifying complex anomalous behaviors. Moreover, we have carried out a feature selection analysis to discover which features capture better the different anomalous patterns that can appear in the system metrics.

Our results show that our framework is able to detect different types of anomalous behaviors with 0.65 mean *Recall* and over 0.80 mean *Precision*, reaching over 0.90 F-score for certain anomaly types. Moreover, our framework is also able to identify the type of complex anomalous behaviors with over 0.80 mean accuracy when employing the adequate features. We found that first order features, such as mean and standard deviation, and linear regression features are useful to detect previously unseen anomalies, whereas second order features are more useful to identify their type.

In the future, a real world dataset should be employed to validate our results, while other improvements could be added, such as using a different behavioral representation for detection and identification, combinations of multiple classifiers, or finding a practical way of choosing the optimal classifier parameters in an online setting.

# Chapter 6

# Decentralized Anomaly Detection in Large-Scale Systems

Dependability in large-scale distributed systems is especially challenging due to their size and complexity. The larger a system is, the higher the probability of errors, and effective anomaly detection in these systems still remains an open problem, as demonstrated by numerous recent failure events [14, 103, 104, 105], such as the cascading failure that forced Southwest Airlines to cancel more than 2,000 flights in August 2016 [14].

Dependability in large-scale systems is also affected by the decentralized and non-linear nature of these systems. Key challenges include extremely large computational and storage requirements, the great variety of behaviors that these systems can display, and the difficulty in analyzing high dimensional data.

In Chapter 3, we present a system of systems setting that our framework can employ to deal with particularly large scenarios in a scalable manner. In this chapter we analyze the problems of anomaly detection in large-scale systems, employ the decentralized system of systems setting to address this problem in an efficient and scalable way, and evaluate the performance of the framework presented in Chapter 3 when dealing with a large-scale dataset.

## 6.1 Overview

Even though many current distributed systems can be considered large-scale systems, a precise definition of what constitutes a “large” system is difficult to obtain. Examples of systems that are undoubtedly large-scale are the data centers maintained by Internet giants such as Google and Amazon. The exact dimensions of these data centers are unknown, but reports estimate that each company uses over 2 million servers to provide their services [44, 45]. This means that a typical data center could be composed of 50,000 to 80,000 servers. Another example are the supercomputers appearing at the *Top 500* list [106]. The first computer on the November 2016 list is the Sunway TaihuLight [107], which is composed of almost 50,000 nodes with a combined amount of over 10 million cores.

Existing works that have addressed the anomaly detection problem specifically for large-scale distributed systems [30, 35, 39, 43, 66, 108, 109], have experimented with systems composed of around 300 nodes, and datasets containing from 500 to 30,000 time series. According to these numbers, a system can be considered large-scale when it is composed of hundreds to tens of thousands of nodes. Nevertheless, there is a gap between the state of the art in anomaly detection and real world large-scale deployments. This gap can be explained by the challenges associated with processing large amounts of information, and the lack of publicly available datasets.

It is clear that detecting anomalies in large-scale computer systems introduces new challenges. On the one hand, the amount of monitoring data that these systems can potentially generate is very large. Collecting 60 to 80 performance metrics per node in a data center of 50,000 servers generates 3 to 4 million readings per unit time. This establishes enormous storage and computational requirements that need to be addressed in scalable ways. On the other hand, anomaly detection in high dimensional data is subject to *the curse of dimensionality* [110]. That is, as the dimension of the data increases, the distance between normal and abnormal data items decreases, making them indistinguishable from the point of view of a detector. This is because, when comparing two $n$-dimensional vectors for a large $n$, a high number of irrelevant vector elements can easily mask the differences between the few elements that are relevant.

In this chapter, we analyze the applicability of the framework described in Chapter 3 to a large-scale scenario. We describe the limitations of our framework in such scenarios, describe how to overcome these limitations, and evaluate our framework's performance. The rest of this chapter is organized as follows: Section 6.2 discusses the dimensionality problem, Section 6.3 presents our framework decentralized setting, Section 6.4 describes the large-scale data generation process employed, Section 6.5 shows the experimental analysis, and Section 6.6 concludes the chapter.

## 6.2 The Curse of Dimensionality

The data dimensionality problem in anomaly detection is especially relevant in large-scale scenarios since the monitoring data is inevitably high dimensional. It has been proven that [111]:

$$\lim_{n\to\infty} \frac{d_{max} - d_{min}}{d_{min}} = 0, \tag{6.1}$$

for many distance measures such as the Euclidean distance, where $n$ is the dimension of the data, and $d_{max}$ and $d_{min}$ are the maximum and minimum vector distances respectively in the $n$-dimensional space. This implies that differentiating between near and far elements becomes more difficult as $n$ increases. This is referred to as the curse of dimensionality.

One of the most typical approaches when dealing with high dimensional data in anomaly detection is to employ dimensionality reduction techniques before data processing [109]. A widely used dimensionality reduction technique is *principal component analysis* (PCA) [63]. PCA transforms $n$-dimensional data to a $p$-dimensional space, where $n > p$, by combining dimensions in a way that retains the maximum possible variance. By using PCA in anomaly detection, irrelevant and duplicate information is removed from the analysis, potentially improving detection rates.

However, dimensionality reduction techniques define the dimensional transformation based on data available at the training phase. This limits the effectiveness of these techniques in situations where training data is unavailable or incomplete, such as in many real world anomaly detection scenarios. For example, PCA might consider memory utilization irrelevant if it displays a constant value under normal conditions. However, eliminating memory usage from the analysis prevents the detection of memory leaks, or other memory related anomalies, in the future. Since our framework assumes a lack of historical data, we address the data dimensionality problem by employing a decentralized approach instead.

## 6.3 A Decentralized Deployment

We address the data dimensionality problem by employing the decentralized system of systems setting introduced in Chapter 3. This setting, depicted in Fig. 6.1, is useful in scenarios where the system can be divided into multiple subsystems in terms of its behavior. However, it can also be used to split the monitoring and analysis of large-scale systems in order to distribute the computational load, and to ameliorate the negative effects of high dimensional data.

The setting is composed of several sets of *Behavior Extractor*, *Behavior Identifier* and *Feedback Provider*, which we will refer to as *instances*. Each instance monitors a subset of nodes or system components, and builds a behavior representation of this subset. If the subset's behavior representation deviates from what is considered normal by the instance, an alert is generated. Alerts are collected by the *Feedback Aggregator*, which notifies the system administrator if necessary. The simplest approach here is to forward any alert received, but other techniques can be implemented such as

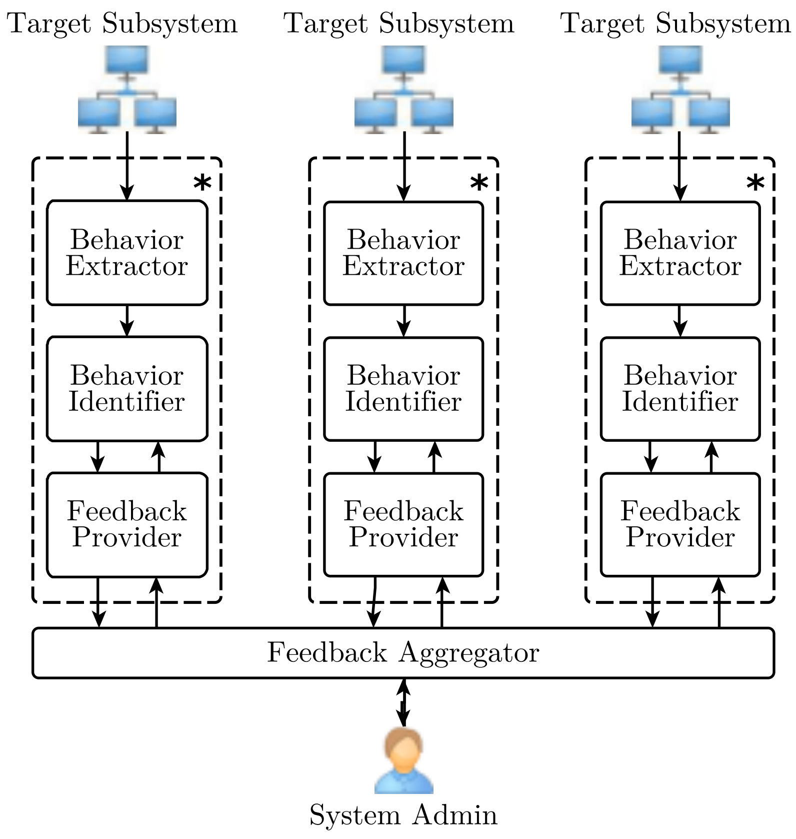


**Fig. 6.1.** Setting employed to analyze distributed systems from a system of systems perspective.

voting or weighted alerts. Instances are independent from each other, and the only communication happens between individual instances and the *Feedback Aggregator*. This provides a great degree of flexibility and scalability that are highly beneficial in large-scale systems.

Detection performance when using this decentralized setting depends on the number of deployed instances and the partitioning of the different subsystems. More instances implies lower dimensional data per instance. However, it also means a more localized analysis that might miss system-wide anomalies, or anomalies characterized by abnormal correlations between components monitored by different instances. Thus, subsystems should be ideally composed of tightly coupled components to obtain a complete characterization of each subsystem's behavior. This division can be achieved to a certain extent in many large-scale systems as component interaction is not uniform across the whole system. That is, subgroups of tightly coupled components can be typically identified. Nevertheless, in the following sections we analyze the impact on our framework's performance of different partitioning strategies under various anomalous conditions.

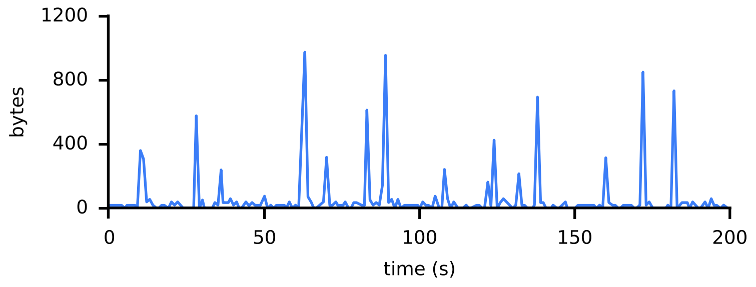
## 6.4 Data Generation

Due to the lack of publicly available datasets, we generate our own data using monitoring data from the synthetic distributed system (SDS) presented in Chapter 4. We replicate and concatenate data obtained from SDS runs to simulate a larger system running for a longer period.

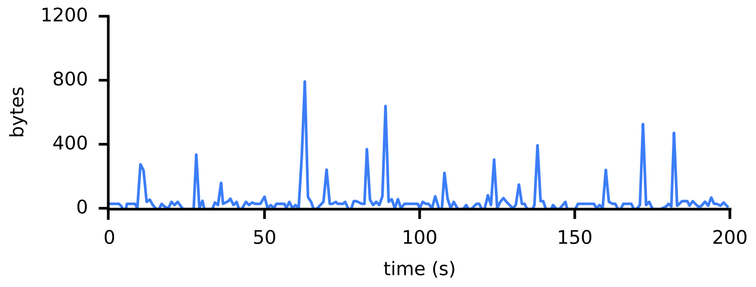
We start with data collected from 250 SDS runs per anomaly type, with deadlock, livelock, memory leak and unwanted synchronization. Each run consists of 5 nodes with 12 performance metrics per node. We split the normal and anomalous part of each node, and obtain 60,000 anomalous time series of different lengths, and around 540,000 normal time series.

We then employ a *bootstrapping* [112] algorithm to create 25 replicas of each time series. Bootstrapping is a technique to replicate time series that maintains certain properties of the original data, like mean and variance. Fig. 6.2 depicts a time series representing 200 seconds of network output and its replica. The replica introduces slight variations to the original data, but keeps the general distribution of values.

After the bootstrapping process, we end up with 1.5 million anomalous time series and 13.5 million normal time series. These time series are of different lengths, and are grouped into 1.25 million nodes with 12 metrics per node. We use this data as a pool of time series of the system metrics under different anomalous conditions. We then concatenate randomly chosen nodes from this pool to generate 120 10-hour long runs of 1,000 nodes each. Each of the runs contains 10 anomalies of 10 minutes long, at random timestamps, and of random types. The number of nodes affected by the



**(a)** Original



**(b)** Replica

**Fig. 6.2.** Network output and a time series replica for a period of 200 seconds.

anomalies ranges from 1 to 100, with 10 runs per each type. This means that at most 100 nodes out of 1,000 display an anomalous behavior simultaneously. This represents 10% of the system, and between the 0.33% and the 0.66% of the time series depending on the anomaly type. We experiment with 60,000 time series of 36,000 data points per run, greatly surpassing the system size employed by similar works [35, 39, 108, 109].

