Carlos Augusto da Silva Cunha

SELF-HEALING TECHNIQUES FOR VIDEO-STREAMING APPLICATIONS

PhD Thesis in Doctoral Program in Information Sciences and Technologies supervised by Professor Marco Vieira and Professor Luis Silva and presented at the Department of Informatics Engineering of the Faculty of Science and Technology of the University of Coimbra.

September 2015

Universidade de Coimbra

FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Carlos Augusto da Silva Cunha

Self-healing Techniques for Video-streaming Applications

PhD Thesis in Informatics Engineering,
submitted to the Faculty of Science and Technology of the
University of Coimbra

Advisors
Professor Marco Paulo Amorim Vieira
Professor Luis Moura e Silva

Coimbra, September 2015

FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Carlos Augusto da Silva Cunha

Técnicas de Auto-Reparação para Aplicações de Streaming de Vídeo

Tese de Doutoramento em Engenharia Informática,
apresentada à Faculdade de Ciências e Tecnologia da
Universidade de Coimbra

Orientadores
Professor Marco Paulo Amorim Vieira
Professor Luis Moura e Silva

Coimbra, September 2015

This investigation was partially supported by FCT-Portugal under grant SFRH/BD/35784/2007, by Polytechnic Institute of Viseu (IPV) and Centre for Informatics and Systems of University of Coimbra (CISUC).

ABSTRACT

The manifesto released by IBM in 2001 observing the software complexity crisis as the main obstacle to the progress of the IT industry, has motivated the industry and the academia to explore new paradigms for software development and for maintenance and management of IT infrastructures. The Autonomic Computing paradigm was proposed by IBM to tackle the software complexity problem. It pursues the development of computing systems that can manage themselves given high-level objectives from administrators. Self-healing is one of the most important aspects of Autonomic Computing, addressing autonomous detection, diagnosis and repair of localized problems, resulting from temporary issues in software and hardware.

Video-streaming services represent one of the classes of services that most benefit from the self-healing concept. These services are characterized by soft real-time data delivery requirements, long sessions and failures with significant impact on the quality of experience of users, due to the upfront time spent when watching videos. The importance of service continuity in these services is reinforced by the sensitivity of users to the degradation of audio/video quality, and also by high quality expectations created by decades of service quality patterns provided by traditional TV.

This thesis tackles the self-healing aspect of Autonomic Computing in two video-streaming approaches: Pure Streaming (RTSP-based streaming) and HTTP Streaming. Pure Streaming is founded on the original concept of streaming, which performs synchronous transmission of video segments (groups of frames) to end-user's players. In HTTP Streaming, the video is entirely downloaded similarly to any other web object (Progressive Download), or requested in small chunks stored either in the same file or in distinct files (Adaptive Bitrate).

We propose a self-healing infrastructure for each video-streaming approach. Each infrastructure represents the framework of an Autonomic Element associated to a server node in a video-streaming delivery system. Its main activities are monitoring (data gathering and failure detection), failure prediction, failure diagnosis and repair. These activities combined constitute the self-healing lifecycle for proactive recovery of failures. The failure assumptions of the self-healing infrastructures include performance failures caused by intermittent faults and transient faults that could be overcome by restarting components or rebooting the system.

The monitoring activity gathers log data to feed the failure detection process and the other self-healing activities. Failure prediction exploits system models trained with batch learning and online learning algorithms to detect abnormal system behaviors before the occurrence of user-visible failures. All predicted failures are diagnosed to determine the failure profile (type, resource and localization of the failure), which

will decide the repair action to be executed. Repair techniques exploit container-based virtualization for efficient failover and reboot of server instances.

All self-healing activities are evaluated experimentally using benchmarks that include representative workloads, combined with fault loads that perturb the main system resources to induce performance failures with several intensities. The key premise to be validated experimentally is that the patterns on log data relative to the period between the fault activation and the failure occurrence can be modeled and recognized to anticipate failures (failure prediction) and classify them (failure diagnosis).

The results from the experimental evaluation conducted in this thesis are resumed as follows. The monitoring activity has low overheads in both Pure Streaming and HTTP Streaming infrastructures. The failure prediction and failure diagnosis activities present recall and precision values not lower than 98%, using our benchmarks in both infrastructures. In recovery scenarios, the failover of the server application using virtual containers requires, in average, 1.5 seconds to checkpoint the server state to another machine. This value also represents the minimum anticipation time provided by failure prediction to rescue the client-server connections. Later, the server can be restored in the fallback host in 1.4 seconds, in average. On the other hand, reboot-based recovery requires less than 2 seconds to be executed. In Adaptive Bitrate streaming, this activity is followed by a server warm-up period with a lifespan of 72 seconds for virtual container reboots, and 253 seconds for operating system reboots.

The main conclusions of our thesis are resumed as follows. The patterns captured by system models can provide high levels of failure prediction performance. These patterns are also powerful discriminators for diagnosis of failures detected or predicted. Yet, the diagnosis performance of predicted failures is better than the diagnosis performance of detected failures, because the latter depends on more complex failure patterns. Container-based virtualization is effective in isolating the performance of the self-healing functionality from that of the video server within the Autonomic Element, installed either in a physical or virtual machine. Container-based virtualization also ensures the execution of repair techniques without impacting the quality of experience of users, using reboot and server migration techniques. The efficiency of repair techniques also minimizes the impact of false positives on the system performance. The analysis of variance of request-response times in Adaptive Bitrate streaming is an effective approach to delimit the server warm-up period without impacting the service quality, after reboot the server's virtual container or the operating system.

Keywords: *Video-streaming, Self-healing, Failure Prediction, Failure Diagnosis, Failure Repair, Performance Failures, Dependability.*

RESUMO

A IBM publicou em 2001 um manifesto sobre a complexidade do software como obstáculo principal para o progresso da indústria de Tecnologias de Informação. Este manifesto levou a indústria e o meio académico a explorar novos paradigmas de desenvolvimento, manutenção e gestão de software. O paradigma de Computação Autónoma foi proposto pela IBM como uma solução para este problema, através do desenvolvimento de sistemas computacionais capazes de se gerir autonomamente a partir de objectivos de alto nível definidos pelos administradores. A auto-reparação é o aspeto da Computação Autónoma responsável pela deteção, diagnóstico e reparação autónoma de problemas resultantes de defeitos de software e hardware.

O conceito de auto-reparação tem uma utilidade muito importante nos serviços de streaming de vídeo. Estes serviços impõem requisitos de transmissão de dados *soft real-time*, sessões longas e falhas com um impacto significativo na qualidade de experiência dos utilizadores, devido ao tempo entretanto despendido pelos mesmos na visualização dos vídeos. A importância da continuidade neste tipo de serviços é reforçada pela sensibilidade dos utilizadores à degradação da qualidade de audio e vídeo e pelas expectativas criadas pelos padrões de qualidade da TV tradicional.

Esta tese aborda a aplicação do conceito de auto-reparação a dois tipos de serviço de streaming de vídeo: Pure Streaming (RTSP Streaming) e HTTP Streaming. Os serviços de Pure Streaming são fundados no conceito original de streaming, baseado na transmissão síncrona de segmentos de vídeo (grupos de *frames*) para os *players* dos utilizadores. Em HTTP Streaming, os dados são transmitidos sem controlo de fluxo, tal como qualquer outro objeto web (Progressive Download), ou solicitados em segmentos armazenados num único ficheiro de vídeo ou em ficheiros distintos (Adaptive Bitrate).

Nós propomos uma infraestrutura de auto-reparação para cada tipo de serviço. Cada uma dessas infraestruturas representa o *framework* de um elemento autonómico associado a um nó de servidor no sistema. As atividades principais deste elemento resumem-se à monitorização (recolha de *logs* e deteção de falhas), previsão, diagnóstico e reparação de falhas. Estas atividades combinadas representam o ciclo de vida da recuperação proativa de falhas. Os pressupostos de falhas das infraestruturas de auto-reparação incluem falhas de performance causadas por defeitos intermitentes ou transientes, reparáveis pela reinicialização de componentes ou do sistema.

A atividade de monitorização captura dados de *logs*, de forma a alimentar o processo de deteção de falhas e outras atividades de auto-reparação. A previsão de falhas utiliza os dados da monitorização para detetar comportamentos do sistema anómalos antes da ocorrência de falhas visíveis pelos utilizadores. As falhas previstas são diagnosticadas de forma a determinar o seu perfil, que vai decidir a escolha do tipo de ação

de reparação a executar. As técnicas de reparação exploram virtualização baseada em contentores virtuais para migração e reinicialização de servidores.

As actividades de auto-reparação são avaliadas experimentalmente recorrendo a *benchmarks* com cargas de trabalho e cargas de falhas utilizadas por perturbar os recursos do sistema, de forma a induzir falhas de performance com várias intensidades. A premissa principal a validar é a existência de padrões nos dados de *log*, no período entre a ativação do defeito do software e a ocorrência de falhas, passíveis de serem modelados e identificados para antecipação (previsão) e classificação (diagnóstico) de falhas.

Os resultados experimentais mais relevantes da avaliação das atividades de auto-reparação podem ser resumidos como se segue. A atividade de monitorização da infraestrutura de auto-reparação tem custos de performance reduzidos. A previsão de falhas e o diagnóstico de falhas apresentam níveis de *recall* e *precision* superiores ou iguais a 98%, em ambas as infraestruturas, utilizando os *benchmarks* propostos. Em cenários de recuperação, a migração do *checkpoint* do servidor para a máquina de destino pode ser feita em 1.5 segundos, em média. Posteriormente, a actividade do servidor pode ser restabelecida na máquina de destino em 1.4 segundos, em média. A reparação através da reinicialização de contentores virtuais requer menos do que 2 segundos para ser executada. Em serviços Adaptive Bitrate, esta atividade é complementada por um período de *warm-up* de 72 segundos para contentores virtuais e 253 segundos para o sistema operativo.

As conclusões principais da nossa tese podem ser resumidas como se segue. Os padrões capturados pelos modelos do sistema fornecem altos níveis de performance na previsão de falhas. Esses padrões são também bons classificadores para diagnóstico de falhas previstas ou detetadas. No entanto, a performance do diagnóstico de falhas previstas é superior à das falhas detetadas, uma vez que as últimas dependem de padrões mais complexos. A virtualização baseada em contentores virtuais é eficaz no isolamento da performance associada à funcionalidade de auto-reparação relativamente à do servidor de vídeo, dentro do mesmo elemento autonómico instalado num máquina física ou virtual. Este tipo de virtualização suporta a execução de técnicas de reparação sem comprometer a qualidade de experiência dos utilizadores, recorrendo a técnicas de reinicialização e migração de contentores virtuais entre máquinas. A eficiência destas técnicas permite também minimizar o impacto de falsos positivos na performance do sistema. Em serviços Adaptive Bitrate, a análise da variância dos tempos de resposta do servidor a pedidos é uma abordagem eficaz para delimitar o período de *warm-up* do servidor sem comprometer a qualidade de experiência dos utilizadores.

Keywords: *Streaming de Vídeo, Auto-Reparação, Previsão de Falhas, Diagnóstico de Falhas, Reparação de Falhas, Falhas de Performance, Confiabilidade.*

ACKNOWLEDGMENTS

After several years of work, this is an important milestone in my life. It is the end of a path full of obstacles rooted in the coordination of a Ph.D. student life with my personal and professional life as a full-time teacher and program coordinator in the Polytechnic Institute of Viseu. During that time, several people through their ideas, opinions, and encouragement, helped me along.

I would like to start by thanking my advisors, Professor Luis Silva and Professor Marco Vieira. They are responsible for opening my mind to new ways of looking at problems and providing support through my academic journey. I am also truly grateful to João Paulo Magalhães, Carlos Vaz, Fernando Costa and Vitor Carreira, which were good traveling companions.

I wish to thank also to the Polytechnic Institute of Viseu for paying me the salary during that time, to do something I have wanted to do since I was kid. Likewise, I would like to thank the Engineering Informatics Department and the Software and Systems Engineering Group (CISUC) for the financial and material support granted.

Finally, I want to thank to my lovely son Ivo and to my lovely girlfriend Claudia for supporting me emotionally. I am also deeply thankful to my parents and my sister for their amazing support and care.

PUBLICATIONS

JOURNALS

1. Cunha, C. A. and Silva, L. M., "Building Autonomic Elements from Video-streaming Servers", IEEE Transactions on Dependable and Secure Computing (TDSC) [Under Revision Process]
   Journal ranking at CORE list: A

2. Cunha, C. A. and Silva, L. M., "Reboot-based Recovery of Performance Anomalies in Adaptive Bitrate Video-Streaming Services", Special Issue on High Performance Computing in Parallel and Distributed Systems (IJHPCN) [Accepted for Publication]
   Journal ranking at CORE list: B

CONFERENCES

1. Cunha, C. A. and Silva, L. M., "Failure Prediction in Video-Streaming Servers Through Performance Analysis of Server and Client-Server Interactions", Proceedings of the The Tenth International Conference on Parallel and Distributed Computing and Networks (PDCN'11), Austria, 2011

2. Cunha, C. A. and Silva, L. M., "Application of a self-healing video-streaming architecture to RTSP servers", The 10th IEEE International Symposium on Network Computing and Applications (IEEE NCA'11), Cambridge, USA, 2011
   Conference ranking at CORE list: A

3. Cunha, C. A. and Silva, L. M., "Separating Performance Anomalies from Workload-explained Failures in Streaming Servers", The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (IEEE CCGRID'12), Ottawa, Canada, 2012
   Conference ranking at CORE list: A

4. Cunha, C. A. and Silva, L. M., "Prediction of Performance Failures in Video-streaming Servers", The 19th IEEE Pacific Rim International Symposium on Dependable Computing, Vancouver, Canada, 2013
   Conference ranking at CORE list: B

5. Cunha, C. A. and Silva, L. M., "SHStream: Self-healing Framework for HTTP Video-Streaming", The 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (IEEE CCGRID'13), Delft, NL, 2013
   Conference ranking at CORE list: A

6. Cunha, C. A. and Silva, L. M., "Reboot-based Recovery of Performance Anomalies in Adaptive Bitrate Video-Streaming Services", The 21th IEEE Pacific Rim International Symposium on Dependable Computing, Zhangjiajie, China, 2015
   Conference ranking at CORE list: B

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The cost of maintenance of computer systems by human operators has increased proportionally to the ever-increasing complexity of these systems in the recent years. The large size of computer systems, the diversity of technologies, the high frequency of updates, the interconnectedness of systems and the adoption of emerging pervasive computing devices demand new approaches to deal with the management of actual computer systems. This problem is even more challenging in services with high sensitivity to performance changes, such as video-streaming services.

Video-streaming services are predominant in the universe of Internet services. Their dramatic growth in the last years has been potentiated by the convergence of TV with the Internet and the increasing popularity of videos in e-learning, VoD and social network services. On top of that, the ever increasing market of mobile video streaming has been boosted by the rollout of the 4G service.

The creation of content, video formats and technology for video-streaming services are hot topics in the research agenda of academics and are leveraging an entire software industry. As well, the advent of HTML5 [Pilgrim 2010] has been in the origin of a galaxy of new technologies, responsible for increasing the pervasiveness of video-streaming services, customized for all types of devices (e.g., smartphones, tablets, PCs).

Several reports show that video content distribution dominates the Internet traffic. More precisely, real-time entertainment is responsible for over 68% of downstream bytes, during the peak period in the North America [SANDVINE 2013]. Other statistics [Systems 2013] show that: Internet video will increase from 17,455 (PB per month) in 2013 to 62,972 (PB per month) in 2018; and IP video traffic will be 79% of all consumer Internet traffic in 2018, up from 66 percent in 2013.

Video-streaming services bring new challenges to content delivery infrastructures. They are throughput-based services with long client-server sessions, during which, data are transmitted to video players in small chunks. These chunks have strict delivery deadlines to ensure service continuity.

Continuity is a critical aspect of video-streaming services due to large user abandonment costs resulting from interruptions in the middle of video playback, after an upfront time invested by end-users watching the videos. Service continuity is easily compromised in the form of quality degradation caused commonly by performance problems occurring server-side or in the network [Wijesekera et al. 1999][Jiang and Schulzrinne 2002]. Quality degradation has a strong impact on the Quality of Experience (QoE), due to the human sensitivity to variations in video and audio quality. Plus,

there are large demands for high quality imposed by the quality patterns provided by traditional TV during decades.

Performance problems in video-streaming systems leading to service quality degradation can have a similar impact on the viewer's experience as service unavailability. Playback interruptions and video glitches are disturbing events that are expected as a consequence of performance degradation. These events require urgent intervention for restoring the service to its original status. In the server context, performance failures can be classified into workload-related failures — due to the lack of or ineffective load control — and performance anomalies caused by software faults.

Detection and localization of performance problems can benefit from the use of models of the system's behavior to recognize anomalies. However, in complex systems with hundreds or thousands of server instances, it is impracticable to maintain specific performance models for each instance and verify compliance of the monitored server behaviors with that models globally. Autonomic Computing [Kephart and Chess 2003][Ganek and Corbi 2003] deals with that problem by exploiting the concept of self-awareness, which states that each Autonomic Element (server, component or subsystem) should be aware of its internal state and behavior. Self-awareness enables continuous analysis of the system behavior, accompanied with diagnosis and recovery activities whenever the Autonomic Element detects any deviation from its normal behavior. Therefore, it is possible to provide timely and decisive responses to performance problems that could impact the users' experience.

This thesis addresses the topic of self-healing in video-streaming services, focusing on monitoring, online failure prediction, proactive diagnosis and proactive recovery activities.

1.1 CONTRIBUTIONS

We propose two self-healing infrastructures for video-streaming services. The first self-healing infrastructure is devised for traditional Pure Streaming services, also known as RTSP services [PIP 1998]. The second infrastructure, named SHStream, targets HTTP Streaming services. Both self-healing infrastructures implement monitoring, failure prediction and failure diagnosis. The HTTP Streaming infrastructure also implements repair using virtualization techniques.

1.1.1 *Self-healing Infrastructures*

Each self-healing infrastructure proposed represents an Autonomic Element in a video system. The design of each self-healing infrastructure addresses important issues, as such:

- Localization of the monitoring probes for data gathering of application-specific performance metrics, and also system, network and service quality metrics;
- Monitoring with low data gathering latencies, to provide fast responses to faulty conditions;
- Scalability of the self-healing activities;
- Performance isolation between the main functionality of the server and the self-healing activities, within the same Autonomic Element.

We propose a HTTP Streaming infrastructure that employs *container-based virtualization* [Soltesz et al. 2007] to ensure performance isolation between the server application and the self-healing activities. Thus, the self-healing functionality can be attached to any video server to build an Autonomic Element. Container-based virtualization is also exploited in the infrastructure to support efficient repair actions.

The monitoring activity is designed for each self-healing infrastructure to comply with data and performance requirements of the Pure Streaming and HTTP Streaming services. Its main tasks are gathering log data and failure detection.

Related Publications: [Cunha and Silva 2011][Cunha and Silva 2013a][Cunha and Silva 2016]

1.1.2 *Failure Prediction*

Failure prediction is a fundamental activity of the self-healing lifecycle. It creates the opportunity to diagnose and repair failures proactively before end-users experience any failure.

We propose a failure prediction approach based on the analysis of patterns of failure, observed in the log data. These patterns are recognized using classifiers created by machine learning algorithms. In the Pure Streaming infrastructure, these classifiers are trained with historical data, using *batch learning* algorithms [Witten et al. 2011]. In the HTTP Streaming infrastructure, the classifiers are trained with *data stream mining* algorithms [Bifet and Kirkby 2009]. This is a new class of *online learning* algorithms used to build classifiers iteratively, allowing the update of classification models when new learning data becomes available. This characteristic contrasts with *batch learning* algorithms, which avoid the update of classification models.

Important issues addressed in the failure prediction activity are: (1) failure prediction performance; (2) look-ahead time provided by failure prediction; and (3) quantity of learning data required to obtain the maximum performance of failure prediction models.

Related Publications: [Cunha and Silva 2013b][Cunha and Silva 2013a]

1.1.3 *Failure Diagnosis*

All predicted failures should be followed by countermeasures to restore the service. Diagnosis is an important activity for selection of appropriate repair actions for the underlying failure causes. These causes are attributed to: (1) software faults leading to system behaviors unexplained by the current workloads (e.g., full CPU utilization when the server is at half of its nominal capacity); (2) load control faults; and (3) network faults.

The Pure Streaming and HTTP Streaming infrastructures implement diagnosis differently. The Pure Streaming infrastructure discriminates workload-related failures from performance anomalies caused by software faults, in the diagnosis activity. The HTTP Streaming infrastructure implements a more complex diagnosis process. It classifies the origin of the error (server machine or network), the resource directly involved (CPU, memory or I/O) and the internal location of the error (server application or system-level). The classification outcomes will decide the granularity of the reboot technique or otherwise, the execution of a failover mechanism.

Important issues addressed in the failure diagnosis activity are: (1) diagnosis performance for failures predicted and failures detected; and (2) quantity of learning data required to obtain the maximum performance of diagnosis models.

Related Publications: [Cunha and Silva 2016][Cunha and Silva 2012]

1.1.4 *Failure Repair*

The SHStream infrastructure exploits container-based virtualization techniques for repairing the Autonomic Element. Repair techniques are coarsely divided into:

- **Server migration techniques** — based on checkpointing and migration of the server application to another machine. These techniques rescue the server in-memory state and client-server connections. They are adequate to recover environmental errors not originated or propagated to the virtual container running the server application;
- **Reboot techniques** — reboot the operating system or the virtual container running the server application. Operating system reboots can be combined with server migration for recovering the server along with client-server connection states. On the other hand, virtual container reboots are required for recovering from virtual container's internal failures (e.g., failures in the server application), but are disruptive for video-streaming connections established between clients and the server.

The time required to execute each repair technique is a main issue addressed in the failure repair activity. The execution time of each repair technique represents the

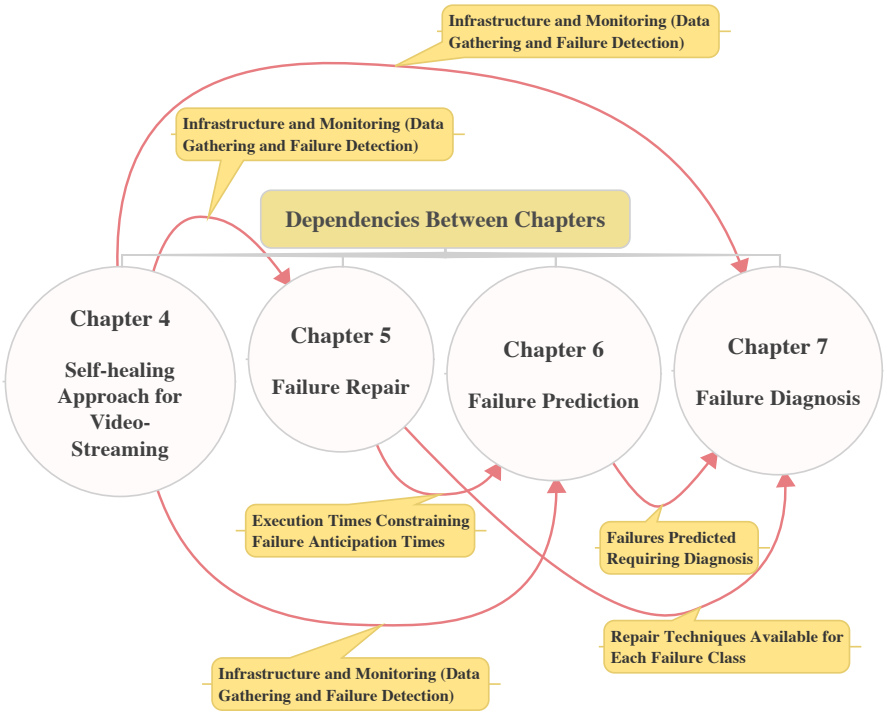


Figure 1: Dependencies between chapters of this thesis.

service downtime visible by end-users. In the same way, the time necessary to rescue the server checkpoint by transferring it to another host, represents the minimum look-ahead time required from the failure prediction activity to restore the server application along with the video-streaming connections in the destination host.

The reboot technique is complemented by a server warm-up approach. The reduction of the system capacity after a reboot would lead to temporary failures. Thus, the server warm-up approach circumvent the problem of providing the service to end-users without failures during the warm-up period, until the system reach its maximum capacity.

Related Publications: [Cunha and Silva 2015a][Cunha and Silva 2015b]

1.2 STRUCTURE OF THE THESIS

This thesis is structured into eight chapters. These chapters have dependencies between themselves (see Figure 1) in terms of concepts, execution cycle of the presented approaches and related experimental results.

Chapter 1 introduces this thesis and Chapter 2 presents the background concepts of Autonomic Computing, self-healing, failure analysis, video-streaming technologies and virtualization. These concepts are necessary to understand the work presented in this thesis.

Chapter 3 presents a state of the art on self-healing systems, video-streaming dependability and recovery using virtualization techniques. It also identifies the research gaps in the literature that will be addressed in this thesis.

Chapter 4 starts by presenting the self-healing approach developed in this thesis, the general problem description, the research goals, the self-healing problem space and the self-healing infrastructures for Pure Streaming and HTTP Streaming technologies. Then it presents the evaluation methodology and the benchmarks used for the evaluation of each self-healing activity in the further chapters. Finally, it describes the monitoring activity designed for each self-healing infrastructure, focusing on the failure model, log data gathered for analysis, failure detection approach and experimental results exhibiting data gathering delays and overheads. The other self-healing activities depend on the metrics gathered for analysis by the monitoring activity and on the outcomes of the failure detection process.

Chapter 5 addresses failure repair using virtualization techniques. Several repair techniques are evaluated along with the server warm-up approach proposed to avoid temporary failures. Repair actions are triggered by the failure detector (reactive repair) or by the failure prediction activity (proactive repair).

Chapter 6 proposes and evaluates experimentally a prediction approach for performance failures in video-streaming systems. The evaluation of the failure prediction performance is performed for the look-ahead time required to execute the repair actions before the occurrence of failures.

Chapter 7 presents the failure diagnosis approach that complements the failure detection (reactive diagnosis) and failure prediction (proactive diagnosis) activities for identification of the failure profile that will indicate the appropriate repair action.

Chapter 8 closes this thesis with the most important conclusions and describes the directions that convey the future work succeeding this thesis.

2

BACKGROUND

This chapter presents the background and the main concepts that sustain the work undertaken in this thesis. Firstly, it starts by presenting an overview of the Autonomic Computing paradigm and its main aspects, with emphasis on the self-healing aspect, which is exploited extensively in this thesis. Secondly, it presents a description of the main concepts of failure analysis and the relation between these concepts. Thirdly, it revisits the most relevant video-streaming technologies and their basic concepts. Lastly, it describes the characteristics of the virtualization technologies exploited by the self-healing infrastructures presented in this thesis.

2.1 AUTONOMIC COMPUTING

The work undertaken in this thesis is motived by the concept of Autonomic Computing. In October 2001, IBM introduced the concept of Autonomic Computing through a manifesto [Horn 2001] used to denote a new generation of computing systems that can manage themselves given high-level objectives from administrators. Autonomic computing refers to a tangled hierarchy of self-governing systems, composed by a myriad of interacting self-governing components that free system administrators from the details of system operation and maintenance. The vision of Autonomic Computing was later published [Kephart and Chess 2003][Parashar and Hariri 2005][Ganek and Corbi 2003] after some development had taken place. Thereafter, this area has attracted numerous computer science researchers from several research domains.

The four aspects of Autonomic Computing are:

- **Self-configuration**. The system configure itself automatically in accordance with high-level policies. New components incorporate themselves automatically into the system, and the rest of the system will adapt to their presence;
- **Self-optimization**. Complex systems may have hundreds of tunable parameters requiring adjustment for the system to perform optimally. Systems must continually seek ways to improve their operation, by monitoring, experimenting with, and tuning their own parameters and upgrade their function proactively, by finding, verifying, and applying new updates;
- **Self-healing**. The system perceive that it is not operating correctly and, without or with limited human intervention, makes the necessary adjustments to restore itself to normal operation;

- **Self-protection**. The system defends itself as a whole against problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures.

Autonomic systems are organized into collections of Autonomic Elements related among them, each one controlling a system constituent known as Managed Element. Autonomic Elements manage their internal behavior in accordance with specified policies. One Managed Element can be any software or hardware resource (e.g., web server or database) that is given autonomic behavior by coupling it with an Autonomic Manager.

IBM suggested a reference model for Autonomic Elements based on control loops called MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge), presented in Figure 2. In the MAPE-K control loop, the data gathered by sensors are analyzed by the Autonomic Manager to monitor the Managed Element. Therefore, using the internal knowledge of the system — produced by all MAPE-K tasks — the Autonomic Manager analyzes whether the monitored information follows a designated action (e.g., by comparing status information to system specific thresholds). Designated actions are aligned with high-level goals established by human administrators for specific Autonomic Elements or the whole system. When the monitored information is not in conformity with the goals, it is planned a series of changes to be effected on the Managed Element.

This thesis addresses the self-healing aspect of Autonomic Computing. Therefore, the Autonomic Manager should hold the features required to provide the Managed Element with self-healing behavior.

2.2 SELF-HEALING SYSTEMS

The term self-healing was drawn from the biological paradigm, as biological systems exhibit adaptation, healing and robustness in face of continuing changing environmental behavior. Accordingly, self-healing systems automatically detect, diagnose and repair localized software and hardware failures resulting from the activation of *permanent faults* or *transient faults* [Avizienis et al. 2004]. Self-healing systems should perceive that they are operating incorrectly and, without or with limited human intervention, make the necessary adjustments to restore itself to normalcy. Most of the time, self-healing activities are performed using knowledge captured from system configurations, log files and runtime information provided by monitored components.

Characterization of the self-healing activities represents one important step towards the comprehension of the responsibilities of this aspect in the construction of full autonomic systems. Researchers have different views on what comprises research in self-healing systems [Saha 2007][Lemos 2003]. However, there is a common acceptance that self-healing:

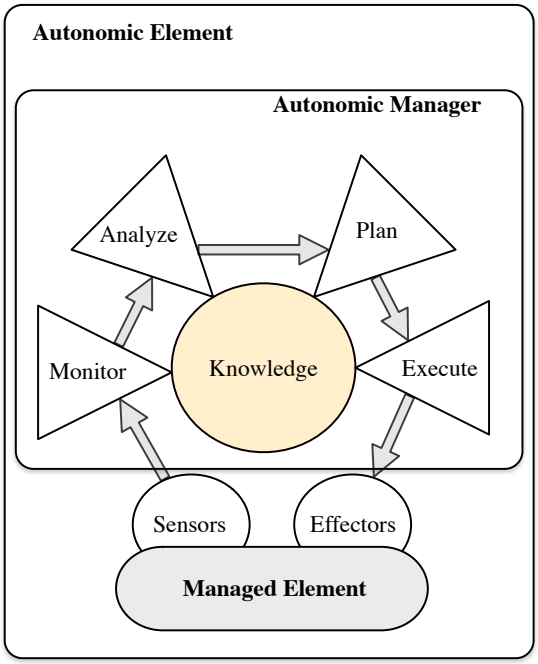


Figure 2: Structure of an Autonomic Element.

- Is equivalent to fault tolerance in dynamic systems that do not know during design time what resources they might have during runtime;
- Deals with imprecise specification and uncontrolled environments;
- Deals with adaptation and reconfiguration of systems according to their dynamics.

As long as self-healing approaches deal with dynamic systems and imprecise specification, their fault assumptions are less restrictive than in traditional fault tolerance approaches. Seeing that, traditional system fault models should be adapted to the characteristics of self-healing systems.

Similarly to other fault tolerance approaches, self-healing solutions should be specified according to the respective problem space. One important contribution for the specification of the self-healing problem space is given in [Koopman 2003]. That paper proposes four categories of aspects for the self-healing problem (Figure 3), based on the same terminology presented in [Avizienis et al. 2004] for fault tolerance, namely:

- **Fault-model**. The fault-model states what faults or injuries should be self-healed, which includes the fault duration, fault source (e.g., operational or implementation errors and defective system requirements) and other fault characteristics;

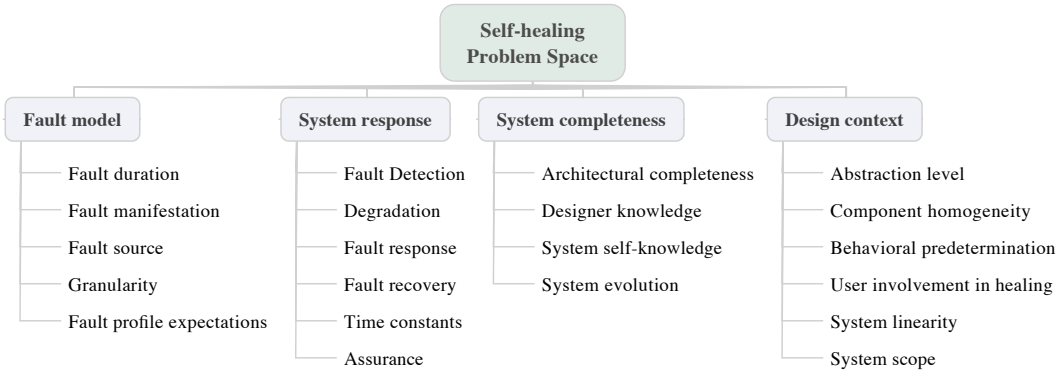


Figure 3: Self-healing problem space.

- **System response**. System response aspects consider fault detection, specification of the degree of degraded operation provided by self-healing, fault response and fault recovery issues. Fault detection is a broad area that includes techniques such as application semantics-driven assertions, supervisory checks, computer answers examination, comparison of replicated components, online self-testing, between others. On the other hand, recovery can be performed partially or completely. The effectiveness of recovery depends on the built-in redundancy and fault response techniques implemented (e.g., fault masking, retry, rollback and rollforward operations);

- **System completeness**. The incompleteness of specifications and designs and the limited knowledge of systems have implications on their recoverability. A thorough understanding of application semantics and about the system behavior in the absence and presence of faults is required to develop self-healing systems. However, there are many challenges to obtain the complete knowledge about the system, such as, the large frequency of updates and the use of Commercial Off-the-Shelf (COTS) components by several modern systems. Self-knowledge and system evolution are important research topics akin to this problem;

- **Design-context**. The design-context addresses abstraction-level problems, such as the component-level heterogeneity, pre-deterministic behaviors, system scope, system linearity and user involvement degree.

The specification of self-healing solutions proposed in this thesis is aligned with the aforementioned problem space. Details of each aspect inherent to the self-healing infrastructures developed in this thesis are presented in Chapter 4.

2.3 BASIC CONCEPTS OF FAILURE ANALYSIS

This section describes the characteristics of a *correct service* and defines several concepts respecting the origin and manifestation of failures.

Functionality, *performance* and *dependability* are three fundamental system characteristics. Functionality represents the specification of what the system is intended to do and is constrained by established performance requirements. Dependability represents the ability of one system to deliver a trusted service, which can be described as its aptitude to avoid functional and performance failures that are more frequent and more severe than is acceptable [Avizienis et al. 2004]. Providing a dependable service is the main goal of any self-healing approach for guaranteeing a correct service to end-users.

One system is providing a correct service to end-users when it yields the functionality intended for the service it provides and satisfies the performance parameters specified for that functionality. However, the ever increasing complexity of actual systems makes them propitious to *latent faults* residing in production systems until their activation. Faults are commonly introduced:

- During the development phase;
- Through interactions with external entities belonging to the system environment (e.g., software, hardware or humans);
- By physical defects at the hardware level.

Under specific conditions, faults are activated as system errors during runtime execution of software, manifesting later as service failures. Thus, failures represent errors visible by end-users, in the form of service unavailability, erratic service or service quality degradation. In general terms, one failure occurs when the service deviates from a correct service state. The meaning of correct service is described by the failure model designated for the service.

Dependability techniques can actuate at either fault, error or failure levels. At the fault level, static analysis tools have been used to help programmers removing defects prior to the release of a software product [Zheng et al. 2006][Xie et al. 2007]. At the error and failure levels, research on fault tolerance techniques have been done exhaustively with the aim of reestablishing fully or partially the functionality and performance defined for the system during faulty periods. Fault tolerance techniques are applied:

- At the error level, when the system monitoring tools detect the presence of errors, by capturing error messages or error signals and actuate to avoid service failures;
- At the failure level, when the service monitoring tools detect service failures and actuate to reestablish the service.

The mechanism for detection of errors and failures is designed for specific error conditions and failure conditions, respectively. These conditions are also associated to the *failure modes* representing the different forms of deviation from correct service.

Figure 4 presents a taxonomy for classification of failure modes, similar to that presented in [Avizienis et al. 2004]. It classifies failures in terms of their *domain*, *detectability*, *consistency* and *consequences*.

In terms of their domain, failures can be classified as *content failures*, when they interfere with the content handled by the system and *timing failures*, when they only interfere with the delays on manipulating the data processed by the system. *Fail-stutter failures* [Arpaci-Dusseau and Arpaci-Dusseau 2001] are timing failures that lead the system performance to levels below its performance specification. Hence, they allow the service to be maintained but with degraded performance. Error conditions associated to these performance problems are also denoted as *performance anomalies* in the literature. When both content and timing are incorrect, the failures are classified as *halt failures* and *erratic failures*. Halt failures (also known as *fail-stop failures*) occur when the system activity is no longer perceptible to end-users. These failures are also associated to *crash failures*, since when the service application crashes, it stops answering requests issued by end-users. This type of failures requires low-complexity detection methods, which looks for the absence of responses to external stimulus (i.e., requests) during a specified period of time. For erratic failures, the service is delivered but is erratic.

In terms of their detection, *detected failures* (*signaled failures*) are detected by detecting mechanisms. Otherwise, they are denoted as *latent errors* (*unsignaled failures*), in case they are undetected. When the failure is detected without failure occurrence, it is considered an *false alarm*.

In terms of consistency, *consistent failures* perturb the service correctness provided to all users. On the other hand, *inconsistent failures* impact the system nondeterministically and would lead to different perception of service failures by different users. Inconsistent failures are also called *byzantine failures* [Lamport et al. 1982]. These failures are characterized by nondeterministic system states and are not detected by conventional detection mechanisms. Instead, they are commonly handled by diversity techniques [Chun et al. 2008], using redundant components and applications with different implementations or running in different infrastructures. Redundancy are complemented with voting strategies (e.g., *majority voting*) [Parhami 1994] over the outputs of all replicas, to decide the system output.

In terms of severity, errors can be *minor*, when their consequences are of similar cost to the benefits provided by correct service delivery. Otherwise, they can be *catastrophic*, when their cost is orders of magnitude higher than the benefit provided by correct service delivery.

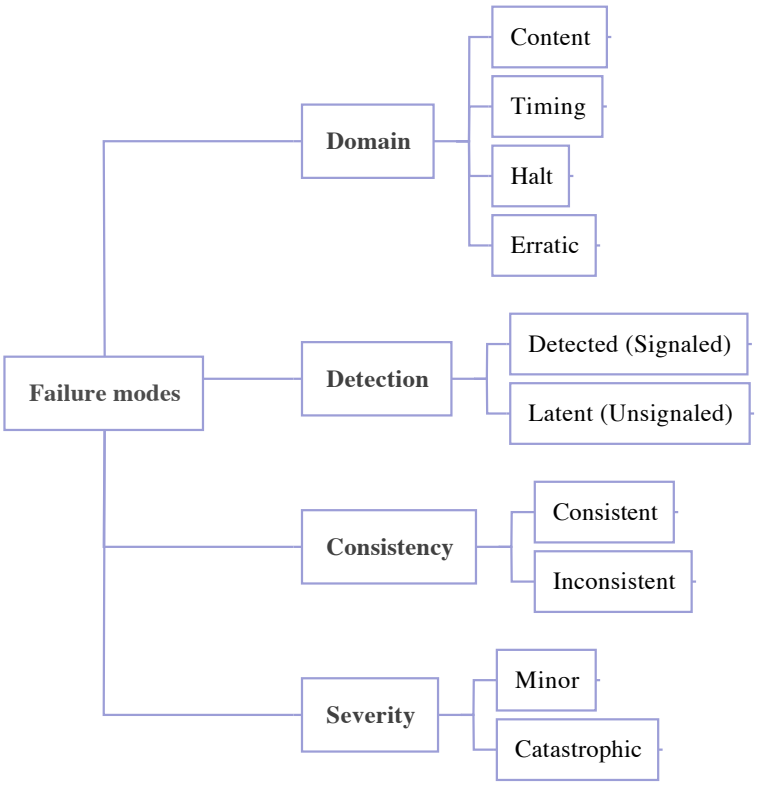


Figure 4: Classification of Failure Modes.

2.4 FAILURE CHARACTERIZATION IN VIDEO-STREAMING SERVICES

Self-healing mechanisms for detection, diagnosis and repair of software failures are necessarily designed for specific failure profiles. Hence, the first step in devising a self-healing system is defining the meaning of failure, which depends on the service type.

The self-healing mechanisms devised in this thesis for video-streaming services cover performance failures caused by server faults (originated at system or application levels), network faults and also load control faults leading to server overloading. These failures are classified in terms of severity as: *hard failures* and *soft failures*.

Hard failures are fail-stop failures. They are universal [Avizienis et al. 2004], since they are objective and unambiguous, and are characterized by the total unavailability of service to end-users. By contrast, soft failures are characterized specifically to one type of service. Plus, they are often the reflex of system performance errors resultant from complex interactions of faulty and non-faulty system elements.

2.4.1 *Soft Failures*

Soft failures are fail-stutter failures. Since this thesis focuses on server-side performance failures, soft failures start on performance degradation of the server or server-side network and ends up on degradation of the service quality experienced by end-users. Accordingly, soft failures are caused by:

- **Large server transmission delays** — the server is one potential contributor to delays on transmission of video-streaming data during performance degradation periods;
- **Network packets delayed or lost** — performance problems in the server-side network (e.g, due to network congestion) can avoid the arrival of frames to video players on time for playback.

In typical Internet services with short request-responses (e.g., web page requests), soft failures are identified through comparison of request-response delays with specified thresholds. However, the specification of soft failures in video-streaming services represents a complex and non-universal problem. In these services, request-responses are broken down into fragments (containing groups of frames) with strict time delivery constraints, which are transmitted progressively by the server often during long periods of time. Thus, failures are experienced by end-users when delays on transmission of individual video fragments will reduce the flow of data required by players for continuous playback of videos. In other words, failures occur when video fragments are undelivered, or delivered by players after their playback time, causing degradation of the Quality of Experience (QoE) of end-users.

Soft failures can be specified in terms of the QoE. Approaches for evaluation of the perceptual quality rely on the analysis of video content being played out for determining the QoE of end-users. The Peak Signal Noise Ratio (PSNR) is a common metric for measuring the video quality through analysis of video content [Klaue et al. 2003] [Bucciol et al. 2005]. However, metrics like PSNR depend not only on the correctness of the video delivery service but also on the encoding quality (not addressed in this thesis). PSNR is unable to discriminate quality degradation introduced at the encoding phase from quality degradation caused by service delivery problems. Additionally, PSNR can only be computed client-side after video decoding, requiring player instrumentation and adding failure detection overheads.