Our large-scale dataset has two main limitations. Firstly, the bootstrapping algorithm generates replicas that are very similar to the original series, limiting the variety of available data. Secondly, the concatenation process eliminates the existing correlations between the different nodes. In real systems, as a result of node interaction, certain metrics may be highly correlated to others. These correlations do not exist in our dataset.

However, since the concatenation process creates 10 hour time series using multiple shorter series chosen at random from a large pool of data, a sufficient variety in the data is guaranteed. In addition, the lack of correlations between nodes is a disadvantage for the classification process because there is less information that can be used to detect the anomalies. Moreover, changes in the correlation of metrics and changes in the underlying distribution of metric values due to the concatenation process can generate false positives. In other words, any deviation introduced in the system metrics due to the concatenation process cannot improve the measured performance because our framework considers these deviations as (false) anomalies. This means that the detection performance observed when using this dataset is a lower bound on the optimal performance. In other words, the limitations of the dataset do not introduce a positive bias in the results.

## 6.5 Experimental Analysis

In this section we study how the proportion of anomalous data processed by each instance and the number of instances that are affected by each anomaly impact on our framework's performance in large-scale scenarios. More precisely, we experiment with four anomalous profiles, as shown in Table 6.1.

**Table 6.1.** Different profiles evaluated.

| Profile | Density | Dispersion |
|---|---|---|
| High Density High Dispersion | 80% - 100% | 10 - 20 |
| Low Density High Dispersion | 20% - 40% | 10 - 20 |
| High Density Low Dispersion | 80% - 100% | 1 - 5 |
| Low Density Low Dispersion | 20% - 40% | 1 - 5 |

Density refers to the number of anomalous nodes per instance, whereas dispersion refers to the overall number of affected instances. Anomalies that affect less than 5 instances are low dispersion anomalies, and anomalies that affect more than 5 instances are high dispersion anomalies.

### 6.5.1 Experimental Setup

We employ the same settings as in Chapter 4. That is, we set $\rho = 0.5$ and $\eta = \frac{1}{\sqrt{t}}$ for the one-class classifier, $\rho = 0$, $\eta = \frac{1}{\sqrt{t}}$ and $\lambda = 1$ for the binary classifier, and experiment with various values of $\gamma$ and $\nu$. Additionally, we employ $Tr = 3600$ (i.e., 1 hour), $Gr = 20$, $z = 128$, and the set of features that obtained the best detection results in Chapter 4: $\{CO, ME, B0, S1\}$. For simplicity, all instances use the same settings, meaning that results could be improved with a better parameter selection mechanism that takes into account the characteristics of each instance, or that modifies certain parameters at run time converging to the optimal values.

We run our experiments on the Phoenix supercomputer [113], where we deploy 200 instances (i.e., combination of *Behavior Extractor*, *Behavior Identifier*, and *Feedback Provider*). To simplify the experiments, all instances run in the same node using 32 cores and 16GB of memory. Since the dataset is organized in 1,000 nodes, each instance monitors 5 nodes (i.e., 60 performance metrics). Feedback is given automatically to every instance after an alert, and while the instance is in the *Alert* state. The *Feedback Aggregator* consists in a simple "OR", that is, alerts from any instance are always forwarded to the system administrator.

As said before, our dataset contains 120 runs with varying number of nodes affected by the anomalies. In Table 6.1, 20% density means 1 anomalous node out of 5, 40% means 2 anomalous nodes, and so on. The distribution of anomalous nodes among the different instances is completely random. That is, each of the 10 anomalies per run affects different nodes in different instances, and only the density remains constant.

We employ the same evaluation metrics as in Chapter 4. That is, *Recall* and *Precision* defined as

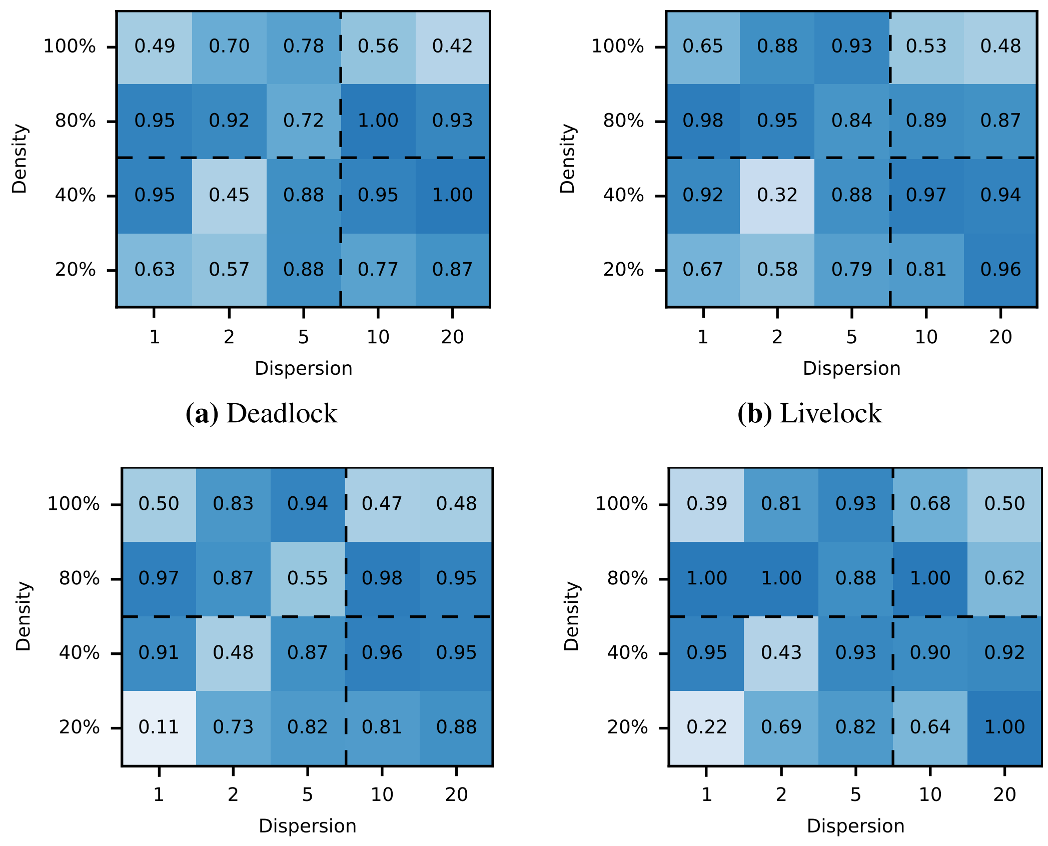
$$\begin{aligned} Recall &= \frac{\text{detected anomalies}}{\text{total number of anomalies}} \\ Precision &= \frac{\text{detected anomalies}}{\text{total number of alerts}}, \end{aligned} \tag{6.2}$$

### 6.5.2 Results

Fig. 6.3 and Fig. 6.4 show mean *Recall* and *Precision* for the different profiles. Data is presented according dispersion and density. *Recall* is presented per anomaly type and *Precision* is computed globally because false alerts cannot be assigned to a particular anomaly. Dashed lines in the plots highlight the boundaries of the profiles outlined in Table 6.1.

Fig. 6.3 shows that the detection rate is not heavily influenced by density nor dispersion. All anomaly types are detected with high accuracy in all profiles in general. Nevertheless, there are some appreciable tendencies, such that the detection accuracy increases with dispersion. For example, all anomaly types are detected more easily when they affect one node in different instances (20 - 20% case) than when they affect one node in one instance (1 - 20% case). This is because the probability of one of the instances detecting the anomaly is greater as the number of affected instances increases.

This has interesting implications, in that it allows large-scale systems to be divided in small subsystems, and thus make the system easier to analyze in terms of computational requirements, because detection performance is not negatively affected by high dispersion. Thus, the partitioning can be performed without taking into account component coupling and still obtain good detection results. The only case in which small subsystems could be detrimental to the detection accuracy is for anomalies characterized exclusively by an anomalous correlation between several nodes that



**(a)** Deadlock **(b)** Livelock

**(c)** Synchronization **(d)** Memory Leak

**Fig. 6.3.** Mean *Recall* obtained with the different profiles.

are monitored by different instances. However, this can be avoided with appropriate partitioning.

Fig. 6.4 shows that *Precision* clearly increases with density, and decreases slightly with dispersion. This can be explained by a higher number of alerts when more instances are affected by the anomalies, and a much accurate detection in high density scenarios because the anomaly has a higher impact on the instance. The number of false alerts can be reduced with a more sophisticated *Feedback Aggregator* in high dispersion scenarios. For example, a simple yet effective technique is to notify the system administrator only if more than one instance raises an alert in a short period of time.

Fig. 6.5 and Fig. 6.6 show *Recall* and *Precision* when the system administrator is notified only if 3 alerts occur within a 50 seconds window. We see that both

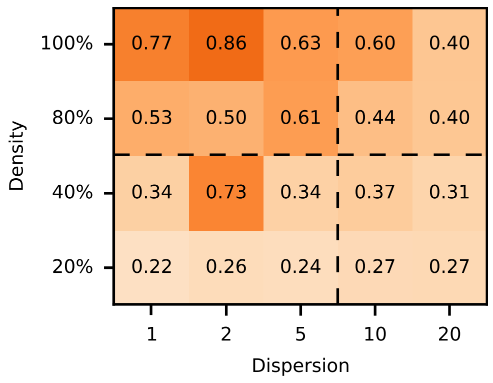


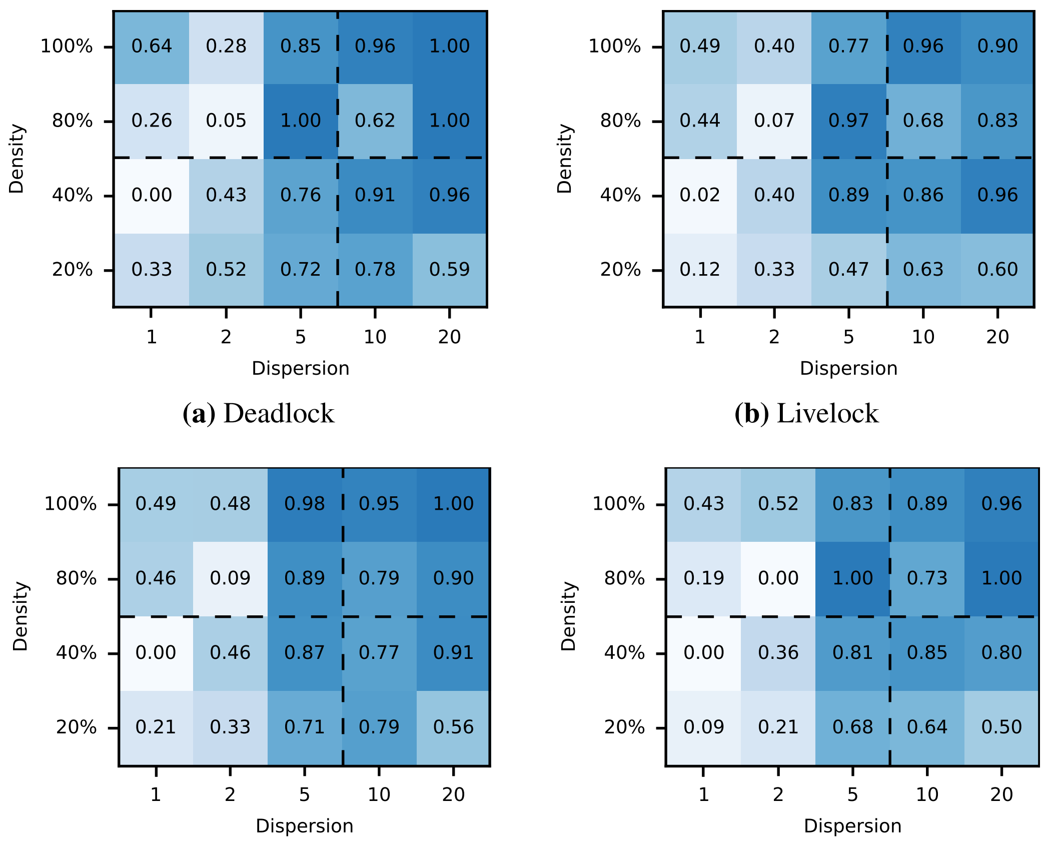
**Fig. 6.4.** Mean *Precision* obtained with the different profiles.

*Precision* and *Recall* improve in high dispersion profiles. However, overall accuracy is reduced in low dispersion cases because the probability of 3 consecutive alerts is lower. Nevertheless, our framework can be easily tailored to the characteristics of the system under study to maximize performance.

### 6.5.3 Discussion

Without aggregating alerts (Fig. 6.3 and Fig. 6.4) mean results are 0.77 *Recall* and 0.45 *Precision*. All anomaly types are detected in most situations, however, when the number of anomalous nodes per instance is less than 20%, detection is achieved with low *Precision* (i.e., a high number of false alarms). This is because one anomalous node per instance means between 1.6% (1 out of 60) and 3.3% (2 out of 60) anomalous time series per instance depending on the anomaly type. At these rates, our framework is unable to differentiate between normal and anomalous behavior.