Discarding the analysis of the video quality at the encoding level, the specification of soft failures in video-streaming must rely on the association of video fragments not delivered, or delivered after their playback time, with the user's QoE. This strategy depends on the protocol adopted for transmission of data between the server and players, as will be further explained.

2.4.2 *Failure Specification Using Protocols Without Data Delivery Guarantees*

The UDP protocol naturally supports graceful degradation of video quality in video-streaming services, by allowing playback of video content without waiting for late packets or the retransmission of lost packets. Therefore, the video playback is performed with the data already received by the players, even with some quality degradation.

Graceful degradation of video quality results into arduous failure detection, due to the complexity of determining the impact of lost packets and packets arriving after their playback time on the end-users' QoE. Several studies determined the quality of multimedia services as a function of three content delivery performance parameters: *packet delivery delays*, *delays variation* and *information loss* [Claypool and Tanner 1999][Jiang and Schulzrinne 2002][Ghinea and Thomas 1998][Cotroneo et al. 2003]. In these studies, the analysis of the impact of service parameters on the perceptual quality is based on thresholds that discriminate quality levels. As well, the SG 12 study group of ITU-T, dedicated to definition of the quality attributes of multimedia services from the user perspective, proposed the use of these service parameters to characterize the quality of multimedia services [Coverdale 2001].

Other empirical studies correlate the user's perceptual quality with data transmission losses. As an example, it was observed in [Wijesekera et al. 1999] that: (1) video quality is visibly degraded when more than two consecutive frame losses is experienced for video, and more than three frame losses is experienced for audio; (2) humans are much more sensitive to audio losses than to video losses and are low sensitive to video rate variations; (3) average losses below 17 out of 100 frames is imperceptible for video playout rates of 30 frames per second; and (4) only audio-video desynchronization above three frames is perceptible by users. These observations are consistent with the findings presented in [Steinmetz 1996].

Previous empirical studies based on thresholding of service parameters for QoE determination have been validated for videos with specific characteristics. However, the lack of recognition and standardization of metrics and corresponding thresholds in the academia and industry makes failure definition non-universal and requiring laborious parameterization of failure detectors for each service configuration. However, this problem is complex because it depends on the empirical mapping between data losses — including data arriving late for playback — and the users' QoE. That mapping is defined by non-universal factors, such as, the minimum quality accepted by a class of users, the distribution of data losses of individual audio and video components over time, the encoding characteristics and the specification of the device's screen used for visualization [Ghinea and Thomas 1998][Wijesekera et al. 1999][Steinmetz 1996].

2.4.3 *Failure Specification Using Protocols With Data Delivery Guarantees*

TCP became the most popular protocol for transmission of video-streaming over the Internet. The firewalls and NAT Traversal problems solved by HTTP over TCP leveraged the adoption of TCP for delivering video-streaming content in the Internet. Popular video-streaming services like Youtube [you 2012] and Vimeo [vim 2012] use TCP to transport video content between video-streaming servers and players.

TCP provides in-order data delivery guarantees to player applications. This characteristic forces video playback to stop when there are delayed or lost packets in the sequence of packets necessary to ensure playback continuity. Thus, graceful degradation of service quality is disallowed, as the TCP protocol forces packets to be received in order before being delivered to the video player. However, this assumption simplifies the failure detection process because the service quality can be measured in terms of the time spent by end-users waiting for watching videos.

Soft failures in video-streaming services provided through TCP can be detected by establishing one single threshold over the time spent by end-users waiting for watching videos. That threshold determines whether the *user waiting time* is large enough to classify the service quality as a failure. The user waiting time is calculated through the analysis of connection establishment delays and player buffer states. The player buffer can be in one of two states: *playing* or *buffering*. Therefore, the user waiting time is calculated by summing the connection establishing time to all (re)buffering times. During playing periods, the user is watching the video and there are enough data stored in the player buffer to ensure playback continuity. After connection establishment or during *buffer underflow* events, the player is in the buffering state and the user is waiting for watching the video.

Notwithstanding the maximum user waiting time tolerated by end-users is subjective, its value is independent of any encoding type and service delivery infrastructure and is also interpretable in the user QoE domain. Thus, the threshold value can be set by a Service Level Objective (SLO) for the video-streaming service. SLOs are specified according to the perception of the minimum service quality accepted by end-users. Standard industry metrics based on user waiting times also exist.

The Keynote StreamQ Grade [key 2010] is a popular metric for classifying user waiting times into service quality degradation levels, in video-streaming services. It is a leading industry standard metric that grades user experience by quantifying the *frustration time* — the time the user is waiting for watching the video. This metric sums the *connection time*, the *buffering time* and the *accumulated rebuffering times* (Figure 5) to grade the service.

The buffering time represents the time required to accumulate the minimum amount of data to start video playback, after the user presses the *play* button on the player. On the other hand, the accumulated rebuffering times are the sum of the times required to accumulate the minimum amount of data to restart the video playback after each

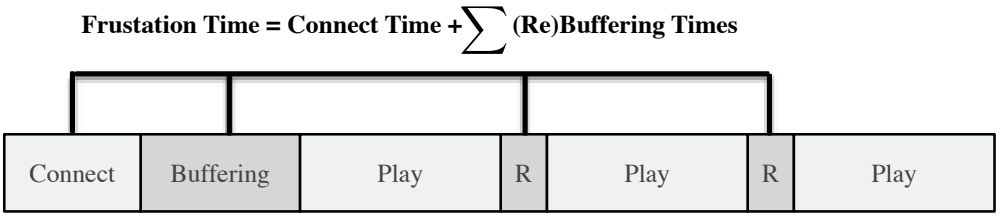


Figure 5: Calculation of frustration times in the Keynote StreamQ metric.

playback interruption due to the lack of buffered data. The Keynote StreamQ Grade attributes the best grade (A+) to frustration times below 6 seconds, and lower grades are classified in periods of 3 seconds above 6 seconds, being F− the lowest grade.

2.5 VIDEO-STREAMING TECHNOLOGIES

Video-streaming services can be classified as Live Streaming and Video-on-Demand (VoD), according to the source of the content.

In Live Streaming services, the video is captured at the source by a camera and is broadcasted to all the client subscribers. The direct transmission of a soccer match is an example of the application of Live Streaming services. The elements of a Live Streaming service are usually a camera, an encoder to digitize the content, a media publisher, and a delivery network to distribute and deliver the content.

In VoD services, the video content is stored in a storage device and is transmitted when end-users request it. Popular Internet services as Youtube [you 2012] and Vimeo [vim 2012] are VoD services. The self-healing infrastructures presented in this thesis are conceived for VoD systems.

2.5.1 *Delivery of Video Content in VoD Services*

Figure 6 presents the elements of a VoD video-streaming service. At the source, the video content is stored in storage devices (e.g., hard disks). When the server receives a user request, the video data is retrieved from the disk to the main memory, where are buffered waiting for its turn to be transmitted over the network. At the client side, the video data are buffered until its turn to be decoded for playback.

Services that deliver video-streaming content are characterized as follows:

TIME DELIVERY REQUIREMENTS Video-streaming services have soft-real-time delivery requirements [Krishna 2001]. That means that data and time integrities should be preserved during the transmission of request-responses, while reducing the delay as much as possible. In VoD applications, the soft-real-time delivery means that the service is more tolerable to longer start-up delays than real-time applications, as long as smooth playback is maintained after the playback has started.

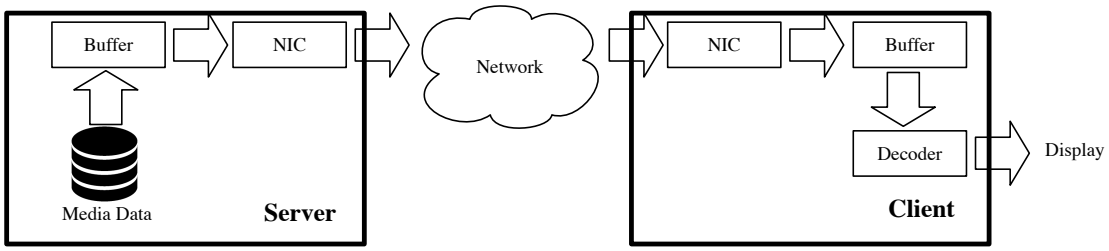


Figure 6: Elements of a VoD video-streaming service.

STREAMING VS DOWNLOAD The relationship between the transmission of data with their consumption is different in download and streaming models. In the typical download model, the client application (e.g., web browser) retrieves the data objects from the server (e.g., web server) and decodes and displays them to the user after completely receiving them. By contrary, in the streaming model, the video is played back simultaneously with the transmission of data between the server and players. Accordingly, after sending a request of the video to the server, the client waits for the reception of the first segment(s) of data and then begins playback while receiving the remaining data. Hence, the data transfer and the playback processes are pipelined to shorten the delay before beginning the playback, as illustrated in Figure 7.

APPLICATION-LEVEL PROTOCOLS The choice of the data delivery protocol adopted for video-streaming services dictates the flow of data and client-server interactions during the data transmission period. Traditional video-streaming services adopt protocols designed specifically for video-streaming services, such as the RTSP protocol. However, most of the modern Internet video-streaming services use the standard HTTP protocol to deliver video data.

The design of the video content delivery infrastructures is conditioned by the application-level protocol adopted, as will be explained in the next sections.

2.5.2 *Pure Streaming*

Traditional video-streaming services are implemented using Pure Streaming techniques and protocols. Pure Streaming is also known as RTSP Streaming, since RTSP is the most popular Pure Streaming protocol for streaming controls [1]. Pure Streaming protocols are designed specifically for streaming data over the Internet.

1 Playback control commands, such as *play*, *pause* and *stop*.

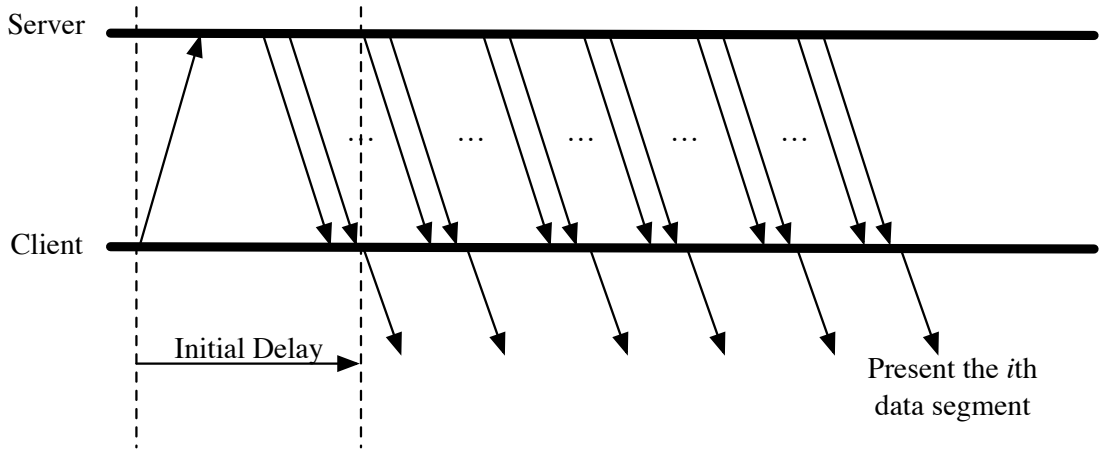


Figure 7: Pipeline between the transmission of video segments and their playback, in video-streaming services.

2.5.2.1 *RTSP Workflow*

RTSP is a stateful and session-oriented video-streaming protocol. After session establishment, the server controls the stream of packets to the client until the end of the video or until the session is interrupted by the client. Both the RTSP server and the client can issue RTSP requests.

Figure 8 presents the RTSP commands issued between the client and the server during a typical video-streaming session. The client starts by issuing a DESCRIBE command to obtain the *presentation description file*, typically in the Session Description Protocol (SDP) format. This file lists the media streams controlled with the aggregate URL. Alternatively, this file can be provided by a web server, while the media data is provided by the RTSP streaming server [Lee 2005].

In the typical case, there is one media stream each for audio and video. A SETUP request is issued for each media stream after the DESCRIBE request. Each SETUP request specifies how a single media stream is transported and contains the local port for receiving RTP data (audio or video) and another for RTCP data. The PLAY request follows the SETUP requests and causes the streams to be played. During playback, the client can issue PAUSE requests to temporarily halt streams. Finally, the TEARDOWN request terminates the session by stopping the streams and freeing all session related data on the server.

2.5.2.2 *Companion Protocols of RTSP*

The RTP [Schulzarine et al. 1996] protocol is used with RTSP for transporting media streams (audio and video). RTP implements strict data rate control policies for providing the transmission bitrates required by players for playback. This protocol provides

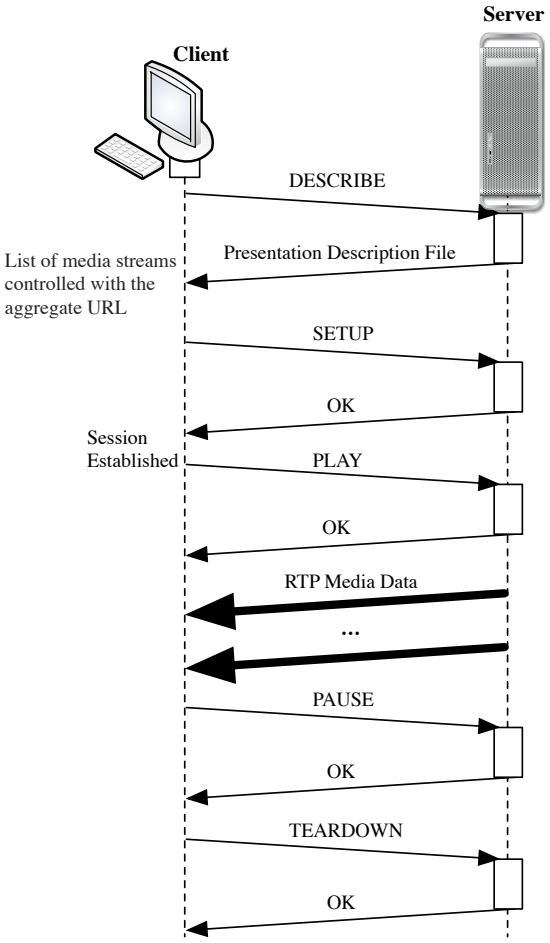


Figure 8: Sequence of commands issued during a video-streaming session using the RTSP protocol.

facilities for jitter compensation and detection of out of sequence arrival in data, which are independent of the lower-layer protocol that carries the RTP packets (TCP or UDP).

RTP is accompanied by the RTP Control Protocol (RTCP) to provide control functions (e.g., synchronization, reporting and data reception statistics). RTCP reports are exchanged between players and the server and include transmission statistics, such as the transmitted octets, packet counts, packet delay variations, packet losses and round-trip delay times. The RTP protocol uses these statistics to adjust the transmission of media data to players.

2.5.2.3 *Buffering*

Player buffers in Pure Streaming services are usually dimensioned only to compensate expected variances on data transmission delays. Therefore, client-side storage require-

ments are minimized and the waste of server resources and network bandwidth is also reduced, since the data are transmitted at the same pace they are consumed by players. This characteristic can be particularly relevant in long videos with high rates of playback abandonment by users or when users perform *time seek* [2] interactions. In such scenarios the efficiency of server, network and client resources is leveraged, since the amount of data downloaded is roughly the same as that used for playback.

2.5.3 *HTTP Streaming*

Modern Internet services eschewed the adoption of streaming-dedicated protocols in favor of the simplicity of the HTTP pull-based protocol. HTTP benefits from the:

- Permeability of HTTP traffic to traverse firewalls;
- Reuse of ubiquitous infrastructures of web servers, caches and Content Distribution Networks (CDNs) to deliver video content in the Internet;
- Ubiquity of web browsers, which are frequently used for displaying media contents;
- The adoption of the recent HTML5 standard for video-streaming technologies.

HTTP Streaming servers are usually standard web servers that are agnostic of video content semantics. Figure 9 shows the architecture of a typical video-streaming delivery service, covering the:

- Preparation of the media according to the service provided. In this stage, the service quality desired and the clients' device characteristics (desktop computer, tablet, smartphone) determine the encoding profiles of the videos provided by the service;
- Storage of media in the origin servers;
- HTTP caches, which could be CDN caching servers or caching proxies on the client's network. This architectural layer could help reduce the initial delay of video playback (particularly important for time seek operations) and reduce the consumption of server resources.

HTTP Streaming services are coarsely classified into *Progressive Download* and *Adaptive Bitrate* services.

2 Seeks the video to any position for playback.

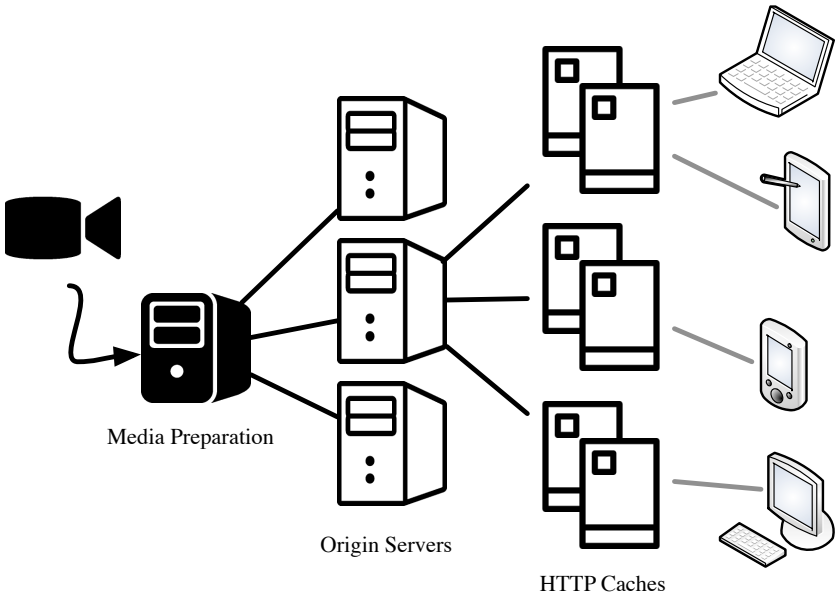


Figure 9: Architecture of a typical HTTP Streaming service.

2.5.3.1 *Progressive Download*

In Progressive Download services, the request-responses are transmitted by the server until the end of the file — similarly to any other static web resource — or until the user stops the transmission explicitly. At the player-side, once the minimum buffered data is obtained, the client starts playing the media simultaneously with the download of the video from the server in the background. This behavior contrasts with the typical download model, which requires the complete download of the video before it can be displayed.

HTTP Streaming servers, in general, deliver video objects without considering their encoding bitrates and consequently, their data rate demand for guaranteeing the continuity of video playback. Thus, the server transmits video content to players as fast as the underlying infrastructure allows. Consequently, the transmission bitrates of request-responses can surpass considerably their respective playback bitrates (equivalent to the encoding bitrates). This characteristic imposes large player buffers for storing video content client-side in Progressive Download services.

Due to their large client-side buffers, Progressive Download services are also probabilistically more resilient to performance fluctuations that affect temporarily the service than Pure Streaming services. The impact that performance fluctuations have on the service quality experienced by end-users depends on the volume of downloaded data stored in client-side buffers before performance degradation is observed. Since HTTP Streaming allows higher transmission data rates, the request-responses that

have been running for longer periods before problems arise (older requests), have a smaller probability of generating service quality issues.

2.5.3.2 *Adaptive Bitrate Streaming*

The simplicity of Progressive Download in HTTP Streaming has some limitations:

- Both the client and server applications lack control over the video downloaded;
- The download process is inefficient, since the streaming users that abandon visualization of videos or seek another playback position in time can download significantly more content than that effectively played;
- The service does not provide adaptation to specific network conditions and client-side resources. As an example, the network bandwidth can be insufficient to maintain the player buffer with enough data for playback.

Adaptive Bitrate (ABR) streaming technologies [Stockhammer 2011] are becoming default technologies for delivering videos over the Internet. They represent a subclass of HTTP Streaming that integrates the benefits of Progressive Download with the efficiency and adaptation features of dedicated Pure Streaming protocols.

ABR video files are decomposed into downloadable segments with short durations that can be requested individually, as illustrated in Figure 10. Thus, instead of requesting the download of the whole file, players can request video segments progressively, according to the progress of playback. This download strategy reduces the waste of server, network and client resources when end-users abandon visualization of videos after a short period of time or when they perform time seek operations.

ABR adds extra complexity to players, but works with low-complex video servers. The reason is that ABR players are in control of service monitoring and adaptation to end-users with different bandwidths and system resources. ABR players select the video encoding profile dynamically during playback, for obtaining the highest video quality, the quicker time seeking or the faster start-up.

ABR videos are encoded at several different bitrate streams, each segmented into small multi-second segments. Therefore, streaming clients can switch between different bitrate streams dynamically, according to their available local resources and network bandwidth (Figure 11). Accordingly, before requesting the next video segment during playback, the player decides the corresponding bitrate with the support of a Media Presentation Description (MPD) file provided by the server.