Table 6.2 summarizes the results obtained for the different profiles. It is clear that our framework performs better in high density scenarios, where mean *Recall* and *Precision* are 0.76 and 0.55 respectively. High density scenarios are scenarios that contain 13% to 17% of anomalous time series per instance. Therefore, the optimal partition size depends on the system under study. In large-scale systems where tightly coupled components can be easily identified (e.g., web server and database), the instances can be set to monitor a higher number of components, as the probability of



**(a)** Deadlock **(b)** Livelock

**(c)** Synchronization **(d)** Memory Leak

**Fig. 6.5.** Mean *Recall* when requiring three consecutive alerts to notify the system administrator.

anomalies affecting more than 13% of the metrics per instance is higher. In systems where anomalies are expected to have a highly distributed impact affecting fewer metrics in many components, instances should be set to monitor fewer components. Another possibility is to set different instances to monitor only one or two metrics (like memory usage) in multiple components. This can be easily done thanks to our framework's flexibility, and could provide better results in scenarios where anomalies have very low impact on the system metrics.

Nevertheless, the results are impressive considering that in high density scenarios the percentage of overall affected nodes is between 0.4% and 10%, and the percentage of affected time series across the whole system is between 0.03% and 1.6%. This is achieved thanks to our framework's decentralized setting.

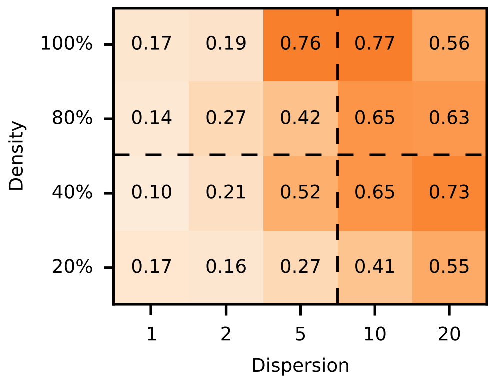


**Fig. 6.6.** Mean *Precision* when requiring three consecutive alerts to notify the system administrator.

**Table 6.2.** Summary of the results obtained for the different profiles.

| Profile | *Recall* | *Precision* |
|---|---|---|
| High Density High Dispersion | 0.71 | 0.46 |
| High Density Low Dispersion | 0.81 | **0.69** |
| Low Density High Dispersion | **0.89** | 0.30 |
| Low Density Low Dispersion | 0.69 | 0.35 |

Since detection rates are high in all cases, the main limitation of our framework is the amount of false positives. This can be solved with better feedback aggregation strategies, as shown in Fig.fig5.6, where mean *Precision* for high dispersion scenarios is 0.61.

Regarding the scalability of our approach, 200 instances running in a node with 32 cores and 16GB of memory are able to process 10 hours of execution from a system composed of 1,000 nodes (i.e., 60,000 time series of 36,000 data points) in around 15 minutes. This is around 1.5 seconds per minute of execution with the 0.8% of the overall computational resources (32 out of 4,000 monitored cores). The overall impact on the system is insignificant considering that a completely decentralized deployment with each instance running in a different node can be employed, and the only required communication is between the instances and the *Feedback Aggregator* when alerts are raised.

## 6.6 Summary

In this chapter we have presented the problem of detecting anomalies in large-scale distributed systems and have proposed a decentralized approach to address the main challenges behind this problem. In our experiments, we evaluated the detection capabilities of the framework presented in Chapter 3 in large-scale scenarios using a synthetic dataset composed of 120 10-hour long runs with 1,000 nodes displaying various complex anomalous behaviors.

Our results show that our framework can be applied to large-scale systems with great success using a decentralized setting. Specifically, our framework is capable of detecting several types of anomalies with overall mean *Recall* of 0.78 and mean *Precision* of 0.45. Moreover, in high density scenarios, our framework achieves mean *Recall* of 0.76 and mean *Precision* of 0.55, with only between 0.03% and 1.6% of anomalous data across the whole system. Additionally, our framework's decentralized setting is highly scalable and consumes less than 0.8% of the available computational resources, meaning that it can be employed in large-scale systems with minimal impact.

These results improve the state-of-the-art in anomaly detection in large-scale distributed systems, as our framework provides a highly scalable and flexible method that can be applied to numerous scenarios without historical data, and in an adaptive manner. Moreover, in contrast to some existing works [34, 35], we do not assume behavioral homogeneity among nodes, and experiment with a much larger system that generates 60,000 readings of monitoring data per unit time.

# Chapter 7

# An Online Approach for the Detection of Novel Botnets

As seen in Chapter 2, anomaly detection has several application domains, including malicious traffic detection. Moreover, malicious traffic detection suffers from similar challenges to anomaly detection in other domains studied in Chapters 4, 5 and 6. This means that we can take advantage of the characteristics of the framework presented in Chapter 3 to tackle the malicious traffic detection problem, while overcoming many of the limitations in existing works.

A particular case of malicious traffic is botnet traffic. Botnets are networks of illegally controlled computers [114] that have become a major threat on the Internet [27]. These networks can generate millions of dollars for their owners [115] by disrupting the normal activities of legitimate Internet users or by stealing sensitive information.

Detecting malicious traffic in order to disrupt botnet activities is essential for the reliability, availability and security of Internet services. However, despite many efforts in this direction [27], key challenges remain. These include the high computational requirements of processing large amounts of information, the similarity between botnet and normal traffic, and the constant creation of new botnet mechanisms to bypass current detection approaches.

In this chapter we analyze the problem of detecting bots in large-scale networks, and propose the use of the framework presented in Chapter 3 to tackle this problem. Employing a scalable distributed deployment, our framework creates a complete characterization of the behavior of legitimate hosts that can be used to discover botnet traffic with high accuracy. Moreover, our framework dynamically adapts to changes in network traffic, and is capable of detecting novel botnets without any previous knowledge about them and without any assumption on their architecture or protocols employed. These features are crucial to nullify the constant efforts by botnet managers to adapt to detection techniques.

## 7.1 Overview

Botnets are networks of illegally controlled computers (i.e., *bots*) typically employed for malicious activities. These networks are created by infecting a large number of computers with *malware* (i.e., malicious software) by means of operating system vulnerabilities, USB drives, or malicious web sites. Once a victim's computer is infected, the botnet software allows an attacker (also known as *bot master*) to take control and carry out malicious activities such as e-mail spamming or distributed denial-of-service (DDoS) attacks. Upon infection, and to remain undetected, bots typically update themselves, disable antivirus applications, block DNS lookups to certain domains, and download and run other types of malware.

Since the appearance of the first Internet Relay Chat (IRC) [69] based bot in 1993 [116], botnets have become a dangerous threat difficult to detect and dismantle. A novel malware used to create large botnets of small devices called Mirai [117] is believed to be behind a recent massive DDoS attack that disrupted access to tens of services in October 2016 [118]. This attack, which employed around 100,000 Internet of Things (IoT) devices, could be the largest DDoS attack in history, with an estimated throughput of 1.2 Tbps. Apart from the economic losses that downtime causes in industry [75], botnets can cause numerous issues in other sectors, such as defense [119] or public administrations [120].

The damaging potential of botnets has led to significant efforts to detect and prevent them [27]. Existing methods for botnet detection can be classified into two groups: signature-based and anomaly-based [27]. Signature-based techniques employ signatures of well-known botnets to detect malicious hosts [121]. Anomaly-based techniques employ machine learning and statistical methods to detect unusual or suspicious patterns in computers' behavior. Anomaly-based techniques have become the main research area in botnet detection [27] due to the limitations of signature-based methods, such as the difficulty in keeping the signature database up to date, and their inability to detect previously unseen botnets.

Anomaly-based methods can be further divided into host-based and network-based [27]. Host-based methods analyze the internal behavior of computers looking for signs of infection in binaries and system calls [122], whereas network-based methods analyze network traffic looking for suspicious patterns [31, 123] or correlations between the behavior of multiple hosts [71, 124, 37, 72] that can be indicative of botnets.

Since malicious activities can be classified as a particular type of error in computer systems [16], network-based botnet detection methods are a particular case of anomaly detection in distributed systems. In this case, anomalies are malicious behaviors, and system metrics are obtained from traffic monitoring rather than hardware counters.

Similar to anomaly detection in distributed systems, efficient and accurate botnet detection remains open for improvement despite the numerous efforts taken towards this direction. The main challenges in the area are [27]: the continuous evolution in botnet creation, maintenance and communication mechanisms; the fact that botnet traffic is very similar to regular traffic in many cases; and the high computational resources required to analyze large amounts of information. In addition, the majority of Internet users are not aware of the risks of botnets or how to protect their devices [27] and, due to the spread of mobile and IoT devices, the number of potential victims has grown exponentially in the last years [125, 126].

Most network-based botnet detection approaches employ machine learning and statistical techniques to differentiate normal from malicious traffic [27]. Since most of these approaches use historical bot data to build their statistical models and classifiers, they are limited to the detection of well-known botnets, or new botnets that display the same well-known behavioral patterns. However, attackers are constantly adapting their methods and implementing new mechanisms to bypass state-of-the-art detection approaches in a kind of arms race [127]. Thus, assuming that future botnets can be detected by analyzing historical data is a critical false assumption, one that will always keep researchers one step behind of attackers.

In this chapter we employ the framework presented in Chapter 3 to detect novel bots in large-scale networks. We show how our framework can be deployed in such scenarios, and analyze its detection performance with a varied and realistic botnet dataset. The rest of this chapter is organized as follows: Section 7.2 describes the characteristics of botnets. Section 7.3 presents the different botnet behavioral patterns that can be employed to detect them. Section 7.4 describes our approach. Section 7.5 presents the experimental analysis, and Section 7.6 concludes the chapter.

## 7.2 Botnet Characteristics

Botnets are composed of three main entities: bot master, command and control infrastructure (C&C), and bots [128]. Fig. 7.1 shows the most typical botnet architecture. Bots are infected computers that can be controlled through some kind of C&C infrastructure by a bot master. Since all communication between bot master and bots goes through the C&C infrastructure, this is the most vulnerable part of the botnet.

The C&C infrastructures can be centralized or decentralized [128]. Centralized C&C infrastructures are composed of one or a small number of servers that transmit

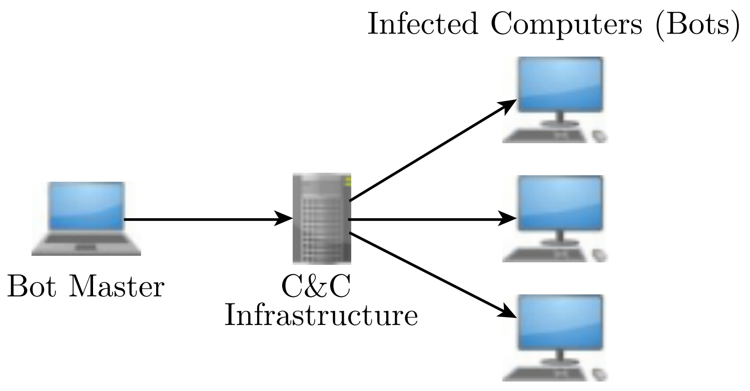


**Fig. 7.1.** Typical botnet architecture.

commands to bots. These infrastructures typically employ the IRC protocol due to its versatility, or the less suspicious Hypertext Transfer Protocol (HTTP) [36] as means of communication. The weaknesses of centralized infrastructures is that they present a single point of failure and that they generate certain traffic patterns that can be easily detected. Decentralized C&C infrastructures employ P2P networks to distribute messages between bots. Modern botnets have moved to this kind of C&C infrastructure [129] due to its robustness and obfuscation capabilities. Decentralized botnets can employ already existing P2P networks, such as Kademlia [130], or build their own P2P network [131].

### 7.2.1 Bot Life Cycle

Despite the various types of botnets, the process for infecting vulnerable computers and transforming them into bots is typically the same [27]. This life cycle is depicted in Fig. 7.2. The process begins with a virus or malware infection through the usual channels (e.g., operating system vulnerabilities, malicious web sites, infected external devices, etc.). This is the *Initial Infection* phase. After a host has been infected, the malware downloads and runs other malicious binaries from a remote database in a *Secondary Injection* phase. These binaries, when executed, convert the host into a bot. In a third phase, the recently activated bot contacts the C&C infrastructure to inform the bot master of its presence. This *Rally* phase is repeated every time that the bot is restarted or reconnects to the network. After this third phase, the bot is ready to receive orders from the C&C, and carries out malicious activities in phase 4. Finally, in a fifth phase, bots can receive maintenance or updates to avoid detection, include new features, or switch to another C&C server.