The MPD file provides the information of each segment, as the timing, URL and video characteristics (e.g., video resolution and bitrates). Since video segments are available at multiple bitrates, then each client decides for the next best segment listed in the MPD to retrieve, by examining several network and device parameters (e.g.,

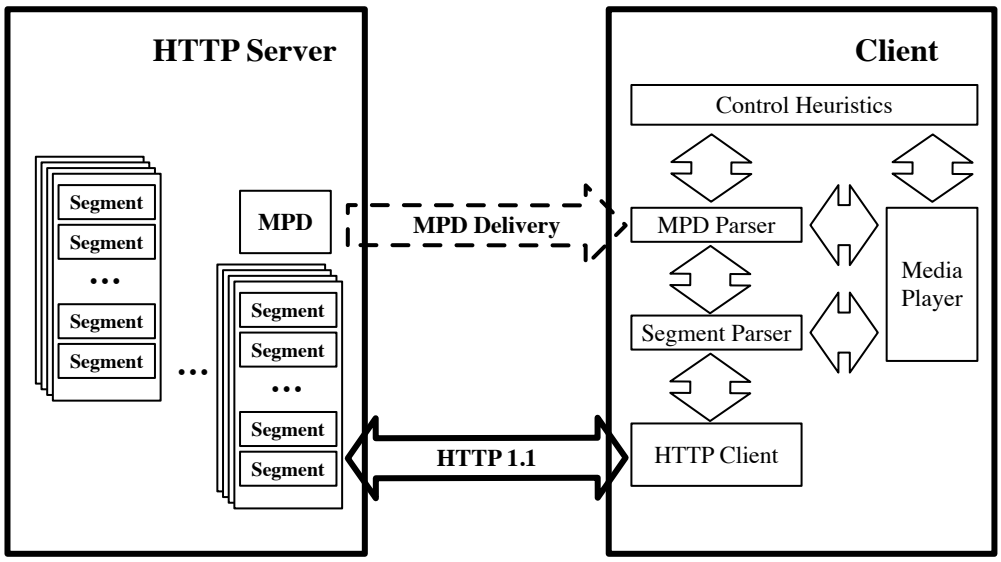


Figure 10: Workflow of ABR media delivery.

available bandwidth, state of TCP connections, display resolution, available CPU and the size of the playback buffer).

ABR video segmentation can follow two approaches:

- **Each video is encoded into a single file** — each request specifies the time range interval of the video for downloading, which is mapped onto the file range offset to be retrieved;
- **Each video segment is encoded into a separate file** — each request specifies the file corresponding to the video segment it references.

The segmentation approach that fragments the video into several files has a similar performance to that of the segmentation using a single file [Summers et al. 2012a]. However, the approach that stores each segment into a independent file, reduces the server-side implementation complexity, since the service can be provided by standard web servers without plugins. Moreover, it reuses the existing HTTP web caches that are prepared to store web resource files. On the other hand, encoding each video segment into a separate file increases the file management complexity, seeing that each video is spread over a large number of files, each of them containing one file segment encoded with a specific quality.

ABR has several implementations. MPEG-DASH [Sodagar 2011] is the only ABR implementation that is an international standard. Other commercial implementations are Microsoft Smooth Streaming, Adobe Dynamic Streaming and Apple HTTP Adaptive Streaming [Stockhammer 2011].

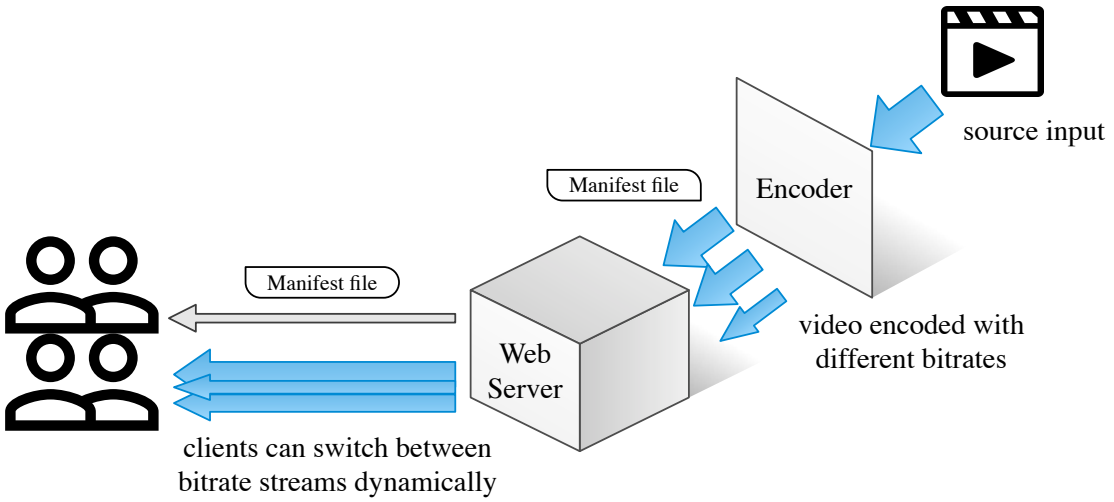


Figure 11: Dynamic switching between video segments encoded with different bitrates.

2.6 FAILURE CONTEXT IN VOD SERVICES

Failures in VoD services can occur at several levels of the video distribution lifecycle, namely:

1. **Encoding** — failures introduced during the video encoding process;
2. **Storage** — disk hardware failures;
3. **Server** — fail-stop and performance failures;
4. **Server-side infrastructure** — failures in infrastructure services (e.g., load balancer);
5. **Network** — connection failures and performance failures (e.g., packet losses and delayed packets);
6. **Client-side** — failures caused by faults originated client-side.

A thorough review of the literature about techniques for handling failures in the different levels of the video distribution lifecycle is presented in Chapter 3.

2.7 VIRTUALIZATION TECHNOLOGIES

The self-healing approaches proposed in this thesis exploit virtualization for performance isolation between the Autonomic Manager and the Managed Element, and

to support recovery through migration of servers between hosts. Virtualization techniques can be classified into two broad categories: *hypervisor virtualization* and *container-based virtualization* [Soltesz et al. 2007].

Hypervisor virtualization, implemented by *full virtualization* or *paravirtualization* technologies [Sahoo et al. 2010], requires the installation of one dedicated operating system instance within each virtual machine, as shown in Figure 12. Consequently, in server migration scenarios, all in-memory data of the operating system and server applications should be checkpointed, transferred and reinstantiated in the destination server, generating significant overheads.

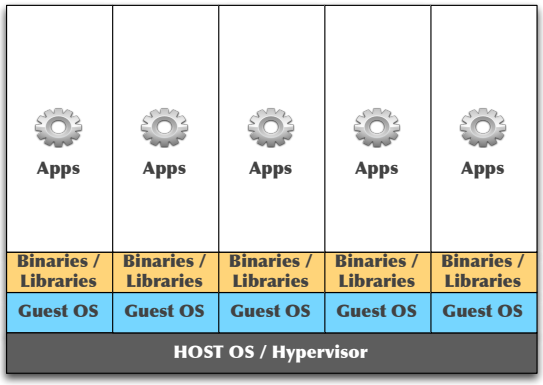


Figure 12: Traditional hypervisor virtualization, implemented by full virtualization and paravirtualization technologies. It requires one full operating system image for each virtual machine instance.

Container-based virtualization technologies use a single operating system instance for all virtual containers sharing the same machine, generating significantly smaller overheads than hypervisor virtualization technologies [Padala et al. 2007][Che et al. 2010]. Container-based virtualization is an operating system-level virtualization approach that allows several isolated user-space instances with their own network addresses, called virtual containers, to run on top of the operating system, as illustrated in Figure 13. The operating system is installed natively on each host machine and the virtualization layer is placed in the operating system kernel to guarantee isolation of processes running in different virtual containers and fair scheduling on utilization of resources by virtual containers. Each virtual container holds a set of processes virtualized with its private memory address space, logical file system and virtual network interface.

The advantages of container-based virtualization over hypervisor virtualization can be resumed to the following:

- **Small overheads** — container-based virtualization adds smaller resource footprints than hypervisor virtualization, as several virtual containers share the same operating system kernel;

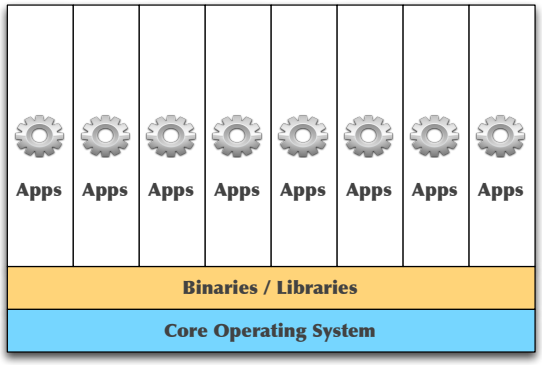


Figure 13: Container-based virtualization shares a single operating system between virtual containers, and in some cases also shares binary and library resources.

- **Small migration costs** — the size of the virtual container is significantly smaller than a typical virtual machine, because the operating system runs natively decoupled from virtual containers. For that reason, the memory addressed by a virtual container is significantly smaller than by a typical virtual machine. Consequently, less data have to be checkpointed, transferred and restored during migration of servers between hosts;

- **Error isolation** — healthy virtual containers are isolated from errors originated within other virtual containers running in the same host. Since propagation of errors between virtual containers is avoided, server migration techniques can be exploited to rescue the healthy virtual containers during faulty periods. As well, container-based virtualization allows rebooting faulty virtual containers independently of either the operating system or the other virtual containers running in the same host;

- **Efficient utilization of resources**. System resources unused by a virtual container can be used by other virtual containers and are only reclaimed by the legitimate virtual container when needed.

Due to the small overheads of container-based virtualization, it can be also combined with hypervisor virtualization to: (1) guarantee performance isolation between applications running within the virtual machine; and (2) perform selective migration of applications between virtual machines.

2.8 CHAPTER SUMMARY

This chapter presented the research domain and general concepts that support the research developed in this thesis. It provided the background on Autonomic Computing

and their aspects, focusing on self-healing, which is a main topic of this thesis. Then, it described the taxonomy of failures along with their characterization in software systems, and specifically in video-streaming services. The architectures, technologies and protocols adopted by traditional and modern video-streaming services have been also described. Finally, it described the characteristics of the virtualization technologies adopted for the development of our self-healing infrastructures.

The vocabulary and concepts presented in this chapter represent the background necessary to understand the related work of our research domain presented in the next chapter, and our work presented in the following chapters of this thesis.

3

STATE OF THE ART

There is an extensive work related to the self-healing area, from complete architectures and infrastructures, to research dealing with specific self-healing activities: *offline fault analysis*[1], *monitoring*[2], *failure prediction*, *failure diagnosis* and *recovery*.

This chapter provides a state of the art on the self-healing and dependability areas related with the research conducted in this thesis. It is structured into three main parts. Firstly, it reviews the literature covering self-healing and dependable systems in general, to provide an overview of the research area. Secondly, it reviews the work on video-streaming dependability, which directly relates to our work. Finally, it presents the gaps on the literature that we fill with our work.

Table 1 presents the list of relevant publications on self-healing architectures, infrastructures and techniques, organized by research domain and self-healing activities addressed. Each of these publications is described in the next sections.

3.1 SELF-HEALING SYSTEMS AND DEPENDABILITY

This section reviews the generic literature on self-healing systems and dependability areas. It revisits the work that contextualizes the research presented in this thesis.

3.1.1 *Software Engineering*

Software engineering techniques have been applied to self-healing and dependable systems at the design and runtime levels. At the design level, they help building software systems that allow recovering their state after the occurrence of failures caused by software faults or by operations performed by human operators by accident — one of the main causes of failures [Oppenheimer et al. 2003]. At runtime, techniques like fault injection are popular in software engineering for exposing the properties of a system, which allows the understanding of how the system behaves under stressed or faulty conditions.

3.1.1.1 *Recovery Oriented Computing*

Recovery Oriented Computing (ROC) [Patterson et al. 2002] is an important research area overlapping with the self-healing requirements. ROC exploits reduction of the

1 Dynamic and static analysis.
2 Including data gathering and failure detection.

Areas	Project/Approach	Offline / Static Analysis	Monitoring a)	Failure Prediction / Anomaly Detection	Failure Diagnosis / Localization	Recovery / Adaptation	Overview and Concepts
Software Engineering	Dependability evaluation in distributed real-time systems [Han et al. 1995]	✔					
	Processor-level fault injection for evaluation of software [Carreira et al. 1998]	✔					
	Validation of the fault-tolerance mechanisms [Arlat et al. 1990]	✔					
	FIG (Fault Injection in glibc) [Patterson et al. 2002a]	✔					
	Recursive Restartability [Candea and Fox 2001a][Candea and Fox 2001b]					✔	
	Recovery Oriented Computing [Patterson et al. 2002a]					✔	✔
	Reversible Systems for Operators [Patterson et al. 2002a]					✔	
	Detection of Web application vulnerabilities [Jovanovic et al. 2006]	✔					
	Automatic workarounds in Web applications [Carzaniga et al. 2008]		✔			✔	
	Instrumentation of applications [Portokalidis et al. 2011]	✔	✔			✔	
	Feedback loops for handling Runtime Exceptions [Gaudin et al. 2011]	✔	✔			✔	
	AOP for generation of code assertions from UML stereotypes [Gorla et al. 2012]	✔	✔		✔	✔	
	Automated recovery of smartphone apps using patching [Azim et al. 2014]	✔				✔	
Statistical Analysis and Data Mining	Anomaly detection in UNIX programs [Forrest et al. 1996]		✔	✔			
	Dynamic identification of dependencies [Brown et al. 2001]				✔		
	Time-based and prediction-based rejuvenation [Castelli et al. 2001]			✔			
	Pinpoint [Chen et al. 2002]		✔	✔	✔		
	Computing optimal rejuvenation schedules [Li et al. 2002]			✔			
	Failure prediction and diagnosis using Time Series [Sahoo et al. 2003]			✔	✔		
	Diagnosis and forecasting of performance states [Cohen et al. 2004]			✔	✔		
	Operator involvement in Failure Detection and Localization [Bodik et al. 2005]		✔		✔		
	Anomaly detection/diagnosis in global applications [Kelly 2005]		✔	✔	✔		
	Forecasting system performance in enterprise systems [Powers et al. 2005]			✔			
	Failure prediction in clusters of scientific applications [Liang et al. 2006]			✔			
	Diagnosis of failures using process traces [Mirgorodskiy et al. 2006]				✔		
	Trend detection and estimation of resources utilization [Grottke et al. 2006]			✔			
	Anomaly detection/diagnosis in Grids using the Fourier [Yang et al. 2007]			✔	✔		
	Diagnosis in Grids using the Otho Toolkit [Hofer et al. 2007]				✔		
	Prediction of response times/free memory [Hoffmann 2007]			✔			
	Rejuvenation in VOIP services [Koutras et al. 2007]			✔			
	Anomaly detection/diagnosis in 3-tier web servers [Cherkasova et al. 2008]		✔	✔	✔		
	Failure prediction in data stream nodes [Gu et al. 2009]			✔			
	Just-in-time anomaly prediction in stream processing clusters [Tan et al. 2010]			✔			
	Anomaly detection in computational nodes [Lan et al. 2010]		✔	✔			
	Consequence-oriented self-healing [Dai et al. 2011]				✔		
	Performance diagnosis through code level analysis [Attariyan et al. 2012]	✔					
	Detection of features misbehavior [Schneider et al. 2014]			✔	✔		
Self-adaptive Systems	DARPA [Laddaga 1997]						✔
	Self-adaption principles and challenges [Laddaga 1999]						✔
	Framework for Adaptive Systems [Cheng et al. 2002]		✔		✔	✔	
	Self-adaptation in legacy systems [Kaiser et al. 2002]		✔		✔	✔	
	Automatic generation of recovery descriptions[Dashofy et al. 2002]					✔	
	Architectural adaptation [Garlan and Schmerl 2002]		✔			✔	
	Rainbow framework [Garlan et al. 2004]		✔		✔	✔	
	Software self-awareness [Robertson and Laddaga 2005]		✔			✔	
	MAPE Cycle in SOA [Gurguis and Zeid 2005]		✔		✔	✔	
	QoS-oriented reconfigurable middleware [Ben Halima et al. 2008]		✔	✔	✔	✔	
	VieCure framework [Psaier et al. 2010]		✔		✔	✔	
	Autonomic Management of Application Workflows [Kim et al. 2011]		✔			✔	
	QoS architectural patterns [Menasce et al. 2011]		✔			✔	
	Adaptive self-testing applications [King et al. 2011]					✔	
	Context-aware feedback loops in service-oriented systems [Villegas et al. 2013]		✔			✔	
Multi-agent and Evolutionary Systems	Multi-algorithm redundancy [Huhns et al. 2003]		✔			✔	
	Self-assembly of structures of agents [Nagpal et al. 2003]		✔			✔	
	Programming paradigm based on cell division [George et al. 2003]					✔	
	Swarm Intelligence [Dai et al. 2006]		✔		✔	✔	
	Self-healing using embryonic models [Miorandi et al. 2010]		✔			✔	

a) Data Gathering and Failure Detection

Table 1: Self-healing architectures, infrastructures and related techniques found in previous work.

Mean Time to Repair (MTTR), as a strategy for increasing availability. It has roots on the assumption that the increase of the Mean Time To Failure (MTTF) has the same impact on availability as the reduction of the MTTR, considering that unavailability is approximately MTTR/MTTF. ROC inspired several lines of research, such as fault injection, error determination and the design of applications to tolerate rebooting of system components and undo operations executed by operators.

Recursive Restartability [Candea and Fox 2001b][Candea and Fox 2001a] is a recovery approach based on the design of systems that gracefully tolerate successive restarts at multiple levels. By rebooting or restarting partially the system, it is possible to avoid bugs leading software applications to crash, deadlock, spin, leak memory, or fail in a way that leaves restart or reboot the unique way to restore the system.

Recursive Restartability also allows strong fault containment. Dependencies between components are organized in a hierarchy of restartable components in which nodes are highly failure-isolated. When one tree node (i.e., component) is restarted, the entire subtree rooted at that node (dependent components) is restarted with. Identification of faulty components is performed using an approach similar to Pinpoint [Chen et al. 2002].

Reversible systems for operators is another research topic explored by ROC. These systems allow recovery from operator errors, one of the main causes of service failures today. They provide operators a mean to retroactively repair latent errors that went undetected until too late, using a 3-step undo process: *rewind*, *repair*, and *replay*. The rewind step rolls back the virtual state. In the repair step, the operator can perform the changes to the system. In the replay step, all user interactions with the system are executed again with the changes performed in the repair step. An undoable email system capable of recovering from errors caused by accidentally deleting messages or by a virus or spam attack is presented in [Patterson et al. 2002].

3.1.1.2 *Fault Injection*

Research on fault injection is vast. Relevant work in this area includes validation of fault-tolerance mechanisms [Arlat et al. 1990], detection of Web application vulnerabilities [Jovanovic et al. 2006], processor-level fault injection for evaluation of software [Carreira et al. 1998], and dependability evaluation in distributed real-time systems [Han et al. 1995].

One important application of fault injection in software engineering is the identification of common patterns of software faults. These patterns lead to the development of software development practices to improve the design of dependable software. The Fault Injection in Glibc (FIG) [Patterson et al. 2002] is an example of a lightweight tool for triggering and logging errors at the application/system boundary. The application of this tool to the evaluation of several popular applications (Emacs, Netscape browser, Berkeley DB database library, MySQL and Apache web server) has exposed effective

practices for programming dependable applications. Examples of these practices are *resource preallocation*, *graceful degradation*, *selective retry* and *process pools*.

3.1.1.3 *Code-level approaches*

Code-level self-healing activities can be incorporated into software at the design time or through instrumentation of existing applications.

In [Gorla et al. 2012], it is addressed the design of applications with mechanisms for detection, localization and recovery of failures. Failure detection uses UML model stereotypes that are mapped onto assertions encapsulated in aspects[3], which are inserted into the system using dynamic load-time weaving. Fault localization is performed through analysis of sequences of method invocations and data exchanged during interactions, to infer models that summarize and generalize the observed executions. Additionally, recovery is performed through workarounds, a sequence of correct operations that is specification-equivalent to the sequence of operations that leads to a failure.

The use of automatic workarounds for self-repair was also explored for Web applications [Carzaniga et al. 2008]. Workarounds are automatically generated sequences of service invocations that have the same intended effect as the failing sequence, determined according to the specifications. These sequences are executed for failure recovery, after the server has been brought back to an internally consistent state. The approach was successfully evaluated using failing sequences of invocations to the Flickr and Google Maps services.

A failure detection and recovery approach for Android applications that seals off the crashing part of applications to avoid future crashes is presented in [Azim et al. 2014]. Recovery relies on bytecode rewriting using static analysis to build models that represent discrete, safe and unsafe points in the app, and transitions between them. After failure occurrence, the app is restarted to a nearby safe point, while the bytecodes associated with the crash point are rewritten.

[Gaudin et al. 2011] proposed a control theory approach for automatically disabling system functionalities that have led to runtime exceptions. The system is instrumented prior to deployment, so that it can later interact with a supervisor that encodes the sequences of actions allowed for the system.

REASSURE [Portokalidis and Keromytis 2011] implements recovery of applications by introducing rescue points in specific code locations. Rescue points are identified and intercepted through instrumentation of the application, to gracefully handle unexpected errors by returning an error code.

3 The main abstraction mechanism of Aspect-Oriented Programming.

3.1.2 *Statistical Learning and Data Mining Approaches*

Statistical learning and data mining areas provide powerful techniques for prediction, detection and diagnosis of failures. These techniques have been applied to performance anomalies in general, to the specific area of software aging and also to prediction of performance in non-faulty scenarios.

3.1.2.1 *Detection and Diagnosis of Performance Anomalies*

Detection and diagnosis of performance anomalies have been studied extensively for both generic systems and for systems supporting specific services.