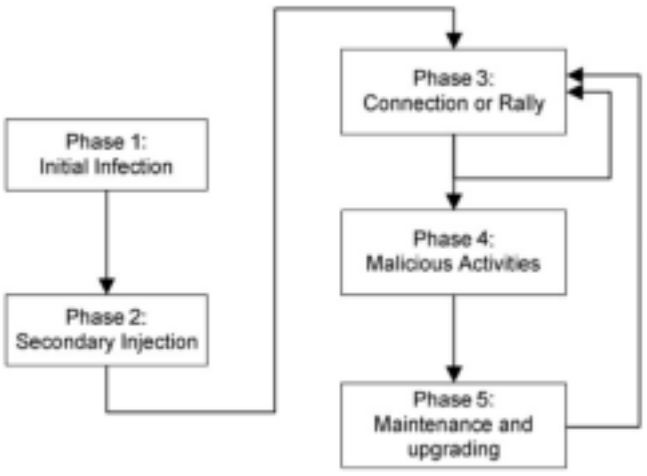


**Fig. 7.2.** Bot life cycle (extracted from Silva et al. [27]).

Phases 3 and 5 are the most vulnerable stages the life cycle of a bot. In these two phases, bots display certain communication patterns that can be recognized by botnet detection systems. Moreover, these phases are likely to occur periodically, thus increasing the chances for detection.

### 7.2.2 Command & Control Communication

Since bots need to communicate with the C&C infrastructure to receive updates and orders, the C&C is the central and most critical part of any botnet. The way bots obtain the location of the C&C varies across different botnets with different C&C architectures, and has evolved over time to minimize detection risks.

In centralized architectures, the simplest approach is to *hardcode* the C&C IP address or domain name in the bot binaries. Hardcoding the IP address has the disadvantages that control over bots is lost if the C&C is taken down, and that authorities can obtain the C&C IP address using reverse engineering. Hardcoding the domain name of the C&C is a better approach, as Dynamic DNS (DDNS) [132] can be employed to change the IP address associated with a domain name if the old address is compromised. However, the botnet still can be easily dismantled by blocking the malicious domain at the DNS provider or the organizational level.

A more sophisticated mechanism typically employed in centralized C&C architectures is a Domain Generation Algorithm (DGA). Bots execute a DGA able to generate thousands of pseudorandom domains per day, and try to connect to a portion of them. The bot master, who also has access to the DGA, can register a fraction of the domains to ensure that bots will access the C&C with certain probability. In this manner, by the time authorities take down a malicious domain, bots have already moved to new ones. This is the mechanism employed by Conficker [133], a prominent botnet that infected between 9 million to 15 million computers, including systems at the United Kingdom Ministry of Defence, the French Navy, and Manchester's City Council and police department.

In decentralized architectures, bots do not typically employ the DNS protocol [134]. Instead, bots receive a list of peers upon infection, and then obtain a more up to date list from these initial peers. The initial peer list can be downloaded in a separate file, or hardcoded in the bot binaries [129, 134]. Other P2P bots periodically send probes to random IP addresses in order to discover active peers in the network [135]. Decentralized botnets can be harder to detect because they do not exhibit certain DNS communication patterns that are easy to recognize by some detection methods.

### 7.2.3 Malicious Activities

The purpose of any botnet is to obtain some kind of gain, whether economic, industrial or political. This gain is obtained by means of a series of malicious or illegal activities. The most typical of these activities are [136]:

- **Malware propagation:** many bots try to spread themselves to other vulnerable computers in the network in order to increase the size and power of their botnet. Bots achieve this by performing port scans and probing other computers looking for operating system or application vulnerabilities.

- **Cyber-fraud:** some bots can steal banking information and passwords by displaying fake versions of certain web sites when the infected computer tries to connect to them [137]. The stolen information is then sent to the C&C.

- **Spamming:** the spread of junk e-mails including advertising, *phishing*, and malware is a common practice that can provide significant profits to the bot master [115].

- **Information theft:** some bots steal sensitive information such as credit card numbers or passwords and transmit them to the C&C. Other botnets have been used for industrial and cyber-espionage [138].

- **Network disruption:** bots can be ordered to send numerous requests to a service in a short period of time to make it unavailable to other users. DDoS attacks are used for economic, political or industrial reasons.

## 7.3 Botnet Detection

Some of the botnets behavioral patterns outlined in Section 7.2 can be exploited to differentiate bot traffic from legitimate traffic in a network. The particular communication patterns that bots exhibit when performing malicious activities and communicating with the C&C can be recognized when looking at the protocol employed, the number of packets exchanged, or the inter-arrival time of packets among others. The behavioral patterns most commonly exploited by existing network-based detection approaches are detailed in the following.

### 7.3.1 Communication using IRC

IRC is a messaging protocol that is very convenient for C&C communication, as it provides one-to-one and one-to-many communication capabilities. Thus, many botnets have historically employed this protocol as means of communication and, as a response to this, many detection approaches analyze the characteristics of IRC traffic [139, 140]. These approaches look for randomly generated IRC nicknames [140], or IRC channels (i.e., chat rooms) with a majority of participants performing certain activities such as TCP SYN scanning [139]. As a response to IRC based detection efforts, modern botnets have moved to other protocols such as HTTP(S) and P2P.

### 7.3.2 DNS Lookups

As detailed in Section 7.2, C&C communication in centralized botnet architectures requires an extensive use of the DNS protocol to make botnets more robust and difficult to dismantle. Bots generate large amounts of DNS queries because C&C resource records are typically delivered with a short time-to-live to allow for fast transitions from one IP address to another, and because DGA algorithms typically try to connect to large amounts of nonexistent domains.

It has been reported that a high number of `NXDOMAIN` responses (i.e., trying to resolve many nonexistent domain names) can be indicative of an infected computer [141, 142]. In addition, botnet DNS queries differ from legitimate DNS queries in that the number of different IP addresses that access a C&C domain is fixed because malicious domains are only accessed by bots; C&C domains are accessed by many computers simultaneously in a short period of time; and C&C communication typically employs DDNS instead of regular DNS [143]. Other works have exploited the fact that C&C domain names are automatically generated and follow certain patterns [144].

### 7.3.3 P2P

Since P2P botnets do not use the DNS protocol [134], detecting decentralized C&C infrastructures can be more difficult than detecting centralized ones. Nevertheless, malicious P2P traffic can be differentiated from legitimate traffic by analyzing the volume of the P2P transfers, the amount of connections and disconnections in the P2P network (i.e., *churn*), and the connection similarities between peers [72, 145]. More precisely, data transfers are typically less voluminous among bots than among legitimate P2P users, P2P botnets are more stable than legitimate P2P networks, and peers in a botnet display similar traffic patterns that do not exist among legitimate P2P users.

Other methods to differentiate between botnet and legitimate traffic in P2P networks include the analysis of the management packets employed to maintain the network structure, such as keep-alive packets [146], and the analysis of the periodicity of packets and the number of IP addresses contacted by each peer [147].

### 7.3.4 SMTP

Spamming is an activity that can generate millions of dollars to bot masters [115]. Bots employed for this purpose make an extensive use of the Simple Mail Transfer Protocol (SMTP), making them vulnerable to botnet detection systems. Spammer bots can be identified by analyzing the content length, time of arrival, and frequency of e-mails [148].

### 7.3.5 Group Activities

Regardless of the C&C architecture and protocols employed, bots can be detected by finding computers with similar behaviors in a network [37, 68, 71, 143, 149]. For example, computers that display similar communication patterns such as average number of bytes per packet and, at the same time, perform suspicious activities such as high DNS requests, are likely to be bots [71]. A limitation of looking for group activities is that it requires the presence of more than one bot of the same botnet in the monitored network.

### 7.3.6 Transport Layer

Finally, bots can also be detected by analyzing network traffic at the transport layer (i.e., TCP and UDP) regardless of the application layer protocol or the C&C architecture employed. The behavior of infected computers differs from the behavior of normal computers in that bots need to perform certain actions as part of their life cycle, such as receiving updates, maintaining open connections, and performing malicious activities. These actions create certain traffic patterns that can be identified by analyzing sent and received TCP and UDP packets. For example, an abnormally high amount of outgoing small packets can be a sign of some kind of port scanning that many bots employ to find vulnerabilities in other computers.

Network traffic is typically arranged in *flows* to facilitate this kind of analysis. A network flow is a collection of packets that share the same source and destination IP, source and destination ports, and protocol. Flows can then be classified according to their number of packets, their duration, or their average packet size to detect malicious activities [123, 150, 151]. It has been suggested that the most useful features to differentiate between normal and malicious flows are average packet length, flow duration, average bits per second, and percentage of small packets exchanged [1].

## 7.4 Detection of Novel Botnets

As previously stated, network-based botnet detection is a particular case of anomaly detection in distributed systems. Thus, we can employ the framework presented in Chapter 3 to tackle this problem. Moreover, due to the characteristics of our framework, we can use it to detect novel botnets for which historical data is unavailable. This is particularly relevant in botnet detection, as bot masters are continuously implementing new botnet communication mechanisms to avoid being discovered.

A monitored network can be composed of hundreds of computers that exchange information between them and with the outside. Each of these computers can be regarded as a component in a distributed system, and their network activities can be seen as the system performance metrics. For example, monitoring the number of packets sent and received by each computer over a time period yields $2n$ time series for $n$ computers. We then can apply the procedure described in Chapter 3 to obtain a feature vector that represents the behavior of the network (or system) in that particular time frame. In this manner, we can model the normal behavior of the network, and label as anomalous (or malicious) any deviation from this model.

However, this methodology has two disadvantages. On the one hand, many real world networks are large-scale, and analyzing high dimensional data is problematic as explained in Chapter 6 due to the curse of dimensionality. On the other hand, most networks are highly dynamic, where computers continuously join and leave. This makes it impossible to compare behavior representations obtained at different time periods as the number of available performance metrics (i.e., $n$) changes over time. The dimensionality problem can be addressed by employing a system of systems approach as showed in Chapter 6. However, coping with network dynamism requires a different methodology.

We propose to model the behavior of hosts instead of the behavior of the whole network to tackle both the data dimensionality and the network dynamism problems in large-scale networks. Thus, we can employ the replicated behavior setting introduced in Chapter 3, which can be seen in Fig. 7.3. In this setting, we deploy a *Behavior Extractor* per computer in the monitored network (labeled as *system component* in Fig. 7.3). Each of these *Behavior Extractors* communicates with a *Behavior Identifier* that maintains a unique model of normal host behavior in the network. The *Behavior*

*Identifier* transmits the classifications to the appropriate *Feedback Provider*, which can then decide whether to raise an alert based on each computer's history. In this manner, the *Behavioral Model* characterizes any legitimate behavior that takes place in the network, such as web browsing or video streaming, and labels as anomalous hosts that deviate from these.

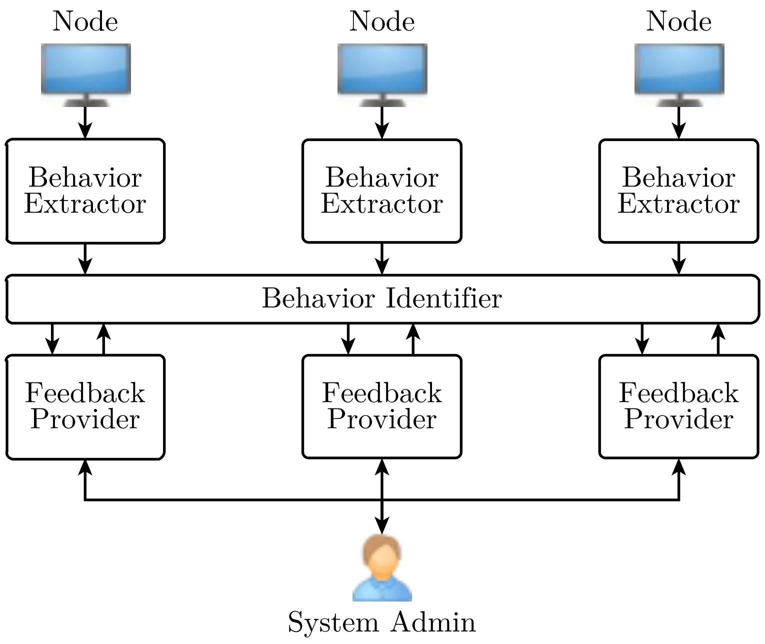


**Fig. 7.3.** Setting employed to analyze distributed systems from a replicated behavior perspective.

This setting solves the data dimensionality problem by employing feature vectors that represent the behavior of a single host instead of the behavior of the whole network. Moreover, since the *Behavior Identifier* acts as a black box from the point of view of the other framework components (see Chapter 3), the *Behavioral Model* is independent from the number of computers in the network. That is, *Behavior Extractor* and *Feedback Provider* instances can be created and destroyed as computers join and leave the network. This solves the network dynamism problem.

It is also worth noting that our framework is orthogonal to the behavioral representation employed. That is, our framework can be employed to analyze protocol specific behaviors such as DNS or SMTP usage, or to analyze protocol independent behaviors such as transport layer network flows.