An approach for detection of anomalies in computational nodes is proposed in [Lan et al. 2010]. Anomalies are detected via automated analysis of system data, using Principal Component Analysis (PCA) and Independent Component Analysis (ICA) for feature extraction and anomaly identification. These techniques were successfully evaluated through fault injection of CPU, memory, I/O and network faults and also deadlock faults.

An offline anomaly detection method for standard UNIX programs is presented in [Forrest et al. 1996]. It scans traces of normal behavior to build a database of characteristic normal patterns (sequences of system calls). Latter, the database is used to scan for new traces that might contain abnormal behavior, recognized as patterns not present in the database.

The problem of detecting and diagnosing performance anomalies in Grid environments is addressed in [Yang et al. 2007]. The Fourier transform is explored to filter periodic patterns of resources utilization. The diagnosis activity is based on a comparison of each resource utilization observed with the expected consumption provided by the baseline model. Experimental results show several improvements over simple window average strategy.

An approach for detection and diagnosis of performance anomalies for separating server overloading states from application logic and configuration faults is presented in [Kelly 2005]. It uses multivariate regression models of aggregate response times from the transaction mix to discriminate between anomaly types. The approach was successfully evaluated using three large data sets collected in global distributed systems.

Failure prediction in clusters of scientific applications is addressed in [Liang et al. 2006]. That work explores temporal and spacial locality of past failures to predict future failures and uses information about non-fatal events to predict application crashes. Transitions between server states according to specific probabilities are modeled using sequential patterns.

Time series methods, rule-based classification and bayesian network models are applied in [Sahoo et al. 2003] to prediction of anomalous events in commercial and scientific applications. Results show that time series are more accurate to predict continuous

variables than to predict event characteristic variables and bayesian networks are better for modeling dependencies between variables for root cause analysis.

Diagnosis of anomalies in Grid services is addressed in [Hofer and Fahringer 2007], using customized wrapper services synthesized by the Otho Toolkit. The approach relies on models formulated by the support staff or application developers to perform fault diagnosis.

In [Mirgorodskiy et al. 2006], it is proposed an approach for localization of anomalies through dynamic instrumentation and analysis of system processes that stop earlier or behave differently than the rest. Anomalies are localized by looking at differences in the control flow across processes. The faulty process is identified as the process that stopped generating traces first or that had shown outlier traces. The cause of the anomaly is determined through the analysis of the last trace entry (i.e., last function called). The approach was validated in a real-world distributed environment.

Root cause analysis at the code level is explored in [Attariyan et al. 2012], for diagnosis of performance anomalies. Diagnosis is implemented by instrumenting binaries at runtime, for tracking dynamic information flows. These flows are used to estimate the likelihood that a block of the application is executed due to each potential root cause.

Performance anomalies in web applications is a research topic studied extensively. Pinpoint [Chen et al. 2002] is a diagnostic tool that relies on data mining techniques to identify combinations of components that are most highly correlated with request failures. This tool monitors request traces automatically without beforehand knowledge of request paths. Request paths are determined by instrumenting the J2EE middleware, to see which components instances are traversed by each request. Faulty modules are pinpointed by analyzing the components used by failed requests, but not used by successful requests. External failure detection detects end-to-end failures, whereas internal failure detection captures HTTP and TCP failures masked by the system, with the support of a packet sniffer tool.

Statistical analysis techniques and visualization tools are proposed in [Bodik et al. 2005] to help operators detecting and diagnosing performance failures. Their methodology relies on learning a baseline of web page hit frequencies and detecting deviations from that baseline. The baseline represents the model of the users' normal behavior. Therefore, the efficacy of the approach is supported by the assumption that the end-users' behavior changes when they encounter a malfunction.

In [Cherkasova et al. 2008], it is proposed an approach for detection of workload changes, performance anomalies and application changes in three-tier web servers. The performance model is based on regression of the CPU consumption, and the runtime behavior is reflected by application performance signatures. The approach is able to accurately detect anomalies as changes in the CPU consumption pattern of the application and identify the transactions responsible for the anomalies using the performance signatures.

Tree Augmented Based Networks (TANs) are explored in [Cohen et al. 2004] to identify correlations between groups of system-level metrics and high-level performance states. These correlations are applied to forecasting and diagnosis of failures in three-tier network services. The interpretability and modifiability properties inherent to TANs are important criteria for their adoption, since they allow human inspection and analysis of models. The experimental results have shown better performance of TANs for diagnosis than for forecasting.

An active dependency discovery technique for diagnosis of web commerce distributed applications is proposed in [Brown et al. 2001]. The approach determines dependencies between request types and components by perturbing the system with specific workloads and applying statistical modeling techniques to compute dependency strengths.

Prediction of performance anomalies in data streaming services was investigated in [Gu and Wang 2009]. It studies the combination of Bayesian classifiers with Markov models to predict bottleneck failures (insufficient CPU, insufficient memory and memory leaks) in distributed data stream nodes. Markov models are applied to forecasting of feature values, used afterwards by Bayesian classifiers to identify symptoms of failure. Additionally, context-aware models are explored to group anomalous occurrences into contexts.

In [Tan et al. 2010], the authors investigated the problem of performing just-in-time anomaly prediction in stream processing clusters. A context-aware anomaly prediction scheme is proposed to avoid redundant learning, combined with decision trees to classify component states. The approach exhibited better performance than monolithic, incremental and ensemble approaches, for several types of stream processing components with real-time performance requirements.

A diagnosis tool that combines Multi-variate Decision Diagrams (MDDs), Fuzzy Logic, and Neural Networks is presented in [Dai et al. 2011]. MDDs determine failure severity levels and are complemented with Fuzzy Logic to infer the possible consequences. Neural network technology is applied to train the fuzzy logic inference.

Identification of behavioral changes in software features is addressed in [Schneider et al. 2014]. The normal behavior of each feature is modeled by Hidden Markov Models and Artificial Neural Networks, using historical performance and configuration data periodically gathered from the system. The approach was evaluated experimentally, using configuration faults and direct fault injections responsible for crashing the service. Experimental results show that both types of models are able to reliably detect faults and generate an accurate ordered list of potential root causes.

3.1.2.2 *Software Aging*

The related work on software aging is vast. This research topic focuses mainly on the analysis of consumption of resources to determine the system degradation level. System degradation is caused by software faults leading to the accrual of errors, mostly

related with the allocation of resources without appropriate further deallocation. In that case, the portion of allocated resources not being used increases along time, contributing to system performance degradation.

Software aging solutions are evaluated according to their ability to quantify levels of degradation of resources and to forecast these levels for determining the appropriate moment to apply software rejuvenation techniques. Rejuvenation employs reboot-based recovery to restore the system to a known good state without degradation.

In [Hoffmann et al. 2007], it is explored the use of statistical regression techniques to predict the response time and the amount of free physical memory of an Apache web server system, using numerical system metrics/parameters. Experimental results show that Universal Basis Functions (UBFs) yield the best results for free physical memory prediction and Support Vector Machines (SVMs) perform better predicting server response times.

The problems of trend detection and estimation of resource utilization in web servers, using non-parametric statistical methods, are studied in [Grottke et al. 2006]. This work combines time series models to support prediction of resource usage using parameters with seasonal patterns. The authors start by testing the statistical independence of data sets using the Mann-Kendall test for trend, and then calculates the slope estimates using the Sen's estimator method. A modified Man-Kendall test addresses seasonal data captured from parameters with seasonal behavior, which is followed by time series analysis to determine sudden local increases that may lead to resource exhaustion. An autoregressive model is created for parameters showing both seasonal and non-seasonal patterns.

Trend analysis of performance degradation in web servers is addressed in [Li et al. 2002], for computing optimal rejuvenation schedules that maximize availability and minimize downtime cost. Parameters estimation is performed through linear regression and ARX models that fit the time series data captured during system-level monitoring. Additionally, the Sen's method estimates the performance degradation slope for each observed resource. A comparison between models shows that ARX models performs better for the estimation of resource exhaustion and are computationally less intensive than linear regression models.

Detection of software aging and estimation of the remaining time until exhaustion of resources are also investigated in [Castelli et al. 2001]. The execution of proactive software rejuvenation for an application, process group or entire operating system is weighted in terms of the downtime generated, for selection of the most appropriate rejuvenation granularity. Analytical models based on Stochastic Reward Nets (SRNs) are applied to both time-based [4] and prediction-based rejuvenation [5], with the goal of maximizing system availability while minimizing cost.

4 Based on the time elapsed since the last rejuvenation.
5 Performed by relying on a variety of indicators of aging.

Software rejuvenation for increasing availability of VOIP services is explored in [Koutras and Platis 2007]. They propose rejuvenation policies indicating when to perform rejuvenation and a semi-Markov process (SMP) to model the time the system enters a new resource degradation level. The VOIP service reliability is evaluated in terms of how the Mean Time To Failure (MTTF) is affected by each rejuvenation policy chosen to maximize the VoIP service availability.

3.1.2.3 *Prediction of Performance*

Statistical analysis of performance data has been explored to predict the system performance with future loads, for load admittance and for planning future resource requirements. This problem is related with the detection of performance anomalies, since both manifest as performance failures. In diagnosis problems, it is often required to determine whether performance failures are caused by software faults or by loads generated by client requests. Therefore, if the system performance degradation is predicted for the organic client requests, then eventual failures are not caused by software faults.

The problem of forecasting performance in enterprise systems for automating the assignment of resources is approached in [Powers et al. 2005]. Forecasting of service level objectives (SLOs) one hour ahead revealed that: (1) Multi-variate Regression and Bayesian Network Classifiers perform better than auto-regression methods; and (2) models are not reusable between machines without accuracy losses, but can be used as bootstrapping models on machines where learning data are scarce.

3.1.3 *Self-Adaptive Software*

Self-adaptive software shares several characteristics with self-healing systems. They both monitor their operations and attempt to correct deviations from the expected behavior. Accordingly, self-adaptive software changes its own behavior when it is not accomplishing what it is intended to do or when its performance can be improved. Self-adaptability is implemented by designing software for having several ways of accomplishing its purpose and having knowledge of its construction (self-awareness) to make changes at runtime.

The DARPA Broad Agency Announcement on Self-Adaptive Software [Laddaga 1997] is one of the most important seminal contributions to research on self-adaptive software. It presents the concepts and principles that have driven the development of self-adaptive software in the last decade. Another seminal publication about self-adaptive systems [Laddaga 1999] pointed software self-evaluation as one of the hardest challenges to self-adaptiveness. The reason is that it is not always intuitively clear how to evaluate functionality and performance at runtime. Other challenges are the

self-evaluation overhead and the lack of adequate metrics for measuring the degree of robustness and adaptation of self-adaptive software.

The principles of self-adaptive systems are explored in [Robertson and Laddaga 2005], using a case study involving systems inspired in the natural vision. The software designed for these case studies has knowledge about the contexts presented by the environment (context-awareness) and meta-knowledge about the state of the program (self-awareness). Therefore, context specific software are synthesized at runtime to adapt to changes in both the environment context and the program state. In the case study, each context is associated to a specific way of interpreting images, whereas transitions between contexts are modeled using Hidden Markov Models.

The related work in self-adaptive software is divided between the adaptation through architectural models and through the use of service-oriented architectures.

3.1.3.1 *Adaptation of Architectural Models*

One of the major research outcomes of self-adaptive software is centered on adaptation of architectural models. Several researchers have proposed architectural models to represent the system as a composition of components. One architectural model sets constraints and properties on the components and connectors, so that it can be used to detect violations on the managed model elements, and thus trigger further adaptations.

A framework for adaptive grid computing written in the domain of architectural models is proposed in [Cheng et al. 2002]. When the framework recognizes the need of adaptation, it follows a repair logic — written in an architectural model language — previously propagated throughout the running system. Data used for analysis are collected using *probes* (the lowest level of abstraction) and are reported using *gauges* to be analyzed according to predefined policies, performance objectives and resource constraints. The use of the framework is illustrated for a load-balancing system.

The Rainbow framework [Garlan et al. 2004] provides architectural-based self-adaptation using a reusable infrastructure, complemented with mechanisms to promote its specialization to specific system requirements. Data are gathered for analysis using probes, whereas features for determining network latencies, available bandwidth and for discovering network resources are implemented by specialized off-the-shelf tools.

An architectural approach similar to Rainbow is presented in [Kaiser et al. 2002] for retrofitting legacy systems with self-healing, self-adaptation and self-management capabilities. The infrastructure is designed to be independent of the running system, with the exception of *probes* and *effectors*, which are specialized to the implementation technology. Additionally, *gauges* and decision mechanisms are specialized to the problem domain and environmental context. The decision and control layer receives information from gauges to decide for the introduction of new modules or for changing system parameters dynamically.

SASSY [Menasce et al. 2011] is a framework for representing system architectures automatically, and dynamically adapt them during runtime to maintain the QoS goals. Architectures are represented by the xADL language [6], using service instances modeled as software components. The self-architecting problem is approached by maximizing the system's overall utility function, subject to a set of constraints (e.g., cost). Adaptation involves the reuse of QoS architectural patterns (one or more components linked by connectors), which capture strategies known to promote QoS attributes.

In [Kim et al. 2011], it is presented a framework for provision of the appropriate mix of resources demanded by application requirements in commodity clusters and clouds. The framework adapts the application and resources to respond to changing requirements or environmental changes in High Performance Computing. Adaptation is performed according to objectives (acceleration, conservation and resilience) while satisfying deadline and budget constraints. The autonomic manager uses a computational model or benchmarks to estimate runtime, and schedules tasks to attain the user objectives within the constraints. Experimental results using a hybrid infrastructure of clusters and clouds, show that the framework can maintain performance and cost objectives and recover from unexpected delays and failures.

Automatic generation of recovery descriptions for specification of architectural reconfigurations is proposed in [Dashofy et al. 2002]. Recovery descriptions are obtained from the differences (*diffs*) between the current and desired architecture. Diffs are architectural patches that are analyzed before being applied to a running system, with the support of *design critics*. Design critics determine whether a patch of an architecture description will create a valid result or not.

Another approach for architectural model-based adaption is proposed in [Garlan and Schmerl 2002]. The adaptation is triggered by violations of constraints established for properties of architectural models. Experimental results show a significant improvement of the system performance using architectural adaptation, when compared with the same infrastructure without it.

A comparative case study performed on three autonomic adaptive applications engineered for self-testing is presented in [King et al. 2011]. The study show up the benefits of having reusable designs, proper tools and frameworks for building self-testable autonomic software. It also identifies the inexperience of the autonomic development and testing teams and the complexity of self-test features as the main threats to the implementation of self-testing software architectures.

3.1.3.2 *Service-oriented Architectures*

Web Services and Service-Oriented Architectures (SOA) are de facto standards for designing distributed applications. However, dealing with the complexity of dependencies between services exposed to permanently changing constraints, both at the

6 xADL is a XML-based architecture description language.

communication and execution levels, is challenging. Self-healing properties can help dealing with such complexity when incorporated in that type of systems, by exploring the dynamic composition of services interconnected via loosely-coupled bindings.

Self-healing is proposed as a web service for SOA environments in [Gurguis and Zeid 2005]. The service implements the MAPE cycle that accompanies the Autonomic Computing original concept. It includes features for discovering, diagnosing and reacting to unexpected events responsible for service malfunctions. Event logs, represented according to the standard Common Base Event Format (CBE) [Ogle et al. 2004], feed the database that maps symptoms to recovery actions.

Self-healing web services are implemented using QoS-Oriented Self-healing middleware (QOSH) in [Ben Halima et al. 2008]. QOSH supports both reactive and proactive self-healing. The latter is implemented through analysis of tendencies in QoS parameters (e.g., continuous increasing response time). Data used for analysis of QoS parameters, in the requester and provider sides, are gathered using metadata fields of SOAP message headers. Diagnosis is implemented through analysis of interactions of each web service with the other web services, to identify the source of degradation. Recovery is implemented through architectural reconfigurations providing single and composed substitutions for the faulty web services.

The VieCure framework performs monitoring, diagnosis and recovery of failures in SOA systems [Psaier et al. 2010]. The self-healing process implemented by the framework covers services provided by human actors and software. Diagnosis is performed by detecting abnormal interaction behaviors (through comparison with previously seen events) between software services, and between human services and software services. Recovery is ensured by each computer node, through control of the throughput of tasks accepted by it and by delegation of tasks to other nodes.

DYNAMICO [Villegas et al. 2013] addresses the management of adaptation properties and goals in service-oriented systems. It focuses on modularization of the feedback loops that manage control objectives over time, context-aware adaptation mechanisms and dynamic monitoring. The applicability of DYNAMICO is demonstrated for a SOA governance application based on an industrial case study, for guaranteeing SLAs in a cloud-based infrastructure.

3.1.4 *Multi-agent Systems*

Agents can be used as building blocks for development of self-healing software. They can be composed dynamically in a system during runtime and are customizable over their lifetime. By design, they can handle unexpected conditions in environments with unpredictable behavior. Being self-aware, multiple cooperative and persistent, agents can adapt to support the overall system objectives.

Multi-agent software have been used to create self-healing software. They assist both conventional software systems and traditional software engineering techniques

to improve the robustness of complex systems. The multi-agent paradigm has several properties aligned with the self-healing requirements, such as high dependability, robustness and high degree of adaptability. Additionally, agent-based programming produces code that is easier to reuse and add to existing systems, seeing that agents are designed to interact with an arbitrary number of other agents.

In [Huhns et al. 2003], the multi-agent paradigm is applied to software development for attaining high levels of robustness, proportionated by multi-algorithm redundancy provided by several agents. Distributed decision making and control using several agents allows detection and correction of inconsistencies in each other's behavior, without a fixed leader or controller. The proposed approach is studied using a case study that involves running several sorting algorithms using several agents.

An approach for engineering self-organizing systems using agents is presented in [Nagpal et al. 2003]. Instead of attributing goals directly to the behavior of individual agents, the authors propose a global decomposition of goals, described at a abstract level, into construction steps to be mapped to local rules implemented by agents. Self-repair of systems is implemented by conceiving agents aware of their other neighboring agents, with the purpose of launching replication when a neighbor agent disappears. Therefore, when an agent fails, it is replaced through reproduction, if there is room at the chosen location for another neighboring agent. Hence, the structure of agents can grow to any size and be automatically maintained through replacement of dying parts.

3.1.5 *Evolutionary Systems*

Self-healing systems have several characteristics encountered in the biological paradigm. One typical analogy for recovering from a error in self-healing systems is, in biological systems, the expectation that the body autonomously recovers from a wound. Research on evolutionary systems, in particular, proportionates important concepts inspired in the biological paradigm with an application into the self-healing domain.

A programming paradigm based on the actions of biological cells is proposed in [George et al. 2003]. The healing concept is built on the cell-division analogy, where cells divide and communicate between each others through diffusion of chemicals within a given radius. The programming paradigm is evaluated using an application-layer peer-to-peer file sharing service running over a wireless network — the service is designed to run on wireless nodes that can be mobile. Results show that movement and death of nodes and network path failures are potentially overcome through cell-division.

Swarm Intelligence is proposed in [Dai et al. 2006] for building self-healing systems. Autonomous diagnosis and curing are consequence-oriented and are implemented in a swarm of robots (computer machines) using prescriptions. Each robot monitors other robots and, in case of detection of faulty behaviors, the underlying problem is

diagnosed using Fuzzy Logic, Bayesian Networks or Hidden Markov Models, to narrow down the possible causes and their domain. The knowledge base used for identification of possible diseases responsible for the symptoms is built up from generic statistics on historical data.

In [Miorandi et al. 2010], it is studied the development of self–healing distributed services, using an architecture inspired on metaphors of biological organisms. Each system is an embryo made of stem cells that develops into a full organism, in which every cell (system node) assumes a different and specialized function. Each system node has a genome that contains the full service specification: (1) the expected service behavior as a whole; (2) single tasks to be performed by cells; and (3) a set of rules for deciding, based on the context, which task is to be performed next. The genome is spread throughout the network through self-replication and is interpreted by each cell. Additionally, cells exchange information about the state of their neighbors for failure detection — upon detection of a failure in a neighboring cell, they replicate the genome to a new cell and re-enter the embryo state.

3.2 VIDEO-STREAMING DEPENDABILITY

Dependability techniques have been applied to video-streaming services at several levels. This section reviews the literature addressing failures in every stage of the delivery of video content to end-users. Figure 14 summarizes the most relevant related work in the area of video-streaming dependability.

3.2.1 *Encoding Failures*

Techniques that measure the QoE of end-users, such as the Peak Signal Noise Ratio (already referred in Section 2.4) have been exploited for detection of encoding errors. However, encoding errors are out of the scope of our work, since they are related with the implementation of video codecs. Our work addresses failures occurring in the infrastructure that delivers video-streaming content to end-users.

3.2.2 *Storage Failures*

Failures in the storage subsystem can be classified into fail-stop failures and performance failures.