## 7.4.1 Behavior Representation

Network traffic typically consists of TCP and UDP packets, and in many cases only the packet headers are available due to privacy requirements. As said before, a common method for analyzing network traffic is to group packets in flows. Flows are collections of packets with the same source and destination IP addresses and ports, and protocol. In this manner, network behavior is modeled from the point of view of the communications that take place, and malicious computers are detected by the characteristics of the flows in which they are involved.

We take a slightly different approach since we are interested in modeling the behavior of computers instead of the behavior of the network. Thus, we compute metrics by looking at the last $N$ packets sent and received by each IP address. For example, if we are interested in detecting DGA based bots, we can use the number of `NXDOMAIN` responses in the last $N$ packets received by each IP address as a metric. The metric is updated with new packet arrivals because we look at the last $N$ packets received. Even though the inter-arrival time of packets is not constant, we can represent the evolution of the metric over time as a time series that can be processed by the *Behavior Extractor*.

Formally, given an IP address $A$, we define $P_j(A,N) = \{p_j, ..., p_{j+N-1}\}$ as the sequence of $N$ packets (in the order they are obtained) that have $A$ as source or destination IP, starting with the $jth$ packet. Then, we define $\{m_i \,|\, 1 \leq i \leq n\}$ as a set of $n$ metrics where $m$ are functions over a set of packets $m_i(P_j) = r_{ij}$, and $r_{ij} \in \mathbb{R}$. We then define $s_{ij} = r_{ij}, r_{i(j+1)}, ..., r_{i(j+z)}$ as the sequence of $z$ readings for metric $m_i$ starting with $r_{ij}$ (i.e., a time series). Thus, the collection of $s_{ij}$ for all metrics can be used to build *Behavior Instances* for each IP address in the network as explained in Chapter 3.

We employ the following 9 metrics to represent host behavior:

- The number of distinct destination IP addresses
- The number of distinct source IP addresses
- The number of distinct destination ports
- The number of distinct source ports
- Total bytes sent
- Total bytes received
- Packets sent
- Packets received
- The time difference between the first and the last packet

## 7.5 Experimental Analysis

In this section, we present a series of experiments to evaluate our framework's ability to detect novel bots in a network. More precisely, we are interested in answering the following questions: i) what is our framework performance in detecting novel bots without using historical bot data?, ii) how do different framework settings affect this performance? (i.e., $Tr$, $Gr$, and $z$), and iii) how many false alarms are generated?

### 7.5.1 Dataset

For our experiments we employ the ISCX botnet dataset [1], a publicly available dataset that consists of a combination of three other datasets, and contains traffic from

16 different IRC, P2P and HTTP based botnets. This makes the ISCX dataset the most realistic and varied dataset that is publicly available [1].

The ISCX dataset consists of two subsets: train and test. The train subset contains $\sim$9 million packets of which $\sim$4.5 million are malicious (i.e., have a malicious IP address as source or destination). The test subset contains $\sim$5 million packets of which $\sim$2.5 million are malicious.

Beigi et al. [1] perform a feature selection process employing the ISCX dataset, and find that the best features for detecting bots are the average packet length, the flow duration, the average bits per second, and the percentage of small packets exchanged. Using these features, Beigi et al. obtain a 75% detection rate and a 2.3% false positive rate. Beigi et al.'s approach is architecture and protocol agnostic, however, it employs historical botnet data, which means that it cannot detect novel botnets.

### 7.5.2 Experimental Setup

We employ the same settings as in Chapter 4. That is, we use $\rho = 0.5$ and $\eta = \frac{1}{\sqrt{t}}$ for the one-class classifier, $\rho = 0$, $\eta = \frac{1}{\sqrt{t}}$ and $\lambda = 1$ for the binary classifier, and experiment with various values of $\gamma$ and $\nu$.

Since we are interested in detecting novel botnets, we only use normal traffic to train our framework. The training set of the ISCX dataset contains $\sim$4.7 million normal packets. We train our framework using between the 5% and the 50% of the normal traffic in the training set, selected by random sampling, and analyze how different training sizes (i.e., the $Tr$ variable) affect our framework performance. We do not employ more than the 50% of the train set because this is enough for our framework to produce satisfactory results.

As said before, we employ 9 metrics to build the behavioral representation of the computers in the network: the number of destination IP addresses, the number of source IP addresses, the number of destination ports, the number of source ports, total bytes sent, total bytes received, total packets sent, total packets received, and the time difference between the first and the last packets. These metrics can be computed from the packet headers, thus, they can be employed even on encrypted traffic, while preserving the privacy of the users.

We compute these metrics on the last 25 packets per IP address (i.e., $N = 25$). This means that IP addresses that are send or receive less than 25 packets are filtered out. This is not a problem since bots usually generate more than 25 packets as part of their typical maintenance and malicious activities before potentially being detected and removed.

As explained in Section 7.3, we represent the collected metrics as time series that can be processed by the *Behavior Extractor*. To generate the feature vectors used in the classification process we employ the same features as in Chapter 6: $\{CO, ME, B0, S1\}$, and experiment with various values of $Gr$, $z$.

As in Chapter 4, feedback is given after every false alert, and we assume that computers do not switch from normal to malicious behavior, or vice versa.

### 7.5.3 Evaluation Metrics

We employ two metrics to evaluate our framework's performance: true positive rate ($TPR$) and false positive rate ($FPR$). Since our main focus is to detect bots in a network, and our framework monitors each IP address independently from each other (see Section 7.4), we define the evaluation metrics per IP address as

$$TPR = \frac{\text{detected bots}}{\text{total number of bots}}$$

$$FPR = \frac{\text{false alerts}}{\text{total number of legitimate computers}} \tag{7.1}$$

$TPR$ measures our framework ability to detect bots and $FPR$ measures the fraction of legitimate computers that are labeled as malicious. The optimal $TPR$ and $FPR$ values are 1 and 0 respectively.

### 7.5.4 Results

In a first experiment, we evaluate the effect of $z$ (i.e., window size) on our framework's performance. Fig. 7.4 shows $TPR$ and $FPR$ obtained for different window sizes when using $Gr = 1$ and $Tr = 10\%$ of the training set. Window size does not significantly affect our framework's performance in this case. This can be because bots display a consistent behavior over time that is significantly different from normal behaviors. The best results are $TPR = 1.00$ and $FPR = 0.082$, and are obtained with $z = 64$.

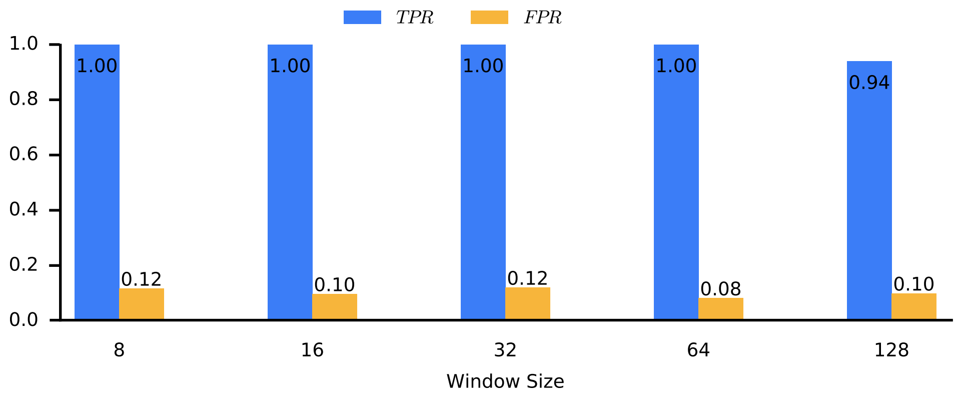


**Fig. 7.4.** $TPR$ and $FPR$ for different window sizes (with $Gr = 1$ and $Tr = 10\%$ of the training set).

In a second experiment, we evaluate how $Gr$ affects our framework's performance. Fig. 7.5 shows $TPR$ and $FPR$ obtained when using different $Gr$ values, $z = 64$, and $Tr = 10\%$ of the training set. Again, we see that $Gr$ does not have a significant impact on performance in this case. This can be because of the same reason as before. Bot behavior is consistent over time, and significantly different from normal behavior. Thus, malicious traffic is classified correctly most of the time, and alarms are generated

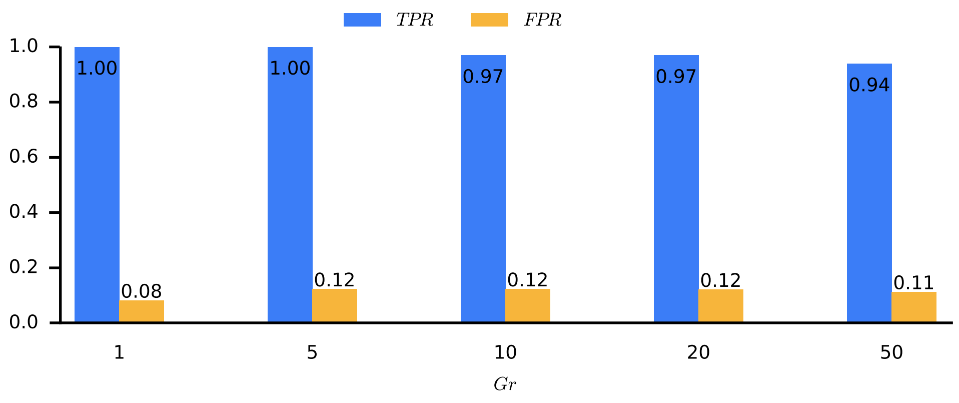


**Fig. 7.5.** $TPR$ and $FPR$ for different $Gr$ values (with $z = 64$ and $Tr = 10\%$ of the training set).

even when requiring 50 consecutive anomalous classifications. The best results in this case are obtained with $Gr = 1$.

In a third experiment, we evaluate the effect of the training size on the detection accuracy. Fig. 7.6 shows mean $TPR$ and $FPR$ for different $Tr$ values. The results are averaged over all $z$ and $Gr$ values. As the train size increases, the number of false positives decreases. This is because our framework receives a better representation of the behavior of legitimate computers, and can recognize them better in the testing stage. However, the decrease in false positives also results in a decrease in detection accuracy because the representation of normal behavior does not generalize so well, that is, the model *overfits* the training data. Using 50% of the training set, our framework obtains $TPR = 0.72$ and $FPR = 0.027$, and the best results are obtained when using $10\%$ of the training set with $TPR = 0.97$ and $FPR = 0.11$.

Finally, in many real world scenarios, a low number of false alerts is more important than a high detection rate. When a large number of false positives is generated, only a fraction of them can be investigated, as this is a time consuming task. As a consequence, true positives become irrelevant because they cannot be confirmed either. In these scenarios, it can be more beneficial to reduce the number of false positives to a manageable amount at the cost of reducing detection accuracy as well.

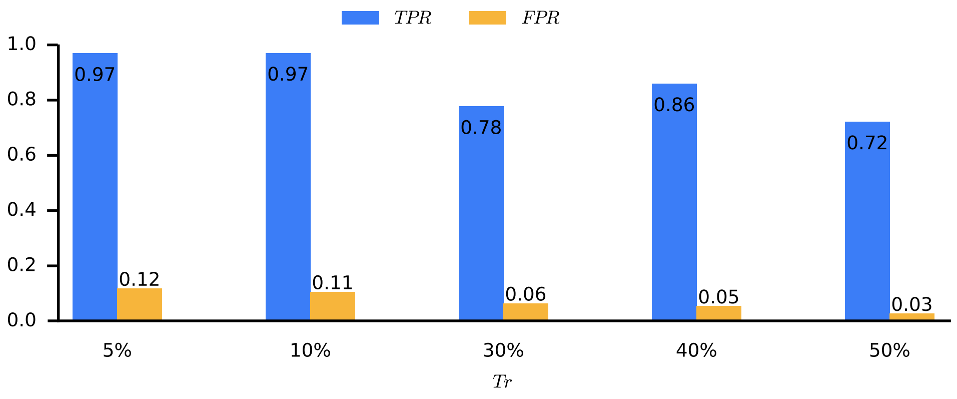


**Fig. 7.6.** Mean $TPR$ and $FPR$ for different $Tr$ values.

Fig. 7.7 presents the Receiver Operating Characteristic (ROC) curve generated by our framework. A ROC curve plots $TPR$ as a function of $FPR$, and shows how these two metrics change for different settings. The curve shows that our framework can be configured to minimize the number of false positives while sacrificing some detection accuracy in scenarios where a low number of false alerts is necessary. The lowest $FPR$ is 0.0017 with a $TPR$ of 0.48.