3.2.2.1 *Fail-stop Disk Failures*

With respect to fail-stop failures, disk arrays have been used during decades as a way of improving parallelism between multiple disks to increase both the aggregate I/O performance and reliability through data redundancy. Redundant Arrays of Inex-

Areas	Project/Approach
Storage	Arrays of Inexpensive Disks (RAID) [Chen et al. 1994]
	Admission control algorithm to guarantee performance levels [Vin et al. 1994]
	Parallel streaming using data striping among servers with proxies [Lee 1998]
	Parallel streaming using data stripping and erasure correction [Lee et al. 2000]
	Server-less architecture using inter-node striping and erasure correction [Lee et al. 2002]
	Distributed video transcoding system in Cloud Computing using Hadoop and MapReduce [Kim et al. 2013]
Network	Graceful QoS degradation using Multiple Description Coders [Reibman et al. 1999]
	Comparison of multiple-description coding and layered coding [Singh et al. 2000]
	FEC performance in multimedia streaming [Frossard 2001]
	Recovering from different packet loss using the same redundant data using FEC [Horn et al. 2001]
	Rate allocation scheme using FEC in order to minimize packet losses in bursty loss environments [Nguyen et al. 2002]
	Performance analysis of implementations of multiple description coding and layered coding [Chakareski et al. 2005]
	Autonomous fault recovery through distribution of layered coding data between server nodes using the Faded Information Field architecture [Nakatogawa et al. 2006]
	FEC strategies over wireless environments [Nafaa et al. 2008]
	Comparison of Multiple Description Coding with FEC [Zhao et al. 2012]
	Correlation of user-induced interruptions of TCP connections with performance metrics [Collange et al. 2012]
Server	Calculation of the resources costs of media requests [Cherkasova et al. 2003]
	Prediction of overloading failures in streaming servers [Covell et al. 2004]
	Energy-based anomaly detection in smart phones [Ickin et al. 2013]

Figure 14: Related work on video-streaming dependability.

pensive Disks (RAID) [Chen et al. 1994] technologies have been widely deployed to prevent data losses resulting from disk failures.

The distribution of video content fragments among an array of servers in the system is also popular to increase parallelism and reliability. Video data are segmented into stripe units that are spread over several servers. In this way, the video content requested by a single client request can be delivered by several servers to achieve linear scalability. Data retrieved by several servers can be merged by clients or by proxies [Lee 1998].

Data striping between servers requires extra redundancy for reliability. Erasure correction has been exploited with data stripping in [Lee and Wong 2000][Lee and Leung 2002] to tolerate failures in individual servers through redundancy. Redundant data blocks are distributed among the servers, so that in case a server fails, the client can recover the unavailable video blocks using the redundant blocks and the main data blocks through XOR operations. This approach is similar to that exploited by RAID systems to recover data losses caused by disk failures with small redundancy overheads.

A distributed video transcoding system for cloud computing that uses Hadoop for data replication between server nodes is proposed in [Kim et al. 2013]. On top of

data replication, the system adopts the fault tolerance and load balancing mechanisms implemented by Hadoop.

3.2.2.2 *Disk Performance Failures*

Video-streaming services are data intensive. Thus, disk I/O is a typical performance bottleneck. Performance failures caused by disk bottlenecks should be avoided through load control mechanisms. That means that user requests should be accepted only when the disk can handle the associated load.

In [Vin et al. 1994], it is proposed an admission control algorithm to guarantee statistical performance levels of streaming services, while maximizing the server utilization. The algorithm relies on the analysis of variation in the access times of disk media blocks to infer the maximum server capacity. The ability of the approach to restrain the server load to its maximum capacity is evaluated successfully through simulation.

Performing load control through the analysis of the disk I/O activity is only effective when the disk is the server bottleneck. In other server configurations and workload types, the bottleneck can be other system resource, such as the CPU or the network.

3.2.3 *Network Failures*

The most common strategies for overcoming network failures in video-streaming services are graceful degradation of video quality and redundancy using erasure correction. These strategies can be employed without mechanisms for detection of network anomalies.

3.2.3.1 *Detection of Anomalies*

Network anomalies are commonly detected through analysis of network parameters, such as, packet losses and packet delays. The analysis of these parameters allows: (1) Adaptive Bitrate video players to switch between videos encoded with different bitrates; and (2) RTSP servers to control the transmission of data to video players.

There are other less common approaches for server-side detection of network anomalies. The work presented in [Collange et al. 2012] shows that the customers' experience can also be used for network monitoring. That work is based on the premise that there are strong correlations between the interruption rates of TCP connections and the network quality-of-service.

3.2.3.2 *Graceful Degradation*

Some video formats embrace graceful degradation of video quality when some packets are not received by the player on time for playback. Hence, during graceful degrada-

tion periods, end-users are able to continue watching videos but with degraded quality (e.g., experiencing glitches or reduced resolution).

Layered coding [Chakareski et al. 2005][Nakatogawa et al. 2006] is a popular graceful degradation technique used for encoding videos with several quality layers. The maximum quality is achieved when all layers are available for playback. When only part of the set of layers is available, it is possible to ensure video-playback continuity without waiting for the reception of all layers. Hence, lost and delayed packets (associated to some layers) during transmission will not interrupt the video playback. However, layered coding requires the *base layer* to provide the basic level of video playback quality. That means that only the *enhancement layers* that improve de video quality can be lost during transmission.

Multiple description coding [Reibman et al. 1999][Singh et al. 2000] is an alternative to layered coding, which decomposes the quality of videos using descriptions (equivalent to layers in layered coding). Each description alone can guarantee a basic level of reconstruction quality of the source, and every additional description can further improve that quality. This characteristic provides additional resilience against packet losses compared with layered coding, since it affords transmission losses in any of the descriptions.

Graceful degradation techniques at the encoding-level help overcoming non-sequential intermittent data losses in the network. However, graceful degradation can only be implemented with network transport protocols without data delivery guarantees (e.g., UDP). The reason is that network transport protocols providing data delivery guarantees (e.g., TCP) force data to be received in order and to be delivered to the application sequently and without loss. Consequently, the player application only receives one frame f_i when all preceding frames f_j (being $j < i$) were correctly received by it, forcing the video playback to stop when one or more frames of the video frame sequence arrive after their playback time.

3.2.3.3 *Erasure Correction*

Erasure correction techniques exploit data transmission redundancy to allow the receiver to correct transmission errors without retransmission. Forward Error Correction (FEC) has been used during decades to encode messages in a redundant way through Error-Correcting Codes (ECCs).

FEC methods has been applied to multimedia streaming services to overcome failures caused by packet losses in video transport [Frossard 2001][Horn et al. 2001][Nguyen and Zakhor 2002][Nafaa et al. 2008]. Contrasting with graceful degradation techniques, which tolerate data losses by degrading the video quality, FEC methods can recover the original data without quality degradation. However, FEC approaches are less effective in bursty loss environments than graceful degradation techniques [Zhao et al. 2012].

3.2.4 *Server-side Failures*

Server fail-stop failures are usually addressed by generic fault tolerance techniques. On the other hand, performance failures have complex failure modes that are specific to the service type. These failures are often caused by: software faults that manifest as performance anomalies (described in Section 2.3) or ineffective load control mechanisms permitting server overloading.

Performance anomalies in video-streaming services is a research topic not so exhaustively explored. The work presented in [Ickin et al. 2013] addresses detection of user interface and network anomalies in video-services provided to smartphones. It shows that the instantaneous power consumption of the smartphone is an accurate predictor of anomalies.

A modeling approach to predict failures caused by saturation of the capacity of video-streaming servers is presented in [Covell et al. 2004]. It uses models that map low-level metrics measurements to utilization of resources, built using a small number of calibration workloads. These models are used to predict the performance impact of new loads, so that the load admitted is controlled to avoid server overloading.

Several benchmarks are proposed in [Cherkasova and Staley 2003] to train performance models represented as cost functions that determine the resources involved in the processing of media workloads. These models are built specifically for requests served either from the disk or from the memory[7]. By determining the cost of each request in terms of resources, the impact of the acceptance of new requests on the server performance can be predicted. Thus, the acceptance of new requests can be controlled to avoid overloading of media servers.

3.3 RECOVERY USING VIRTUALIZATION TECHNIQUES

Failover is a popular fault tolerance strategy that switches the service provided by an active computer server to a redundant or standby computer server, upon the failure of the active computer server. Its implementation relies commonly on *checkpointing* techniques [Elnozahy et al. 2002] for rescuing the server application state and client-server connection states to another computer server. However, since server application states are stored in application-specific structures and connection states are operating-system structures inaccessible by applications, the failover process would require several levels of instrumentation.

Virtualization techniques simplify the implementation of failover. It provides server migration with generic checkpointing and migration of video-streaming services between hosts, while maintaining the same network IP address. Generic checkpointing of virtual machines avoids the assumption of applications designed (or instrumented) to checkpoint and restore their state during recovery. Additionally, virtualization allows

7 Requests benefiting from temporal locality.

generic recovery of client-server connections. This feature, implicit in virtual machines, constitutes an alternative to native migration of TCP connections, which require operating system instrumentation [Sultan et al. 2002].

Server migration requires access to the last state of the server application and each client-server connection in the active host, for guaranteeing resumption of the service in the fallback host[8]. However, failures are unexpected events that can occur anytime and consequently, it is nontrivial to provide guarantees of obtaining the last server state during failover. One solution for guaranteeing service continuity after failover is to implement *synchronous checkpointing* techniques [Marcus and Stern 2000], for ensuring permanent consistency between the active and standby replicas.

3.3.1 *Synchronous Checkpointing*

Lock-step checkpointing [Bressoud and Schneider 1996] is an active-standby replication technique commonly used to maintain consistency between two virtual machine replicas. This synchronous replication technique works by updating one virtual machine replica in the standby host on each update of the virtual machine replica running in the active host, guaranteeing that both replicas are consistent anytime. Thus, the last state of the virtual machine running in the active host can be recovered anytime using the replica of the standby host.

Synchronous replication techniques similar to the lock-step checkpointing technique ensure service continuity after failover, when the checkpointed virtual machine state is correct (fault-free). However, notwithstanding several optimization attempts have been done to these techniques [Lu and cker Chiueh 2009], they still have scalability problems and introduce complexity and overheads [Elnozahy et al. 2002], compromising their use in services with data-intensive workloads [Cully et al. 2008][Tamura 2008]. Most overheads are caused by network latencies compromising the replication performance of symmetric virtual machine replicas.

3.3.2 *Single Checkpoint*

Proactive recovery — implemented with the support of failure prediction — allows the adoption of efficient techniques for migration of server checkpoints. These techniques are based on the assumption that before the failure of the active server, it is possible to perform a copy of the server checkpoint from the active host to the fallback host.

Virtualization technologies have evolved to provide mechanisms with capabilities for migration of virtual machines between hosts with small service downtimes. These mechanisms are valuable because virtual machine checkpoints are large and thus require long periods of time to be transferred between hosts, even in a local area network.

8 Destination host of the migrated server instance.

These checkpoints enclose the operating system, applications and their in-memory states.

Techniques for migrating virtual containers using a single checkpoint can be classified into *stop-and-copy migration* and *live migration*. Typical stop-and-copy migration processes [Sapuntzakis et al. 2002][Akoush et al. 2010] involve:

1. Stopping the virtual machine and consequently, the service provided by the server running inside. This action involves checkpointing all in-memory states required to resume the service afterwards;

2. Copying the virtual machine image with the checkpoint to the destination host, throughout the network. Alternatively, when replicas of the virtual machine image coexist in the origin and destination hosts, only the checkpoints are copied between hosts;

3. Starting the virtual machine in the destination host.

The process of copying the virtual machine image and/or checkpoint to the destination host is commonly the main contributor of service downtimes during server migration. To reduce the cost of copying data between hosts, several virtualization technologies implement techniques for *live migration* of virtual machines. Live migration introduces very low downtimes and has two implementations: *pre-copy migration* [Theimer et al. 1985][Clark et al. 2005] and *demand-migration* [Zayas 1987] techniques.

3.3.2.1 *Pre-copy Migration*

In pre-copy migration, the virtual machine's memory pages are copied between hosts during the warm-up phase of the migration process (Figure 15). During this phase, the memory pages are copied without stopping the service in the origin host, which stills in production until the end of the migration process. If some memory pages change (i.e., become *dirty*) during this process, they will be copied again until the rate of re-copied pages is higher than the page dirtying rate. In a second stage, after stopping the virtual machine in the origin host, the remaining dirty pages will be copied to the destination host. Finally, the service is reestablished in the destination host.

The pre-copy migration process reduces the service downtime and consequently, the likelihood of QoE degradation resulting from server migration between hosts is reduced as well.

3.3.2.2 *Demand-Migration*

In demand-migration, the memory pages are copied after stopping the service in the origin host and starting the service in the destination host. The server starts in the destination host with a minimal subset of the execution state of the virtual machine. Then,

	Migration		
	Stage 1	Stage 2	Stage3
VM1 Running	VM1 Running / Warm-up phase	VM1 Stop and Copy	VM1 Stopped
VM2 Stopped	VM2 Receiving Memory Pages	VM2 Receive Pages	VM2 Resume Service

Figure 15: Steps of pre-copy migration of virtual machines.

whenever the virtual machine references pages that have not yet been transferred, it generates page faults. These page faults are trapped and redirected towards the origin host over the network, to be transferred to the destination host.

Demand-migration reduces the virtual machine migration time but introduces significant overheads after migration and for that reason, is less popular than pre-copy migration.

3.4 RESEARCH GAPS

Self-healing systems should distinguish between their normal and faulty states and/or behaviors, diagnose failure conditions and execute the adequate repair actions to restore the system to a correct or error-free state. To implement self-healing video-streaming systems, it is required to fulfill several research gaps in the literature.

Most of the research gaps identified are related to prediction, detection, diagnosis and repair of performance anomalies in the video-streaming domain, and to the integration of the video-streaming systems with the self-healing activities.

3.4.1 *Research on Performance Anomalies*

Performance anomalies are a research topic that has been studied for services with characteristics different than those of video-streaming services. Software aging and rejuvenation are examples of extensively studied subjects associated to this topic. Performance anomalies have also been studied in the context of failure diagnosis to identify system elements with abnormal behavior, which are those more likely to be responsible for the failures.

In video-streaming services, the failure prediction and diagnosis activities should work with small time resolutions of log data and provide responses to anomalous states in short periods of time. These requirements are challenged by the: (1) variations of individual system parameters/metrics, which demand more complex models and domain-specific metrics to avoid compromising the accuracy of failure prediction

and diagnosis; and (2) management and efficient exploitation of system models to anticipate user failures and thus, employ proactive remedial actions.

Repair of performance anomalies is another area of concern, which commonly relies on reboot techniques to restore the system to normal operation. However, reboot operations can be disruptive to the established client-server sessions. This can be critical in some video-streaming technologies that require the continuity of the sessions until the end of playback. It is important to investigate reboot techniques that support the rescue of client-server sessions to another hosts. Still, these techniques should control the impact of the reboot downtime and the further initial server warm-up period on the QoE of services.

3.4.2 *Research in Self-healing Video-streaming Systems*

Performance issues have a significant impact on the QoE of video-streaming services. Therefore, the anticipation of performance failures, accompanied with the subsequent proactive diagnosis and repair activities, are desirable in this type of services. The design of an automatic proactive recovery lifecycle integrating all these activities would help maintaining high levels of service.

The use of Autonomic Computing in video-streaming systems deserves a long research agenda. The performance sensitivity of video-streaming systems makes them excellent candidates for the implementation of the Autonomic Computing principles, such as the self-awareness and self-control capabilities. Thus, Autonomic Elements can be build from video-streaming servers by adding them self-awareness through the local management of server models, and efficient self-control capabilities through the continuous analysis of the server behavior while ensuring reduced response latencies to performance issues.

The self-healing activities implemented by each Autonomic Element should be designed to work together, by respecting the specification dependencies between self-healing activities. As an example, failure diagnosis should be addressed according to the automatic repair actions available. That means that the diagnosis classification outcomes, such as, the type and localization of the failure will determine the appropriate repair action to be executed. As well, failure prediction should provide the look-ahead time required to execute each specific repair action.

3.5 CHAPTER SUMMARY

This chapter reviewed the state of the art on self-healing systems and dependability of video-streaming services. The literature about self-healing involves multidisciplinary research work in the areas of software engineering, statistical learning, data mining, self-adaptive systems, multi-agent systems and evolutionary systems. On the other

hand, dependability of video-streaming services has been mainly studied at the storage and network levels.

The related work presented in the literature allows the understanding of the published work and the research gaps that are fulfilled with our work. The major research gaps result from the overlapping of the self-healing concepts with the dependability issues raised by performance failures in video-streaming services.

The next chapter explains in detail the research problems addressed in this thesis and presents our self-healing infrastructures for video-streaming systems.

4

SELF-HEALING APPROACH AND INFRASTRUCTURES FOR VIDEO-STREAMING

Video-streaming users are sensitive to QoS fluctuations. QoS degradation is usually manifested by: (1) rebuffering events that interrupt the video playback; (2) the increase of (re)buffering times; and (3) glitching in the display of video. These events diminish the users' QoE proportionally to their severity and to the upfront time invested by each user watching the video.

In video-streaming services, the request-response data can be sent progressively by servers to players during several seconds, minutes or even hours. To ensure video playback continuity and avoid QoS degradation, each video fragment (group of several frames) transmitted in the course of each request-response should be delivered by players before its playback time. Performance anomalies in the network and server infrastructures are a common cause of QoS failures due to data arriving too late for playback.

Proactive recovery is a promising strategy for overcoming performance failures before their occurrence, by grounding execution of repair actions with failure prediction and failure diagnosis activities. Failure prediction addresses the identification of specific patterns in the log data that indicate performance failures in the future. On the other hand, failure diagnosis classifies these patterns in terms of fault type and/or location. Both failure prediction and failure diagnosis activities are challenged by the difficulty in recognizing and classifying automatically recurring patterns of system behavior. These patterns can be complex and depend often on the underlying infrastructure and respective configurations.

The repair activity is challenged by the recovery of client-server sessions/connections and the impact of repair actions on the QoE of the service. Thus, the migration of client-server sessions between hosts, the reduction of the server downtime during reboots and the control of the server load during the warm-up stage are research issues to be addressed.

All self-healing activities are grouped in an element of the infrastructure (Autonomic Manager) that controls the element represented by the video server application (Managed Element). These elements are performance-isolated to avoid interference between their activities. Performance isolation can be achieved in the same machine through virtualization of by installing the Autonomic Manager and the Managed Element in independent machines.

This chapter contextualizes and describes our self-healing approach and the self-healing infrastructures for HTTP Streaming and Pure Streaming (RTSP Streaming)

services. It starts by presenting the self-healing problem space, assumptions and goals. Then, it presents the self-healing activities and components of each infrastructure. Afterwards, it specifies the methodology and benchmarks (with the workloads and faults loads) employed for the evaluation of the self-healing activities presented in the next chapters of this thesis. Finally, it presents the performance and QoS metrics covered by the monitoring activity and the monitoring overheads observed experimentally.

4.1 INFRASTRUCTURE REQUIREMENTS

The self-healing infrastructures proposed in this thesis adopt proactive recovery as a strategy for ensuring the continuity of video-streaming services without QoE degradation. Therefore, they should provide facilities for: (1) providing log data with small time resolutions to the self-healing activities; (2) maintaining a knowledge base with the models representative of healthy and unhealthy system behaviors created dynamically using log data; and (3) implementing efficient repair.

The research goals established for the HTTP Streaming self-healing infrastructure are the following:

1. Conceiving the structure of an Autonomic Element build from each video server instance, while ensuring performance isolation between the main server functionality and the self-healing functionality;

2. Devising an efficient data gathering mechanism for providing system, application and network performance metrics/parameters values, periodically, to self-healing activities;

3. Ensuring that the structure of the Autonomic Element is designed to facilitate the implementation of efficient reboot and checkpoint mechanisms.

We establish equivalent goals for the Pure Streaming self-healing infrastructure. However, in the HTTP Streaming infrastructure, the performance isolation between the self-healing functionality and the server is ensured through virtualization. By contrast, in the Pure Streaming infrastructure, the self-healing functionality runs in a physical machine independent of the server's machine. Consequently, the repair actions are not implemented in this infrastructure, since they depend on the virtualization infrastructure.

The decision for different infrastructures for HTTP Streaming and Pure Streaming services provides diversity that can be exploited for the evaluation of our approach under different underlying infrastructure configurations (e.g., virtualized or non-virtualized) and machine learning strategies (e.g., offline learning or online learning).

Fault Model	**Fault Duration**	Intermittent/transient faults			
	Fault Manifestation	Anomalous system behavior	QoS degradation	Fail-stutter	Fail-stop
	Fault Source	Client workloads	Server application faults	System faults (operating system/other processes)	Network faults*
	Granularity	Server application	Virtual container (Heisenbugs)	System	
	Fault Profile Expectations	Unexpected faults / Performance Failures			
System Response	**Fault Detection**	Failure detection by external agent	Load analysis	Failure prediction through anomaly detection	
	Degradation	Full performance		QoS degradation	
	Fault Response	Redirect requests			
	Fault Recovery	Automatic diagnosis of failures predicted or detected	Rebooting server application	Rebooting virtual container	Migration of virtual containers
	Time Constants	Data gathering and data analysis delays with minimum impact on look-ahead times provided by failure prediction		Recovery delays lower than look-ahead times	
	Assurance	External QoS Monitoring		Statistical guarantees of failure prediction performance updated continuously	
System completeness	**Architectural completeness** **Designer knowledge** **System self-knowledge**	Only the video-streaming technology is known. The system behavior is learned dynamically using machine learning algorithms			
	System evolution	Retraining machine learning models		Online learning	
Design context	**Abstraction level**	Virtual container and computer system			
	Component homogeneity	Not applicable			
	Behavioral predetermination	Behavior is discovered using machine learning			
	User involvement in healing	Fully automatic			
	System linearity	Not applicable			
	System scope	Multiple computers in a closed network system		Users are in an Internet-based system	

* Addressed by failure diagnosis but not covered by failure prediction

Table 2: Self-healing problem space, describing the fault model, system response and assumptions about the knowledge and the design of the system.

4.2 SELF-HEALING PROBLEM SPACE

Self-healing infrastructures should provide adequate system responses to failures. The effectiveness of these responses are evaluated according to the specification of the fault model and the assumptions about system completeness and design context, as described in Section 2.2. Table 2 presents the problem specification that will drive the design of the self-healing infrastructures presented in this chapter.

4.2.1 *Fault-model and System Response*

The fault profile assumed by our self-healing approach comprehends both expected and unexpected faults, as long as they expose system abnormal behaviors that are in the origin of performance failures. Accordingly, our failure assumptions follow not only the traditional fail-stop model associated to hard failures, but also the fail-stutter model representing system performance irregularities responsible for QoS degradation.

Recovery techniques have two strategies. They can be applied before failure occurrence (proactive recovery), using failure prediction techniques, or after failure occurrence (reactive recovery). Figure 16 illustrates the normal and faulty system states with the corresponding detection and repair strategies. Intuitively, proactive recovery — triggered by detection of performance anomalies — is the most interesting approach,

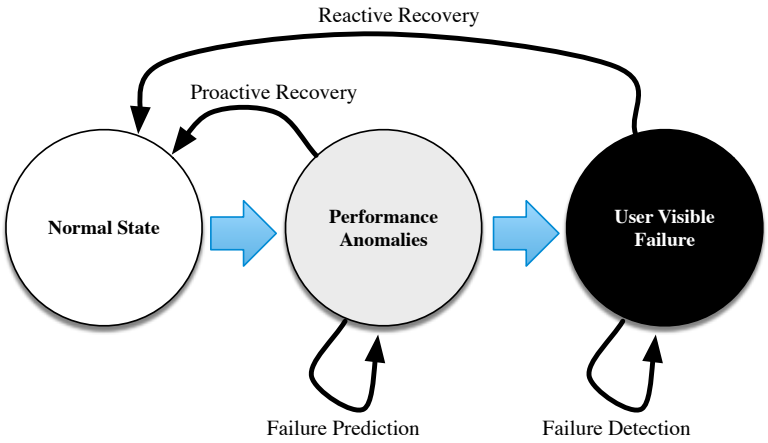


Figure 16: Normal and faulty system states with corresponding detection and repair strategies.

since it provides an opportunity to fix the problem before end-users start experiencing failures.

Selection of recovery techniques is dependent on the failure types. Performance failures are one of the main types of failures occurring server-side [Oppenheimer et al. 2003][Pertet and Narasimhan December 2005]. Figure 17 presents the taxonomy of performance failures in video-streaming services and the respective repair techniques explained later in this section. We coarsely classify performance failures according to their source into:

- **Client-workloads overloading**, when client-workloads leads the server to surpass its nominal capacity. This type of performance failures are caused by workloads generated by organic client-server requests that force the server infrastructure to exceed its maximum load capacity;
- **Performance anomalies**, when the server misbehavior is explained by software faults — e.g. memory leaks, configuration errors or parasite processes sharing resources with the video server.

Client-workload overloading and performance anomalies have different causes and require different recovery strategies.

4.2.1.1 *Client-workload Overloading*

Client-workload overloading failures are caused by load balancing errors or ineffective load control. In a typical cluster configuration, the requests are assigned to server instances by load balancers according to specified policies. Load balancing errors leading to uneven distribution of loads between machines occur even in large scale global services, such as Amazon EC2 [ama 2012] and Google Applications[goo 2012]. Also,

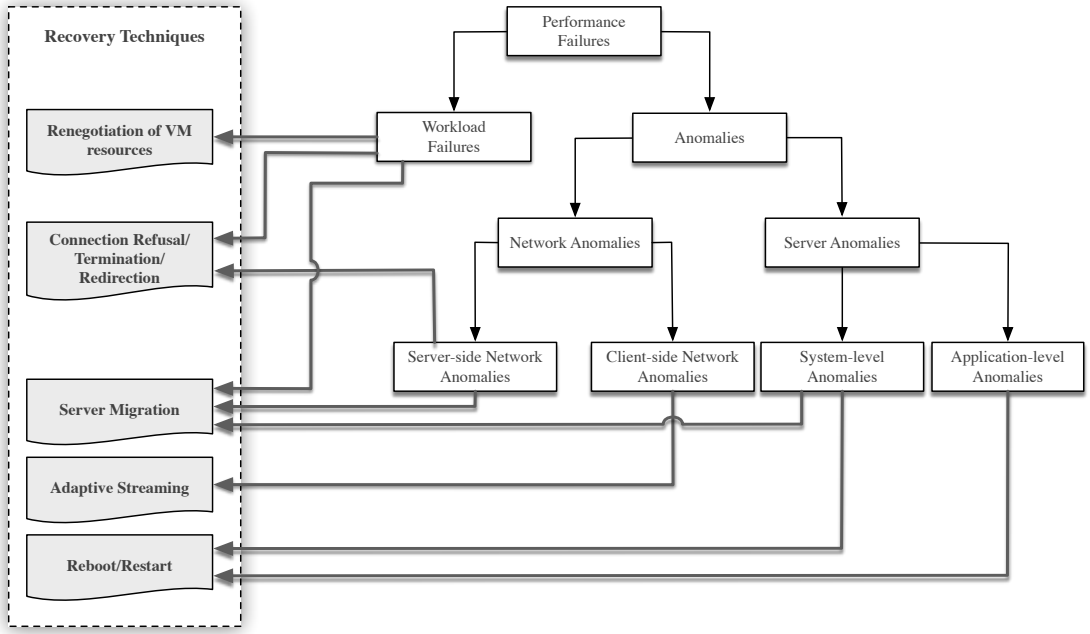


Figure 17: Taxonomy of performance failures and corresponding repair techniques.

in video-streaming services, the load admission activity is complex and depends often on the workload type. Factors like the video encoding bitrate, the video format and the popularity of the videos provided by the server, interfere with the maximum number of requests allowed simultaneously by the server. Thus, workload type changes lead often to service failures caused by server overloading [Cherkasova et al. 2008][Cherkasova and Staley 2003].

Workload-related failures are caused by external factors to the server. However, the context of the Autonomic Element is delimited by the server instance, which should guarantee self-protection against this type of failures by detecting and diagnosing them. In which concerns to recovery, all the repair actions pursue the reduction of the server load, as such:

- Selective termination of requests being handled by the server;
- Renegotiation of server resources, when the server is running over a virtualization infrastructure;
- Migration of the server application to a better provisioned host.

The first action disrupts part of the connections established between clients and the server and thus, should be avoided whenever it is possible. The last two actions relies on infrastructure capacity management features for negotiating more resources

or moving the server application from a underprovisioned host to a better provisioned host with sufficient resources to sustain the workload.

4.2.1.2 *Performance anomalies*

Performance anomalies are triggered by transient faults, also known as Heisenbugs [Gray 1996], a class of faults that are difficult to reproduce and, consequently, to diagnose. Performance anomalies can be breakdown into *network anomalies*, *system anomalies* and *application anomalies*.

In video-streaming services, network anomalies have standard solutions based on graceful degradation of video quality. Examples of these solutions are the Adaptive Bitrate streaming techniques to adapt the service to limited network conditions [Stockhammer 2011] (e.g., the recent MPEG-DASH standard [Sodagar 2011]), temporal data redundancy [Feamster and Balakrishnan 2002] and spatial data redundancy [Puri and Ramchandran 1999].

System anomalies and application anomalies bring the server to undeterministic states at irregular and unpredictable times. During these periods, the observed application and/or system behavior is unexplained by the observed workload. That means that the type and volume of client requests processed by the application suggests a different behavior (e.g., higher than expected request-response times). These types of performance anomalies can be commonly handled by rebooting the faulty components [Li et al. 2002][Grottke et al. 2006][Candea and Fox 2003] and/or migrating healthy components to another failure-free infrastructure for reestablishing the service there [Dobre et al. 2011].

We anticipate user-visible failures through detection of performance anomalies, using classification models created by supervised learning algorithms [Hastie et al. 2001] for capturing scenarios representative of normal and anomalous system behaviors.

Since it is often unlikely to attain 100% of accuracy using classification models, the unpredicted failures are detected later by a failure detector, when they manifest as QoS degradation or fail-stop failures. Reactive recovery is applied to these failure scenarios.

4.2.2 *Assumptions About System Completeness*

The design of a self-healing system is constrained by the assumptions of the knowledge about the system behavior. The specification of most systems is incomplete and thus, the self-healing approaches must deal with the limits of knowledge.

The architecture of our self-healing system is known but not its behavior during normal and faulty periods. An Autonomic Element encloses one server application that are usually delivered without the specification of the performance behaviors expected for the underlying infrastructure. On top of that, these behaviors can change along with new software updates and changes in the underlying infrastructure. There-

fore, system behaviors are modeled automatically in our self-healing systems using machine learning algorithms, for normal and faulty periods.

Workload-related failures have less complex server behaviors than performance anomalies, since the fault profile is known and the failures are explained by the exhaustion of specific system resources. Still, their failure patterns are also difficult to diagnose. Notwithstanding server overloading is often bound to specific resources, the resultant error patterns can involve several resources in complex ways. Since resources have dependencies between themselves, the exhaustion of one resource can lead to exhaustion of the others. As an example, memory exhaustion often leads to *trashing* in the *paging system*, originating intense CPU utilization and I/O activity at the beginning, while moving *pages* from memory to disk and back again. Later, this behavior can be followed by low CPU utilization when: (1) the operating system reduces the multi-programming level to reduce the CPU load; and (2) the CPU starts waiting for disk operations, while moving pages from memory to disk and back again. The complex interactions between resources demand diagnosis models able to identify the patterns associated to workload-related failures, using the same approach used for performance anomalies.

4.2.3 *Design-context*

The design context is delimited by the system scope. In our infrastructures, the system scope is confined to the Autonomic Element, which is composed by the server application instance (Managed Element) and the self-healing functionality (Autonomic Manager).

In both self-healing infrastructures, the server application is changed to incorporate probes that provide relevant monitoring data to the self-healing activities. On the other hand, all the core self-healing functionalities are added without design assumptions about the Managed Element.

4.3 SELF-HEALING INFRASTRUCTURE FOR PURE STREAMING

Figure 18 presents our self-healing infrastructure for Pure Streaming services. It includes the monitoring, failure prediction and failure diagnosis activities.

The *server host* represents the Managed Element in the Autonomic Computing architecture. It has installed the video-streaming server application and the apparatus necessary to generate the logs required by the self-healing activities.

The *probing agent* monitors the service using a video-streaming workload generator, which establishes synthetic RTSP requests with the server periodically and collects service quality measurements. The probing agent also runs a performance analyzer, which represents the Autonomic Manager in the Autonomic Computing architecture. The performance analyzer integrates data relative to service quality metrics and sys-

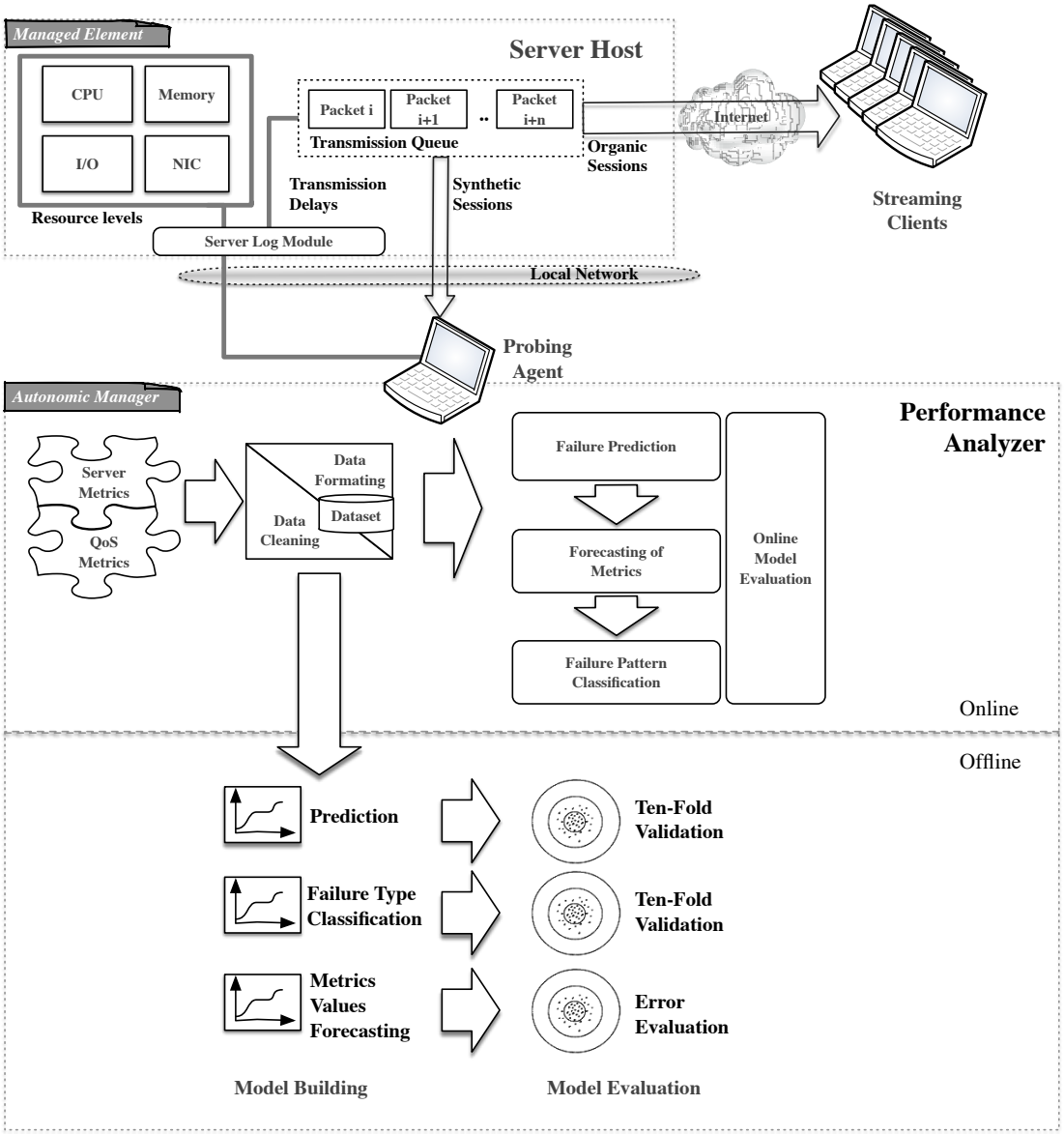


Figure 18: Self-healing infrastructure for Pure Streaming.

tem, network and application-level metrics/parameters gathered server-side. Afterwards, it uses these data for failure detection, failure prediction and failure diagnosis.

4.3.1 *Data Gathering*

Server-side performance metrics are gathered by the Server Log Module and are provided to the performance analyzer using a pull-based approach[1]. System parameter values (e.g., CPU and memory usage) are provided by the operating system. On the other hand, application-level performance metrics require server application instrumentation and the development of a video server application module.

Application-level metrics expose the server transmission delays of frames to end-users (players). Pure Streaming servers control the timing for transmission of video frames to end-users. The frames read from disk are grouped into packets that are queued and scheduled by the server application to be transmitted at a specific time. The scheduled time is estimated according to statistics provided by RTCP reports issued periodically by players (e.g., round-trip delay and inter-arrival jitter). Therefore, the server application performance can be measured through comparison of the packet scheduled times with the respective transmission times. In other words, server performance degradation is being experienced at the server application-level when packets are being transmitted after their scheduled time. In addition to the data transmission delays, it is also measured at the application-level the client-server interaction delays relative to RTSP commands (e.g., PLAY or PAUSE).

The performance analyzer integrates server-side metrics/parameters with service quality metrics obtained by probing the server with synthetic sessions. Data integration involves preprocessing operations, such as, formatting, correction of structural errors, calculation of derived metrics, temporal alignment of data coming from the several sources and finally, flattening data into a single table. Each table row represents a log instance representative of the global service performance at a specific time.

4.3.2 *Prediction and Diagnosis of Failures*

Failure prediction and failure diagnosis are online activities that relies on classification models [Hastie et al. 2001] to detect performance anomalies and determine their cause, respectively. At the prediction time, performance anomalies signal errors that should be handled before they manifest as service failures. Later, performance anomalies are classified in the failure diagnosis activity by classification models, using metrics values forecasted from the prediction time to the failure time by means of regression models [Hastie et al. 2001] (described in Section 7.2).

Initially, all models are evaluated offline using historical data through *ten-fold validation* (described in Section 6.3.2). Afterwards, they are evaluated online using new arriving data, according to the correctness of predictions performed by them.

1 The probing agent requests the log data every t seconds.

4.3.3 *Components of the Evaluation Infrastructure*

For the evaluation of the self-healing approach proposed in this thesis for Pure Streaming systems, we use the following elements in the infrastructure:

- **Darwin Streaming Server** (DSS) [dar 2012], an open-source popular streaming server;
- **DSS Log Module** build by us to capture application-level metrics server-side. This module is presented as the Server Log Module in the infrastructure;
- **System Activity Report** (SAR) tool [sar 2012], for capturing system metrics in the server host;
- An instrumented version of the **StreamingLoadTool load generator** [str a] (distributed with the DSS). We instrumented this tool for probing the service quality, by generating synthetic streaming sessions periodically and obtaining service quality metrics over these sessions;
- **Weka** [Hall et al. 2009], an implementation of machine learning algorithms for feature selection, model building and model evaluation.

Details of the configurations, metrics analyzed and activities performed by the infrastructure are presented in the respective chapters of this thesis.

4.4 SELF-HEALING INFRASTRUCTURE FOR HTTP STREAMING

The self-healing infrastructure for HTTP Streaming is named SHStream. This infrastructure is presented in Figure 19. It represents an Autonomic Element in the system, which is divided into two main parts: the virtual container running the web server (Managed Element) and the SHStream self-healing application (Autonomic Manager) that implements the self-healing activities.

The Autonomic Element represents either a virtual machine or a non-virtualized machine running the video server application. It incorporates the Autonomic Manager and the Managed Element in the same machine, by employing container-based virtualization for isolating the performance of both elements. Thereupon, the self-healing functionality is attached transparently to standard video servers without performance interference between them.

In HTTP Streaming services, the video server application is usually a web server. Accordingly, the virtual container runs the web server with the video-streaming module that implements HTTP requests semantics for video-streaming — e.g., time seek requests that enable viewers to jump to any part of the video. Plus, a web server module developed by us (mod_SHS) provides the metrics required for measuring the application-level performance.

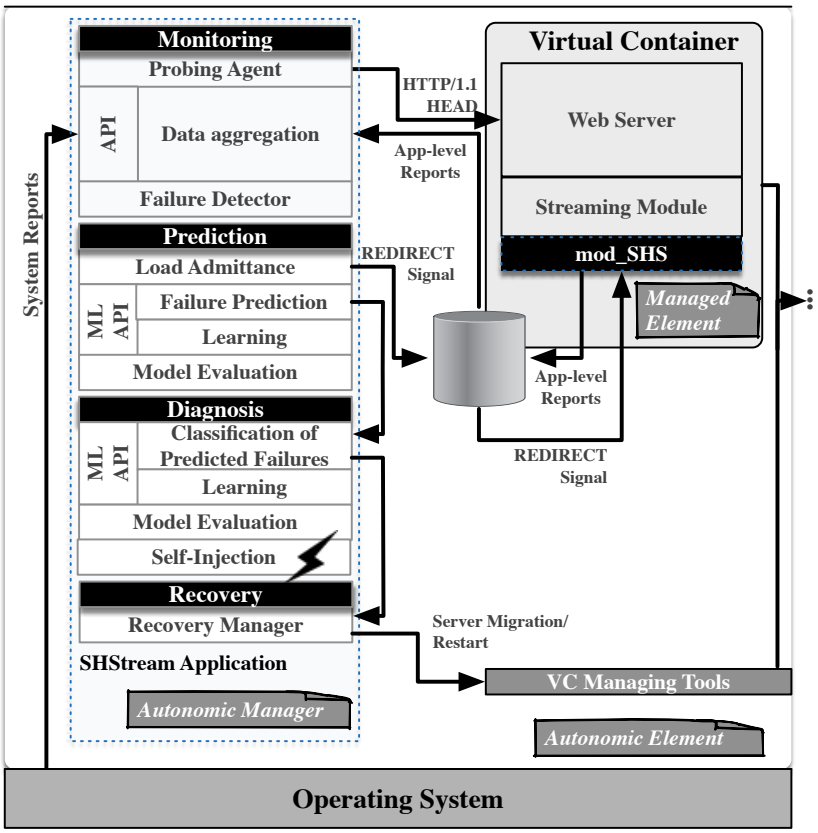


Figure 19: SHStream infrastructure.

4.4.1 *Self-healing Activities*

SHStream is divided into four functionality groups: *monitoring*, *failure prediction*, *failure diagnosis* and *recovery*. Each of these groups is associated to a specific stage of the self-healing cycle.

In the first stage, the monitoring group of functionalities manages the data gathering process and performs failure detection. It starts by aggregating, cleaning and formatting the log data every n seconds, gathered from:

- **Application-level reports**, containing web server performance metrics;
- **System-level reports**, which also include network statistics associated to network interfaces;
- **Web server probing status reports**, created using HTTP HEAD requests issued every n seconds to determine server responsiveness.

At each period, these reports are integrated to build a log instance. The resultant log instance is used by the failure detector to determine the service state: normal state or failure state.

In the second stage, the failure prediction group of functionalities uses each log instance for load admittance and for:

- **Online learning** — incremental learning of *normalcy patterns* [2] and *pre-failure patterns* [3] (described in Section 6);
- **Failure prediction** — prediction of performance failures through detection of pre-failure patterns;
- **Model evaluation** — continuous evaluation of the pool of models created using several algorithms and selection of the best prediction model from the pool.