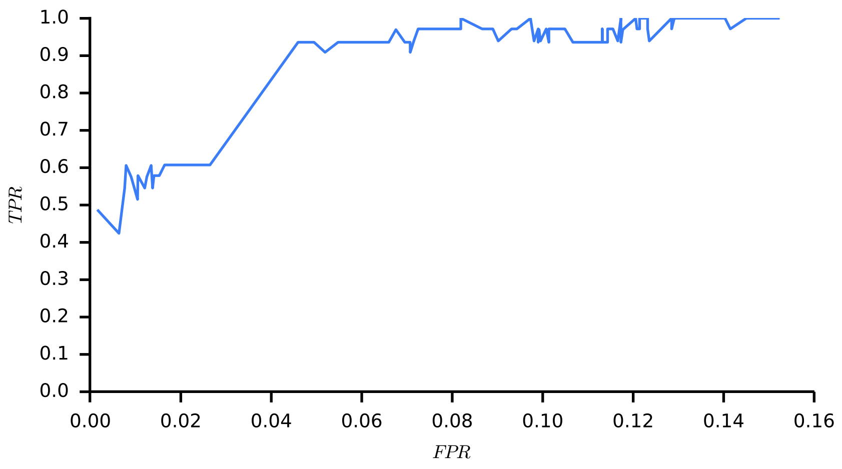


**Fig. 7.7.** ROC curve obtained for different framework settings.

### 7.5.5 Comparison with Existing Work (Beigi et al. [1])

Table 7.1 shows a comparison of our results with the results reported by Beigi et al. [1] when using the ISCX dataset. The first row of Table 7.1 shows our results when maximizing the overall accuracy (i.e., the mean of $TPR$ and $FPR$), and the second row shows our results when minimizing the number of false positives.

**Table 7.1.** Results comparison with Beigi et al.

| | $TPR$ | $FPR$ | False Alarms |
|---|---|---|---|
| Max Accuracy (Us) | **1.00** | 0.082 | 519 |
| Min $FPR$ (Us) | 0.48 | **0.0017** | **11** |
| Beigi et al. | 0.75 | 0.023 | $\sim$5,800 |

Beigi et al. compute the $TPR$ and $FPR$ in relation to the classification of network flows instead of IP addresses, thus, the results are not directly comparable. However, Table 7.1 shows that our approach can achieve a perfect detection rate with much less false alarms. The number of false alerts raised by our framework is 519 and 11 depending on the maximization criteria. Beigi et al.'s approach generates an approximate number of 5,800 false positives. Assuming that all the false positives are reported, this is one to two orders of magnitude above our number of false alerts. As said before, a low number of false positives is crucial in many real world scenarios, as investigating alerts is a time consuming task that requires expert knowledge. Moreover, our framework is able to achieve these results without historical data, being able to detect novel botnets exclusively based on a characterization of normal host behavior.

### 7.5.6 Discussion

Our framework can detect novel bots with a $TPR$ of 1.00 and a $FPR$ of 0.082 or, alternatively, with a $TPR$ of 0.48 and a $FPR$ of 0.0017. The latter means only 11 false alerts in a dataset with 5 million network packets and only 0.5% malicious computers.

That is, in scenarios where a low number of false positives is crucial, our framework can detect half of the infected computers with extremely low false alarm overhead.

Our framework maintains similar or better detection accuracy than similar works [1], while reducing the number of false alerts by one to two orders of magnitude. This is a significant improvement as a reduction of alerts is critical to reduce system administration costs, and to be able to investigate the notifications that correspond to real threats. Moreover, our approach does not rely on historical malicious data, can be used in a *plug-and-play* manner, and adapts to changes in network behavior by means of dynamic training. This means that our framework is able to detect novel botnets with communication mechanisms different from the methods employed by existing botnets. This is of the utmost importance since attackers are constantly adapting to avoid detection.

Despite the very promising results obtained, we identify some issues that should be addressed in the future. Even though the ISCX dataset is the most realistic and varied publicly available botnet dataset [1], our approach should be deployed in a real world scenario to fully evaluate its performance. More precisely, malicious traffic in the ISCX dataset represents the 44.97% of the total traffic, and hosts do not change their behavior from malicious to normal or vice versa. In a real scenario, the number of false positives generated by our approach could be higher due to the percentage of malicious traffic being lower, hosts exhibiting intermittent behavior, and IP addresses being reused by different hosts. Thus, in the future, we plan to deploy our framework in our campus network to further investigate its detection capabilities in real world scenarios, as well as to discover ways of minimizing the overhead produced by false alarms.

In addition, our approach relies on the fact that bot behavior is different from legitimate behavior from the point of view of the network traffic. Even though this is true for existing botnets, evasion mechanisms could be implemented in the future to mimic legitimate hosts and avoid detection. Nevertheless, bots need to deviate from normal behavior at some point in order to be useful to bot masters. Thus, despite possible evasion techniques, differences in behavior will likely be present at certain communication level that can be exploited by our framework by collecting the adequate metrics.

## 7.6 Summary

In this chapter we have presented the problem of detecting malicious activities in a network. This problem can be formulated as a particular case of detecting anomalies in a distributed system, meaning that it can be tackled by the framework described in Chapter 3. Thus, we have shown how our framework can be deployed using a replicated behavior setting to detect novel bots by analyzing network traffic.

Through an extensive experimental analysis using the most realistic and complete publicly available botnet dataset [1], we have proven the applicability of our framework in real world scenarios. Our framework is able to achieve a $TPR$ of 1.00 and a $FPR$ of 0.082 if a certain number of false alerts can be tolerated, or a $TPR$ of 0.48 and a $FPR$ 0.0017 if minimizing the false alarms is crucial.

Our results significantly improve the results obtained by other works on the same dataset [1], while introducing the advantages of an adaptive training and the ability to detect previously unseen malicious traffic. These two advantages are critical since attackers are constantly adapting their methods to avoid detection.

# Chapter 8

# Conclusions and Future Work

Distributed systems are pervasive and have become a critical building block in our society. The increasing size and complexity of these systems creates a need for better tools to achieve dependability. However, large-scale complex systems present new challenges for dependability and, more precisely, for fault tolerance and anomaly detection in the many domains where it can be applied.

In this thesis we have focused on anomaly detection in three domains: distributed systems, large-scale systems, and malicious traffic detection. As seen in Chapter 1 and 2, these three domains share key challenges and limitations. This has allowed us to tackle the anomaly detection problem in the three domains from a general approach that overcomes many of these limitations.

As mentioned in Chapter 1, the main contributions of this thesis are: i) a black box framework for anomaly detection in large-scale distributed systems, ii) a methodology for the identification of complex anomalous behaviors in distributed systems, iii) a decentralized architecture for anomaly detection in large-scale systems, and iv) a method for the detection of previously unseen malicious traffic in large-scale networks. In the following we detail these contributions and how the work in this thesis accomplishes them.

### A Black Box Framework for Anomaly Detection in Large-Scale Distributed Systems

Chapter 3 has presented our black box framework for anomaly detection. This framework has been designed to overcome the main limitations identified in existing works in the three domains explored in this thesis (see Chapter 2). Thus, our framework analyzes system behavior over time taking into account metric correlations, does not require historical failure data and thus can detect previously unseen anomalies, adapts to changes in the normal behavior of the system under study, can incorporate contextual information into the behavioral analysis, and is able to identify the type of known anomalies. Moreover, our framework provides a flexible and scalable design that makes it applicable to numerous scenarios regardless of the architecture of the system under study or the type of anomalies that are to be detected.

The suitability, scalability and high detection accuracy of our framework has been proven in Chapters 4, 5, 6, and 7, where the framework has been successfully employed to detect a wide variety of anomalies in different scenarios.

### A Methodology for the Identification of Complex Anomalous Behaviors in Distributed Systems

Complex anomalous behaviors are a major problem in distributed systems [19, 20]. However, few existing works have studied these behaviors in the literature [21, 35, 86]. In this thesis, we have provided insight on how some of these behaviors affect the system performance metrics (see Chapter 4), and have taken advantage of the outstanding capabilities of the framework presented in Chapter 3 to detect and identify the type of this behaviors with high accuracy, and in a way that overcomes many limitations in the area of anomaly detection in distributed systems.

The results obtained in Chapters 4 and 5 show that our approach is clearly superior in several ways to similar works in the area. Our framework improves the state-of-the-art in anomaly detection in distributed systems by achieving an increased detection accuracy, while providing certain characteristics that none of the existing works provide, as seen in Chapter 2. More precisely, our framework outperforms similar works by not relying on historical data, which is unavailable in most cases, adapting to changes in behavior, and not making assumptions on the characteristics of the anomalies.

**A Decentralized Architecture for Anomaly Detection in Large-Scale Systems**

Large-scale systems are pervasive and a central part of current society. However, few works address the anomaly detection problem in a particularly large scenario [30, 35, 39, 43, 66, 108, 109]. Moreover, as seen in Chapter 2, anomaly detection in large-scale systems suffers from similar challenges and limitations as anomaly detection in distributed systems. In Chapter 6, we have presented a decentralized approach to tackle the anomaly detection problem in large-scale systems in a way that overcomes many of these limitations.

Through an extensive experimental study, we have proven that the framework presented in Chapter 3 provides a highly scalable solution to anomaly detection in large-scale scenarios that is superior to similar works in the area. Similar to anomaly detection in distributed systems, existing works that address anomaly detection in large-scale systems often rely on historical failure data, provide architecture specific solutions, and do not analyze system behavior in a general manner. In contrast, we have provided an architecture agnostic solution that can detect previously unseen anomalies, identify their type, and that does not make assumptions on the characteristics of the anomalous behaviors. Moreover, we have evaluated our method in a much larger scenario than those employed by similar works, consisting of 1,000 nodes and 60,000 metric readings per unit time. This has proven that our framework can be employed in large-scale scenarios with minimal overhead.

**A Method for the Detection of Previously Unseen Malicious Traffic in Large-Scale Networks**

Botnets are one of the major existing threats in the Internet [27]. Their detection is crucial for our security, privacy, and safety as Internet users. However, botnet detection remains an open problem due to key challenges, such as the constant evolution of botnet communication mechanisms and the difficulties in differentiating botnet from legitimate traffic. In this thesis, we have provided a scalable solution to this problem that greatly improves the state-of-the-art in the area.

As seen in Chapter 2, existing botnet detection approaches are limited by the use of historical botnet data, the lack of adaptiveness, and by providing protocol and architecture specific methods. In Chapter 7, we have demonstrated how the framework described in Chapter 3 can be employed in a scalable manner to tackle the botnet detection problem with great success. In the experimental analysis presented in Chapter 7, we have seen that our framework can detect botnets with high accuracy, while producing a number of false alarms one to two orders of magnitude smaller than similar works in the area [1]. Moreover, our framework is capable of achieving this without relying on historical botnet data, which means that previously unseen botnets can also be detected. This makes our framework extremely robust to the constant botnet evolution, and provides a hard to evade solution that cannot easily become obsolete.

## 8.1 Directions for Future Work

The work presented in this thesis has contributed to various research areas where anomaly detection can be applied. The most evident continuation to this thesis would be to apply the ideas proposed in Chapter 3 to address the anomaly detection problem in other domains where our framework design can be beneficial, such as in sensor networks, the Internet of Things, IP networks, or multi-agent systems. Moreover, the work presented in this thesis is also applicable to other scientific fields, such as cognitive science for the analysis of electroencephalography data [152], or engineering for the diagnosis of industrial systems [153].

Another logical continuation to this thesis is the improvement of the machine learning techniques employed by the framework presented in Chapter 3. For example, the two-step classification process would greatly benefit from a mechanism to adjust the classifier hyperparameters in an online manner, or from other classification strategies, such as an ensemble of classifiers with different settings (e.g., window sizes). Other research directions that we will explore in the future are detailed in the following.

### Multilevel and Context-Aware Anomaly Detection for Large-Scale Virtualized Infrastructures

We identify three main challenges that need to be addressed in anomaly detection: i) large-scale scenarios, 2) multilevel anomalies, 3) contextual anomalies. These three challenges have not received the necessary attention in the literature, and are becoming more relevant as systems grow in size, and due to the popularity of virtualized environments.

Despite the contribution of this thesis to large-scale scenarios, there is still a significant gap between current research and real world deployments (see Chapter 5). In this thesis we have experimented with 1,000 nodes and 60,000 system metrics, and existing works that focus in large-scale systems experiment with systems of 250 to 800 nodes, and with 500 to 16,000 system metrics. However, real world large-scale systems are composed of 50,000 to 80,000 computers, which can easily result in over 1 million system metrics. Moreover, if we consider that each computer can allocate numerous virtual machines in current virtualized infrastructures, the number of available metrics rises to the order of $10^7$. Our framework is capable of addressing this challenge and thus, in the future, we should aim for high scalability as a priority, and should try to gradually approach the numbers exhibited by real world deployments.

In addition to this, due to the increasing popularity of virtualized environments, there is a strong need for multilevel and context-aware anomaly detection. In these environments, new virtual machines are constantly created, removed, and reallocated, and anomalies can occur at application, virtual machine, and physical machine level. Moreover, these anomalies cannot be characterized unless context in defined in a very precise manner. For example, high CPU and memory utilization can be normal or abnormal depending on the amount and type of virtual machines allocated to a particular node.

#### False Alarm Reduction for Malicious Traffic Detection

Despite the numerous works that have addressed the botnet detection problem, botnets are still one of the most serious threats on the Internet. The main reason for this is the constant evolution of botnets and their evasion mechanisms. This thesis has contributed to ameliorate this problem with the detection of novel botnets without historical data. However, a main challenge remains in decreasing the number of false alarms while maintaining a high detection accuracy.

A small number of false alarms is critical for efficient botnet detection, as the consequence of a high number of false alarms is that most alerts end up not being investigated due to the high human and economic costs that it entails. This means that true positives are not discovered either, making the whole process useless. In the future, we will focus on detecting malicious traffic in a real world large-scale network while reducing the number of false alarms to a manageable size. This can only be accomplished by involving network administrators in the anomaly detection design process to better understand their requirements and limitations as security experts.

# References

[1] E. B. Beigi, H. H. Jazi, N. Stakhanova, and A. A. Ghorbani, “Towards Effective Feature Selection in Machine Learning-Based Botnet Detection Approaches,” in *Proceedings of the IEEE Conference on Communications and Network Security*, 2014, pp. 247–255.

[2] M. Castells, *The Rise of the Network Society*. Wiley, 2011.

[3] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 4th ed. Addison-Wesley, 2005.

[4] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A Survey,” *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[5] E. Ronen, C. O’Flynn, A. Shamir, and A.-O. Weingarten, “IoT Goes Nuclear: Creating a ZigBee Chain Reaction,” Cryptology ePrint Archive, Report 2016/1047, 2016, http://eprint.iacr.org/2016/1047.

[6] “Google Search,” http://www.google.com.

[7] “Facebook, Inc.” http://www.facebook.com.

[8] “Amazon.com, Inc.” http://www.amazon.com.

[9] B. Cohen, “The BitTorrent Protocol Specification,” 2008.

[10] R. Dingledine, N. Mathewson, and P. Syverson, “Tor: The Second-Generation Onion Router,” in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[11] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/.

[12] W. E. Wong, D. Vidroha, A. Surampudi, H. Kim, and M. F. Siok, “Recent Catastrophic Accidents: Investigating How Software Was Responsible,” in *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement*, 2010, pp. 14–22.

[13] C. Cachin and M. Schunter, “A Cloud You Can Trust,” *IEEE Spectrum*, vol. 48, no. 12, pp. 28–51, 2011.

[14] “Southwest Airlines computer outage costs could reach $82M,” http://www.bizjournals.com/dallas/news/2016/08/11/southwest-airlinescomputer-outage-costs-could.html.

[15] “The Cost of Downtime,” http://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/.

[16] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[17] F. Salfner, M. Lenk, and M. Malek, “A Survey of Online Failure Prediction Methods,” *ACM Computing Surveys*, vol. 42, no. 3, pp. 10:1–10:42, 2010.

[18] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[19] S. D. Gribble, “Robustness in Complex Systems,” in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001, pp. 21–26.

[20] J. C. Mogul, “Emergent (Mis)Behavior vs. Complex Software Systems,” in *Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems*, 2006, pp. 293–304.

[21] D. J. Dean, H. Nguyen, and X. Gu, “UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems,” in *Proceedings of the 9th International Conference on Autonomic Computing*, 2012, pp. 191–200.

[22] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly Detection: A Survey,” *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, 2009.

[23] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, “Performance Anomaly Detection and Bottleneck Identification,” *ACM Computing Surveys*, vol. 48, no. 1, pp. 4:1–4:35, 2015.

[24] X. Shu, D. D. Yao, and B. G. Ryder, “A Formal Framework for Program Anomaly Detection,” in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses*, 2015, pp. 270–292.

[25] M. Thottan and C. Ji, “Anomaly Detection in IP Networks,” *IEEE Transactions on Signal Processing*, vol. 51, no. 8, pp. 2191–2204, 2003.

[26] Y. Zhang, N. Meratnia, and P. Havinga, “Outlier Detection Techniques for Wireless Sensor Networks: A Survey,” *IEEE Communications Surveys Tutorials*, vol. 12, no. 2, pp. 159–170, 2010.

[27] S. S. Silva, R. M. Silva, R. C. Pinto, and R. M. Salles, “Botnets: A Survey,” *Computer Networks*, vol. 57, no. 2, pp. 378–403, 2013.

[28] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov, “BotGrep: Finding P2P Bots with Structured Graph Analysis,” in *Proceedings of the 19th USENIX Security Symposium*, 2010.

[29] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, “PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems,” in *Proceedings of the 32nd International Conference on Distributed Computing Systems*, 2012, pp. 285–294.

[30] Q. Guan, S. Fu, N. Debardeleben, and S. Blanchard, “Exploring Time and Frequency Domains for Accurate and Automated Anomaly Detection in Cloud Computing Systems,” in *Proceedings of the 19th Pacific Rim International Symposium on Dependable Computing*, 2013, pp. 196–205.

[31] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant, "Botnet Detection Based on Traffic Behavior Analysis and Flow Intervals," in *Proceedings of the 27th IFIP International Information Security Conference*, 2013, pp. 2–16.

[32] A. W. Williams, S. M. Pertet, and P. Narasimhan, "Tiresias: Black-Box Failure Prediction in Distributed Systems," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[33] X. Gu and H. Wang, "Online Anomaly Prediction for Robust Cluster Systems," in *Proceedings of the 25th International Conference on Data Engineering*, 2009, pp. 1000–1011.

[34] L. Yu and Z. Lan, "A Scalable, Non-Parametric Method for Detecting Performance Anomaly in Large Scale Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 1902–1914, 2016.

[35] Z. Lan, Z. Zheng, and Y. Li, "Toward Automated Anomaly Identification in Large-Scale Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 2, pp. 174–187, 2010.

[36] G. Gu, J. Zhang, and W. Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[37] H. Choi, H. Lee, and H. Kim, "BotGAD: Detecting Botnets by Capturing Group Activities in Network Traffic," in *Proceedings of the 4th International Conference on Communication System Software and Middleware*, 2009, pp. 2:1–2:8.

[38] K. Ozonat, "An Information-Theoretic Approach to Detecting Performance Anomalies and Changes for Large-Scale Distributed Web Services," in *Proceedings of the 38th International Conference on Dependable Systems and Networks*, 2008, pp. 522–531.

[39] Y. Tan, X. Gu, and H. Wang, "Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures," in *Proceedings of the 29th Symposium on Principles of Distributed Computing*, 2010, pp. 173–182.

[40] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan, "Statistical Techniques for Online Anomaly Detection in Data Centers," in *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management*, 2011, pp. 385–392.

[41] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online Detection of Utility Cloud Anomalies Using Metric Distributions," in *Proceedings of the IEEE Network Operations and Management Symposium*, 2010, pp. 96–103.

[42] A. Karasaridis, B. Rexroad, and D. Hoeflin, "Wide-Scale Botnet Detection and Characterization," in *Proceedings of the 1st Conference on Hot Topics in Understanding Botnets*, 2007.

[43] S. Fu, J. Liu, and H. Pannu, “A Hybrid Anomaly Detection Framework in Cloud Computing Using One-Class and Two-Class Support Vector Machines,” in *Proceedings of the 8th International Conference on Advanced Data Mining and Applications*, 2012, pp. 726–738.

[44] “Google Data Center FAQ,” http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq/.

[45] “5 Numbers That Illustrate the Mind-Bending Size of Amazon’s Cloud,” https://www.bloomberg.com/news/2014-11-14/5-numbers-that-illustrate-the-mind-bending-size-of-amazon-s-cloud.html.

[46] J. Álvarez Cid-Fuentes, C. Szabo, and K. Falkner, “Online Behavior Identification in Distributed Systems,” in *Proceedings of the 34th Symposium on Reliable Distributed Systems*, 2015, pp. 202–211.

[47] ——, “Anomaly Detection in Distributed Systems Without Historical Data,” *IEEE Transactions on Dependable and Secure Computing*, (To be submitted).

[48] Y. Tan, “Online Performance Anomaly Prediction and Prevention for Complex Distributed Systems,” Ph.D. dissertation, North Carolina State University, 2012.

[49] J. Álvarez Cid-Fuentes, C. Szabo, and K. Falkner, “An Adaptive Framework for the Detection of Novel Botnets,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security*, 2017, (Under Review).

[50] “Yahoo! Webscope dataset ydata-labeled-time-series-anomalies-v1_0.” [Online]. Available: http://labs.yahoo.com/Academic_Relations

[51] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, “Using Magpie for Request Extraction and Workload Modelling,” in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[52] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, “Pip: Detecting the Unexpected in Distributed Systems,” in *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, 2006, pp. 115–128.

[53] C. S. Hood and C. Ji, “Proactive Network-Fault Detection,” *IEEE Transactions on Reliability*, vol. 46, no. 3, pp. 333–341, 1997.

[54] W. W. S. Wei, *Time Series Analysis*. Addison-Wesley, 1994.

[55] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*. Elsevier, 2014.

[56] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, “Anomaly? Application Change? or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change,” in *Proceedings of the 38th International Conference on Dependable Systems and Networks*, 2008, pp. 452–461.

[57] J. W. Tukey, *Exploratory Data Analysis*. Addison-Wesley, 1977.

[58] E. L. Lehmann and J. P. Romano, *Testing Statistical Hypotheses*. Springer, 2005.

[59] T. Kohonen, "The Self-Organizing Map," *Neurocomputing*, vol. 21, no. 1–3, pp. 1–6, 1998.

[60] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "CloudPD: Problem Determination and Diagnosis in Shared Dynamic Clouds," in *Proceedings of the 43rd International Conference on Dependable Systems and Networks*, 2013, pp. 1–12.

[61] I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, Z. Zhu, S. Bagchi, S. P. Midkiff, M. Kistler, and A. Gheith, "Automatic Problem Localization via Multi-Dimensional Metric Profiling," in *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*, 2013, pp. 121–132.

[62] M. Gabel, K. Sato, D. Keren, and S. Matsuoka, "Latent Fault Detection with Unbalanced Workloads," Lawrence Livermore National Laboratory, Tech. Rep., 2014.

[63] I. Jolliffe, *Principal Component Analysis*. John Wiley & Sons, Ltd, 2014.

[64] E. O. Aapo Hyvärinen, Juha Karhunen, *Independent Component Analysis*. Wiley, 2004.

[65] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[66] Q. Guan and S. Fu, "Adaptive Anomaly Identification by Exploring Metric Subspace in Cloud Computing Infrastructures," in *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*, 2013, pp. 205–214.

[67] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, pp. 35–45, 1960.

[68] W. T. Strayer, R. Walsh, C. Livadas, and D. Lapsley, "Detecting Botnets with Tight Command and Control," in *Proceedings of the 31st IEEE Conference on Local Computer Networks*, 2006, pp. 195–202.

[69] J. Oikarinen and D. Reed, "Internet Relay Chat Protocol," Internet Requests for Comments, RFC Editor, Tech. Rep., 1993.

[70] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "BotHunter: Detecting Malware Infection Through IDS-driven Dialog Correlation," in *Proceedings of 16th USENIX Security Symposium*, 2007, pp. 12:1–12:16.

[71] G. G. R. Perdisci, J. Zhang, and W. Lee, "BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection," in *Proceedings of the 17th USENIX Security Symposium*, 2008, pp. 139–154.

[72] T. F. Yen and M. K. Reiter, "Are Your Hosts Trading or Plotting? Telling P2P File-Sharing and Bots Apart," in *Proceedings of the IEEE 30th International Conference on Distributed Computing Systems*, 2010, pp. 241–252.

[73] X. Yu, X. Dong, G. Yu, Y. Qin, D. Yue, and Y. Zhao, "Online Botnet Detection Based on Incremental Discrete Fourier Transform," *Journal of Networks*, vol. 5, no. 5, pp. 568–576, 2010.

[74] J. Kwon, J. Lee, H. Lee, and A. Perrig, "PsyBoG: A Scalable Botnet Detection Method for Large-Scale DNS Traffic," *Computer Networks*, vol. 97, pp. 48–73, 2016.

[75] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy, "Experience Report: Anomaly Detection of Cloud Application Operations Using Log and Cloud Metric Correlation Analysis," in *Proceedings of the 26th International Symposium on Software Reliability Engineering*, 2015, pp. 24–34.

[76] T. Wang, J. Wei, W. Zhang, H. Zhong, and T. Huang, "Workload-Aware Anomaly Detection for Web Applications," *Journal of Systems and Software*, vol. 89, pp. 19–32, 2014.

[77] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 1012–1021.

[78] Q. Guan, Z. Zhang, and S. Fu, "Proactive Failure Management by Integrated Unsupervised and Semi-Supervised Learning for Dependable Cloud Systems," in *Proceedings of the 6th International Conference on Availability, Reliability and Security*, 2011.

[79] R. J. Alcock and Y. Manolopoulos, "Time-Series Similarity Queries Employing a Feature-Based Approach," in *Proceedings of the 7th Panhellenic Conference on Informatics*, 1999, pp. 27–29.

[80] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. MIT Press, 2012.

[81] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, "Pegasos: Primal Estimated Sub-Gradient Solver for SVM," *Mathematical Programming*, vol. 127, no. 1, pp. 3–30, 2011.

[82] Vert, J. P. and Tsuda, K. and Schölkopf, B., "A Primer on Kernel Methods," in *Kernel Methods in Computational Biology*. MIT Press, 2004, ch. 2, pp. 35–70.

[83] J. Kivinen, A. Smola, and R. Williamson, "Online Learning with Kernels," *Transactions on Signal Processing*, vol. 52, no. 8, pp. 2165–2176, 2004.

[84] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A Practical Guide to Support Vector Classification," 2003.

[85] J. C. Platt, N. Cristianini, and J. Shawe-Taylor, "Large Margin DAGs for Multiclass Classification," in *Proceedings of the 12th Conference in Advances in Neural Information Processing Systems*, 1999, pp. 547–553.

[86] C. Stewart, K. Shen, A. Iyengar, and J. Yin, "EntomoModel: Understanding and Avoiding Performance Anomaly Manifestations," in *Proceedings of the 18th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010, pp. 3–13.

[87] C. Reiss, J. Wilkes, and J. Hellerstein, “Google Cluster-Usage Traces: Format + Schema,” Google, Inc., Tech. Rep., 2011.

[88] “MEDEA: Modelling, Experiment DEsign and Analysis,” https://github.com/cdit-ma/MEDEA.

[89] J. M. Slaby, S. Baker, J. Hill, and D. C. Schmidt, “Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-Time and Embedded System QoS,” in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006, pp. 350–362.

[90] G. Pardo-Castellote, “OMG Data-Distribution Service: Architectural Overview,” Real-Time Innovations, Inc., Tech. Rep., 2005.

[91] G. Ricart and A. K. Agrawala, “An Optimal Algorithm for Mutual Exclusion in Computer Networks,” *Communications of the ACM*, vol. 24, no. 1, pp. 9–17, 1981.

[92] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[93] “Jenkins.” [Online]. Available: http://jenkins-ci.org/

[94] D. T. Pham and M. A. Wani, “Feature-Based Control Chart Pattern Recognition,” *International Journal of Production Research*, vol. 35, no. 7, pp. 1875–1890, 1997.

[95] K. Duan, S. S. Keerthi, and A. N. Poo, “Evaluation of Simple Performance Measures for Tuning SVM Hyperparameters,” *Neurocomputing*, vol. 51, pp. 41 – 59, 2003.

[96] B. D. Fulcher and N. S. Jones, “Highly Comparative Feature-Based Time-Series Classification,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 3026–3037, 2014.

[97] Y.-L. Wu, D. Agrawal, and A. El Abbadi, “A Comparison of DFT and DWT Based Similarity Search in Time-Series Databases,” in *Proceedings of the 9th International Conference on Information and Knowledge Management*, 2000, pp. 488–495.

[98] R. J. Freund, W. J. Wilson, and P. Sa, *Regression Analysis*. Academic Press, 2006.

[99] X. Wang, K. Smith, and R. Hyndman, “Characteristic-Based Clustering for Time Series Data,” *Data Mining and Knowledge Discovery*, vol. 13, no. 3, pp. 335–364, 2006.

[100] F. Mörchen, “Time Series Feature Extraction for Data Mining Using DWT And DFT,” Philipp University of Marburg, Tech. Rep., 2003.

[101] Z.-Q. Lu, “Estimating Lyapunov Exponents in Chaotic Time Series with Locally Weighted Regression,” Ph.D. dissertation, University of North Carolina at Chapel Hill, 1994.

[102] V. Černý, "Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm," *Journal of Optimization Theory and Applications*, vol. 45, no. 1, pp. 41–51, 1985.

[103] "Today's Outage for Several Google Services," 2014. [Online]. Available: https://googleblog.blogspot.com.au/2014/01/todays-outage-for-several-google.html

[104] "Twitter Suffers Large Outage on Web and Mobile ," https://www.theguardian.com/technology/2016/jan/19/twitter-down-over-web-and-mobile#img-1.

[105] "Amazon Web Services Outage Causes Australian Website Chaos," http://www.smh.com.au/technology/technology-news/amazon-web-services-outage-causes-australian-website-chaos-20160605-gpc41p.html.

[106] "Top500 Supercomputer Sites," https://www.top500.org/.

[107] J. Dongarra, "Report on the Sunway TaihuLight System," University of Tennessee, Tech. Rep., 2016.

[108] Q. Guan, D. Smith, and S. Fu, "Anomaly Detection in Large-Scale Coalition Clusters for Dependability Assurance," in *Proceedings of the International Conference on High Performance Computing*, 2010, pp. 1–10.

[109] R. J. Hyndman, E. Wang, and N. Laptev, "Large-Scale Unusual Time Series Detection," in *Proceedings of the 15th International Conference on Data Mining Workshops*, 2015, pp. 1616–1619.

[110] A. Zimek, E. Schubert, and H.-P. Kriegel, "A Survey on Unsupervised Outlier Detection in High-Dimensional Numerical Data," *Statistical Analysis and Data Mining*, vol. 5, no. 5, pp. 363–387, 2012.

[111] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When Is "Nearest Neighbor" Meaningful?" in *Proceedings of the 7th International Conference on Database Theory*, 1999, pp. 217–235.

[112] H. D. Vinod and J. López de Lacalle, "Maximum Entropy Bootstrap for Time Series: The meboot R Package," *Journal of Statistical Software*, vol. 29, no. 5, pp. 1–19, 2009.

[113] "Phoenix High Performance Computing," https://www.adelaide.edu.au/phoenix/.

[114] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon, "Peer-to-Peer Botnets: Overview and Case Study," in *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets*, 2007.

[115] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna, "The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-scale Spam Campaigns," in *Proceedings of the 4th USENIX Conference on Large-Scale Exploits and Emergent Threats*, 2011, pp. 4–4.

[116] "Eggdrop," http://eggheads.org/.

[117] "Hackers Release Source Code for a Powerful DDoS App Called Mirai," https://techcrunch.com/2016/10/10/hackers-release-source-code-for-a-powerful-ddos-app-called-mirai/.

[118] "DDoS Attack That Disrupted Internet Was Largest of Its Kind in History, Experts Say," https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet.

[119] "French Fighter Planes Grounded by Computer Virus," http://www.telegraph.co.uk/news/worldnews/europe/france/4547649/French-fighter-planes-grounded-by-computer-virus.html.

[120] "Conficker Left Manchester Unable to Issue Traffic Tickets," https://www.theregister.co.uk/2009/07/01/conficker_council_infection/.

[121] P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda, "Automatically Generating Models for Botnet Detection," in *Proceedings of the 14th European Symposium on Research in Computer Security*, 2009, pp. 232–249.

[122] E. Stinson and J. C. Mitchell, "Characterizing Bots' Remote Control Behavior," in *Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2007, pp. 89–108.

[123] S. Saad, I. Traore, A. Ghorbani, B. Sayed, D. Zhao, W. Lu, J. Felix, and P. Hakimian, "Detecting P2P Botnets Through Network Behavior Analysis and Machine Learning," in *Proceedings of the 9th Annual International Conference on Privacy, Security and Trust*, 2011, pp. 174–180.

[124] T.-F. Yen and M. K. Reiter, "Traffic Aggregation for Malware Detection," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008, pp. 207–227.

[125] P. Traynor, M. Lin, M. Ongtang, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta, "On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 223–234.

[126] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, Detection and Analysis of Malware for Smart Devices," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 961–987, 2014.

[127] C. Czosseck, G. Klein, and F. Leder, "On the Arms Race Around Botnets - Setting Up and Taking Down Botnets," in *Proceedings of the 3rd International Conference on Cyber Conflict*, 2011, pp. 1–14.

[128] E. Cooke, F. Jahanian, and D. McPherson, "The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets," in *Proceedings of the USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop*, 2005.

[129] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos, "Highly Resilient Peer-to-Peer Botnets Are Here: An Analysis of Gameover Zeus," in *Proceedings of the 8th International Conference on Malicious and Unwanted Software*, 2013, pp. 116–123.

[130] P. Maymounkov and D. Mazières, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,” in *Proceedings of the 1st First International Workshop on Peer-to-Peer Systems*, 2002, pp. 53–65.

[131] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling, “Measurements and Mitigation of Peer-to-Peer-Based Botnets: A Case Study on Storm Worm,” in *Proceedings of the 17th USENIX Security Symposium*, 2008.

[132] S. Thomson, Y. Rekhter, and J. Bound, “Dynamic Updates in the Domain Name System,” Internet Requests for Comments, RFC Editor, Tech. Rep., 1997.

[133] “Conficker Worm: Help Protect Windows from Conficker,” https://technet.microsoft.com/en-us/security/dd452420.aspx.

[134] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich, “Analysis of the Storm and Nugache Trojans: P2P Is Here,” *;login:*, vol. 32, no. 6, 2007.

[135] R. Schoof and R. Koning, “Detecting Peer-to-Peer Botnets,” University of Amsterdam, Tech. Rep., 2007.

[136] S. Khattak, N. R. Ramay, K. R. Khan, A. A. Syed, and S. A. Khayam, “A Taxonomy of Botnet Behavior, Detection, and Defense,” *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 898–924, 2014.

[137] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, “Your Botnet is My Botnet: Analysis of a Botnet Takeover,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 635–647.

[138] “China’s global cyber-espionage network ghostnet penetrates 103 countries,” http://www.telegraph.co.uk/news/worldnews/asia/china/5071124/Chinas-global-cyber-espionage-network-GhostNet-penetrates-103-countries.html.

[139] J. R. Binkley and S. Singh, “An Algorithm for Anomaly-Based Botnet Detection,” in *Proceedings of the USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop*, 2006, p. 43–48.

[140] J. Goebel and T. Holz, “Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation,” in *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets*, 2007.

[141] R. Villamarín-Salomón and J. C. Brustoloni, “Identifying Botnets Using Anomaly Detection Techniques Applied to DNS Traffic,” in *Proceedings of the 5th IEEE Consumer Communications and Networking Conference*, 2008, pp. 476–481.

[142] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, “From Throw-Away Traffic to Bots: Detecting the Rise of DGA-based Malware,” in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 24–24.

[143] H. Choi, H. Lee, H. Lee, and H. Kim, “Botnet Detection by Monitoring Group Activities in DNS Traffic,” in *Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, 2007, pp. 715–720.

[144] F. Haddadi, H. G. Kayacik, A. N. Zincir-Heywood, and M. I. Heywood, “Malicious Automatically Generated Domain Name Detection Using Stateful-SBB,” in *Proceedings of the 16th European Conference on Applications of Evolutionary Computation*, 2013, pp. 529–539.

[145] J. Zhang, R. Perdisci, W. Lee, X. Luo, and U. Sarfraz, “Building a Scalable System for Stealthy P2P-Botnet Detection,” *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 27–38, 2014.

[146] B. Rahbarinia, R. Perdisci, A. Lanzi, and K. Li, “PeerRush: Mining for Unwanted P2P Traffic,” in *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013, pp. 62–82.

[147] N. Kheir and C. Wolley, “BotSuer: Suing Stealthy P2P Bots in Network Traffic through Netflow Analysis,” in *Proceedings of the 12th International Conference on Cryptology and Network Security*, 2013, pp. 162–178.

[148] H. Husna, S. Phithakkitnukoon, S. Palla, and R. Dantu, “Behavior Analysis of Spam Botnets,” in *Proceedings of the 3rd International Conference on Communication Systems Software and Middleware*, 2008, pp. 246–253.

[149] M. Yahyazadeh and M. Abadi, “BotGrab: A Negative Reputation System for Botnet Detection,” *Computers & Electrical Engineering*, vol. 41, pp. 68–85, 2015.

[150] S. García, M. Grill, J. Stiborek, and A. Zunino, “An Empirical Comparison of Botnet Detection Methods,” *Computers & Security*, vol. 45, pp. 100–123, 2014.

[151] M. Stevanovic and J. M. Pedersen, “An Analysis of Network Traffic Classification for Botnet Detection,” in *Proceedings of the International Conference on Cyber Situational Awareness, Data Analytics and Assessment*, 2015, pp. 1–8.

[152] C. Kirch, B. Muhsal, and H. Ombao, “Detection of Changes in Multivariate Time Series with Application to EEG Data,” *Journal of the American Statistical Association*, vol. 110, no. 511, pp. 1197–1216, 2015.

[153] L. H. Chiang, E. L. Russell, and R. D. Braatz, *Fault Detection and Diagnosis in Industrial Systems*. Springer, 2001.