The SHStream application is a framework for multi-model failure prediction. It works with several machine learning algorithms to provide diversity of models. At each failure prediction iteration, the classification output given by the model with the best prediction performance in the framework is taken to decide whether proactive recovery is needed. In the same iteration, the performance statistics of each model are updated and are used to select the model that will be used for predicting failures in the next iteration. These statistics expose the accuracy of predictions, by taking into account the observation or not of failures predicted in the past.

In the third stage, each failure predicted by the classifier is handled by the failure diagnosis group of functionalities. Thus, the log instance of the failure predicted is classified by several diagnosis classifiers and the classification output given by the most accurate model classifier will decide which type of recovery action is needed. Diagnosis models are trained and evaluated similarly to the failure prediction models.

Finally, the recovery manager coordinates the execution of repair actions, selected according to the diagnosed failures. Repair techniques include the reboot at different granularity levels — full machine reboot and virtual container reboot — and migration of virtual containers between hosts.

4.4.2 *Virtualization*

The SHStream application runs in the server's machine in the *virtual environment zero* — also called *host domain* — to promote:

- **Integration** and **performance isolation** — virtualization isolates the SHStream application from the web server application (both running in the same machine);

2 Log data associated to error-free periods.

3 Log data associated to performance anomalies preceding user-visible failures.

- **Dependability**, **self-responsibility** and **scalability** — avoidance of single points of failure, since the self-healing functionality fails along with the server application. Scalability problems are also avoided, since each server application is controlled by its own self-healing apparatus;
- **Timeliness** — all communication delays are minimized by performing data gathering, data analysis and direct server repair actuation in the server's machine.

Virtual containers can run inside a typical standard virtual machine to structure the Autonomic Element. Due to the small size of virtual containers, they can be rebooted or migrated between hosts, with significantly smaller overheads than virtual machines.

4.4.3 *Components of the Evaluation Infrastructure*

The infrastructure used for evaluation of the self-healing approach proposed in this thesis for HTTP Streaming servers has the following elements:

- **OpenVZ** [ope] for container-based virtualization. OpenVZ is open-source, well documented and provides a rich set of features for migration of virtual containers;
- **Lighttpd web server 1.4.30**, a popular and efficient web server that has been used by video-streaming services such as Youtube, for several years, to deliver videos [Do Cuong 2007];
- **mod_SHS module**, a Lighttpd module implemented by us to gather application-level metrics data. The mod_SHS module also performs load control, by redirecting new connections to other server instances when the server reaches its capacity. The disk is used for communication between the mod_SHS module and the SHStream application, using a directory shared between the virtual container and the host domain;
- **H264 Streaming module** (mod_h264_streaming version 2.2.7) [h26 2012] for the Lighttpd web server. This module performs bandwidth control and handles time range requests and requests targeting specific video segments in Adaptive Bitrate streaming.

The SHStream application represents the Autonomic Manager in the Autonomic Element. It is developed in Java and has the following main dependencies:

- **SIGAR** [sig 2012], the API for gathering reports of system statistics;
- **Massive Online Analysis (MOA)** [Bifet et al. 2010], the library with the implementation of online machine learning algorithms.

4.4.4 *Load Control*

Load control mechanisms handle the problem of avoiding server overloading failures. Load control is a typical problem of any Internet service. A service should not accept more load than its underlying infrastructure could handle. Load balancing and load admittance errors combined with underprovisioned server resources during workload peaks make overloading one probable cause of failure [Pertet and Narasimhan December 2005][Oppenheimer et al. 2003].

In other types of Internet services, load admittance is usually performed statically by limiting the number of requests being accepted, by limiting selectively the acceptance of sessions according to their type [Cherkasova and Phaal 2002] or by modeling the server performance (e.g., using queueing theory [Aweya et al. 2002][Robertson et al. 2003]). In video-streaming services, the load admittance problem has been addressed by assuming stable workload types when determining the server capacity and its current utilization [Cherkasova and Staley 2003] and through dynamic prediction of server saturation by considering the current utilization of resources to decide the acceptance of new loads [Covell et al. 2004]. However, these approaches are less effective in controlling the server load when the workload type changes. Such is the case of a change in the temporal locality of reference of the workload that can change the volume of requests supported by the server and the profile of consumption of resources by each request. As well, these approaches are ineffective when other applications running in the same machine interfere with the resources available to the video server application.

SHStream implements a self-protection mechanism against overloading, which redirects video-streaming requests when they cannot be afforded by the server or the network bandwidth. This mechanism anticipates overloading conditions when other load control mechanisms external to the Autonomic Element fails. Notwithstanding load control is not in the core of our research, it represents an important activity of our self-healing infrastructure required to protect the server against this type of failures.

4.4.4.1 *Load Control Implementation*

Load control is implemented in SHStream through the analysis of the server throughput provided in excess to the requests being handled. Thus, in spite of changes on the throughput provided by the server (increase or decrease), the acceptance of new requests will adapt accordingly.

New requests are accepted by the server when the *excess margin*, calculated in (1) as the difference between the average of the ratio of actual transmission bitrates TBR to encoding bitrates EBR, for the requests associated to videos with the highest bitrate, is higher than 1 plus a safe margin α. The value of α should be carefully chosen, because

it represents the margin of server throughput provided in excess to actual requests that can be used by new requests.

$$\textit{excess margin} = \frac{\sum_{i=1}^{nCurrent} \frac{TBR(i)}{EBR(i)}}{nCurrent} \qquad \textit{excess margin} > 1 + a \tag{1}$$

In our implementation, the excess margin focuses only on the requests with the maximum bitrate in the current workload. The rationale behind this decision is that these requests are the first being impacted by reduction of throughput. Intuitively, since the throughput is not shaped per request according to the respective video encoding bitrate, the requests targeting videos with the highest encoding bitrate are those with lowest throughput in excess.

SHStream controls load admittance using a flag file to inform the mod_SHS Module that the server has reached its capacity. The existence of this file indicates that new connections should be redirected to an alternative host, by sending a HTTP REDIRECT command to the client. The client then reissues the connection request to the alternative host provided in the REDIRECT command.

The problem of selecting alternative hosts during redirection of requests are orthogonal to our work. Host selection policies (e.g., random selection and resource-aware selection of hosts) have to be developed to complement our load admission mechanism.

4.4.4.2 *Limitations*

Notwithstanding the advantages of performing dynamic load admission based on the excess of service bandwidth, one pitfall should be mentioned. There is a temporal gap between the acceptance of new requests and their reflection in the excess margin of throughput. This phenomena is explained by startup delays on transmission of request-responses, measured from the time the server accepts each request until stabilization of its transmission bitrate.

Figure 20 illustrates two scenarios exhibiting the impact of the acceptance of new requests and the termination of requests in the throughput excess margin. The termination of requests is represented by a suddenly increase of the excess margin. By contrast, the acceptance of new requests is represented by a suddenly decrease of the excess margin followed by an increase until stabilization. The explanation for this behavior is that new requests are accounted by $nCurrent$ in the formula presented in (1) before the server starts transmitting request-responses data. Consequently, the average of the ratio of TBR to EBR drops, forcing temporarily the excess margin to drop below the safe margin a. During that period, the server redirects new requests, until the data transmitted for the last requests reflect in the excess margin.

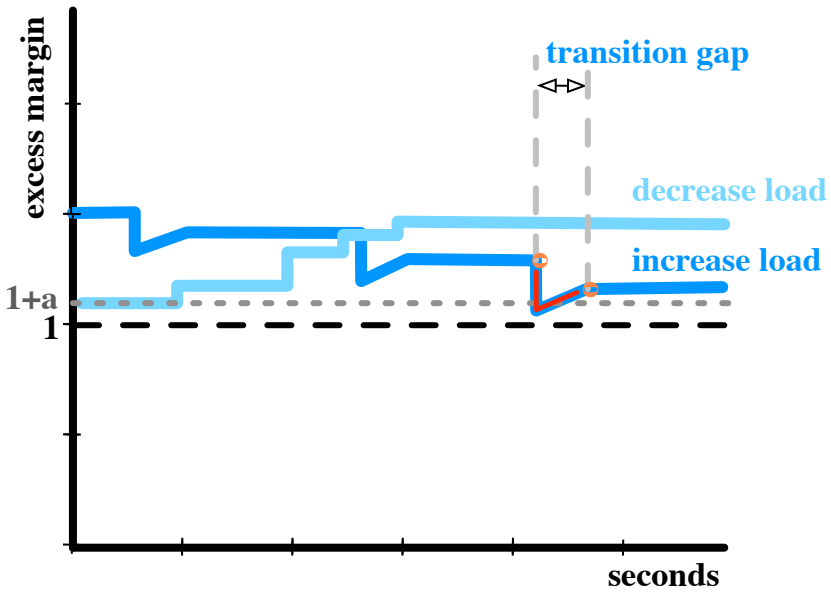


Figure 20: Two scenarios illustrating the excess margin of throughput, when the server receives and terminates several requests simultaneously.

Transition gaps can lead to rejection of connections, even when the server can afford them. However, this is an expected scenario only when the server is close to the limit of its capacity and the number of requests in the startup phase is significant.

4.5 EVALUATION METHODOLOGY

This section describes the testbed and benchmarks used to evaluate each of the self-healing activities presented in the respective chapters of this thesis.

4.5.1 *Experimental Testbed*

Our experimental testbed presented in Figure 21 has the following configuration:

- Five machines configured with an Intel(R) Pentium(R) D CPU 3.00GHz, 4Gb of RAM, running the Linux 2.6.18 – 92.1.22.el5 Kernel;
- All machines are connected by a 1Gbps Ethernet network;
- Workload generators installed on two machines, for avoiding client-side overloading. We use the httperf tool [Mosberger and Jin 1998] for generation of HTTP video-streaming requests and the StreamingLoadTool for generation of RTSP requests;

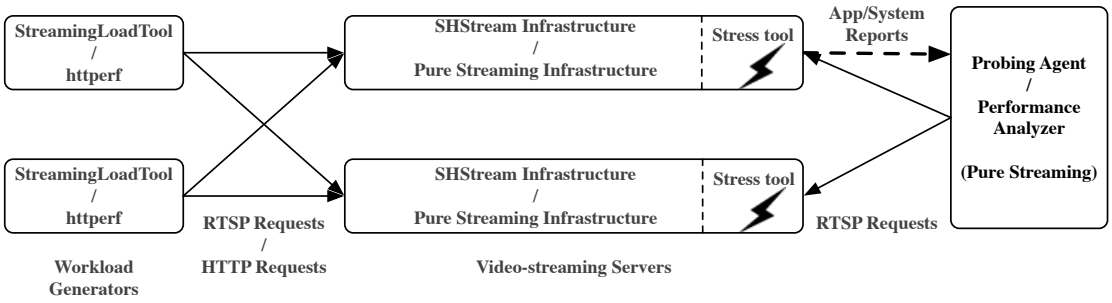


Figure 21: Experimental Testbed

- Two machines run an instance of the Pure Streaming and HTTP Streaming infrastructures;
- One machine runs the probing agent (implementing the performance analyzer) of the Pure Streaming infrastructure;
- Server-side performance failures are induced using the Stress tool [str b].

The experimental testbed runs benchmarks representative of typical system usage for evaluation of the self-healing activities. Client workloads are an important component of benchmarks, which allows evaluation of the system behavior in the presence of representative client-server interactions.

4.5.2 *Workload Characterization*

We define client workload as the load imposed by client requests to servers. Standard web server benchmarks, such as SPECweb 2009 [Corporation 2009] reflect traditional web server traffic.

Traditional web items are typically small and benefit intensively from locality [Pariag et al. 2007][Brecht et al. 2004]. Video-streaming traffic contrasts with the characteristics of traditional web traffic, since videos are commonly large objects that are streamed during long periods of time and only a small portion of the video content is effectively downloaded during the lifespan of most request-responses [Finamore et al. 2011][Gill et al. 2007]. At anytime, streaming users can stop watching the videos and consequently, avoid downloading the unplayed portion of the video content not downloaded yet. This behavior contrasts with other short web items, as those accessed by web pages, which are downloaded completely before the user evaluate his interest in the respective content.

Client workloads are basically characterized by three parameters: *popularity*, *encoding bitrate* and *number of connections* of videos [Cherkasova and Staley 2003][Covell et al.

2004]. Other parameters also important are the *user abandonment*, the *video duration*, the *inter-arrival request times* and the *download bitrate*.

4.5.2.1 *Popularity of Videos*

Popularity represents the dispersion of video-streaming requests over video files during a delimited period of time. In *popular workloads*, the requests handled simultaneously by the server are distributed over a small number of video files, as illustrated in Figure 22. By contrast, in *unpopular workloads*, the requests are distributed homogeneously over a large number of video files.

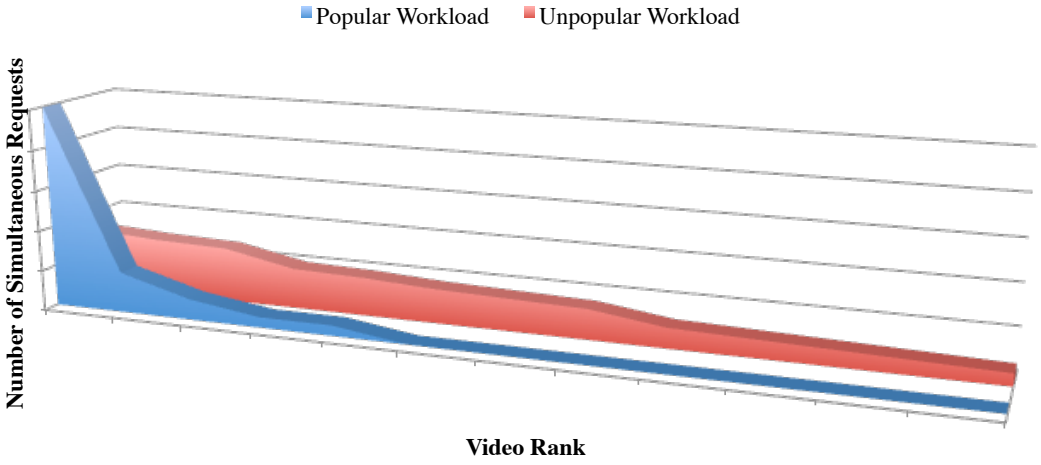


Figure 22: In popular workloads, the requests are concentrated in a small number of videos. By contrast, in unpopular workloads, the requests are spread out over a large number of videos.

We devised three workload configurations to represent the popularity of videos:

- **Cached** — the same file is streamed by all requests;
- **Disk** — each request streams exclusively one video file;
- **Mix** — the distribution of requests over videos files is performed according to a Zipf distribution.

The Cached and Disk workloads represent popularity extremes. The Cached workload explores the scenario where the video-streaming content requested are extremely popular, obtaining the maximum benefit from temporal locality. On the contrary, the Disk workload explores the scenario of a pure disk-intensive workload, obtaining the lowest temporal locality.

The Mix workload represents popularity according to a Zipf distribution. Previous studies have shown that the Zipf distribution fits the popularity of web traffic in general [Breslau et al. 1999]. Even web caching strategies are designed to account for a Zipf distribution in the number of requests for webpages [Adamic and Huberman 2002]. As well, several studies suggest that the popularity of video-streaming traffic also follows a Zipf distribution, as observed for Youtube [Gill et al. 2007] and other video-streaming services [Yu et al. 2006][Cherkasova and Gupta 2004]. However, there is no universal agreement regarding the acceptance of the Zipf distribution for modeling popularity of video-streaming workloads. Several studies addressing characterization of video traffic showed that the Zipf distribution does not fit the popularity of videos, due to a drop-off in the tail of the observed distribution [Cha et al. 2007][Mitra et al. 2011], as shown in Figure 23. Other studies have shown that the Weibull and Gamma distributions have better adjustment to popularity of video objects in Internet video services [Cheng 2007][Cheng et al. 2008][Abhari and Soraya 2010].

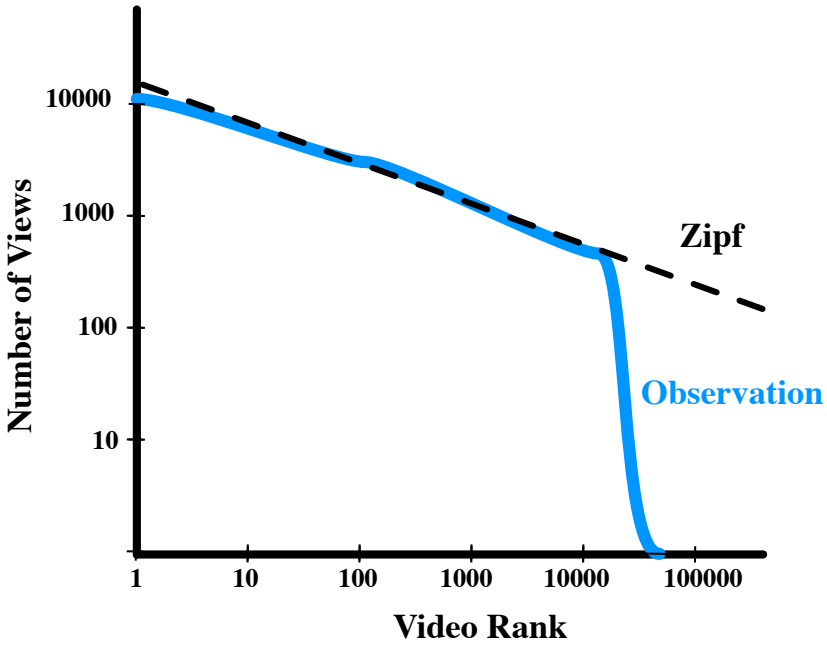


Figure 23: Typical ranked video view count in log-log scale, as observed in some video-streaming services.

The benchmarks devised in this thesis for the evaluation of the self-healing infrastructures adopt the Zipf distribution for modeling popularity. We believe that the adjustment error of Zipf relatively to real popularity distributions will not interfere with the representativeness of our results. Also, the adoption of Zipf is a fair assumption due to the observation of this distribution in popular video-streaming services (e.g., Youtube) and the lack of consensus about the distribution that could fit better the pop-

ularity of videos. Our assumption was also followed in a recent work for the same reasons [Summers et al. 2012b].

4.5.2.2 *User Abandonment*

Since videos are downloaded progressively by end-users, when they lose interest in a given video, they will stop downloading it. Thus, the workload should incorporate the portion of each video that is effectively downloaded.

Workload studies addressing user abandonment are rare. In the Youtube service, was observed that only about one quarter of the transactions were complete [Gill et al. 2007]. However, to represent user abandonment of visualization of videos, we use the statistics provided by a more recent study [Finamore et al. 2011] about the Youtube service. To simplify the reproduction of user abandonment from the Cumulative Distribution Function (CDF) presented in that study, we selectively discretize the CDF and select key points from it. Accordingly, we force randomly 40% of the videos to terminate at 10% of playback time, plus 20% of videos at 20% of playback time and finally, plus 30% at 50% of playback time.

4.5.2.3 *Video Duration*

According to [Ameigeiras et al. 2012][Zink et al. 2009], approximately 90% of the Internet videos have a duration between 10 seconds and 16 minutes. We adopt the same configuration using 100 different durations, starting at 10 seconds and ending at 1000 seconds, at intervals of 10 seconds.

4.5.2.4 *Encoding Bitrate*

In [Ameigeiras et al. 2012], it is shown that 99% of video clips of the three most popular video configurations, have a video stream encoding rate below 500 kbps, 810 kbps and 1300 kbps. Similar results can be found in [Gill et al. 2007]. In [Zink et al. 2009], most videos are encoded with a bitrate varying between 632 and 908 kbps. Another study [Finamore et al. 2011] shows that the 360p videos in Youtube (currently the default choice) surrounds 600Kbps video rate, in average, while 480p and 720p videos surrounds 1 Mbps and 2.5 Mbps in average, respectively.

In our benchmarks, all videos are encoded by H.264 at bitrates of 650 Mbps (360p), 1 Mbps (480p) and 2.5 Mbps (720p), covering the most popular video encodings.

4.5.2.5 *Download Bitrate*

In HTTP streaming, the download bitrates are higher than the encoding bitrates during periods of normalcy. Some server implementations have two different behaviors with respect to the download bitrate provided for each request: *burst* and *throttling*. The burst phase has a lifespan of a few seconds, during which, all data downloaded by

players are transferred at the maximum speed allowed by the server and network. Afterwards, in the throttling phase, the traffic is shaped by the server. In the case of Youtube, during the throttling phase, the video is sent at the speed of the video clip encoding rate multiplied by 1.25 [Ameigeiras et al. 2012].

The lifespan of the burst phase depends on the available network bandwidth and server resources. In the Youtube, the burst phase is limited by an amount of approximately 40 seconds of video data downloaded [Ameigeiras et al. 2012], but is higher for low bitrate videos.

Since the H264 Streaming module used in our HTTP Streaming infrastructure only allows specification of the burst period in seconds, we define an approximate value for that phase in seconds, considering the time required for downloading 40 seconds of video for the intermediate bitrate (1 Mbps) using our infrastructure, in average.

4.5.2.6 *Inter-arrival Request Times*

In our benchmarks, the request inter-arrival times follows a Poisson distribution [Forbes et al. 2011]. Request inter-arrival times have been modeled by a Poisson process for several years for web workloads in general [Arlitt and Williamson 1997][Adhikari et al. 2012][Gupta et al. 2009] and, specifically, for video-streaming [Kang et al. 2008][Mori et al. 2010][Adhikari et al. 2012].

The benchmarks presented in this chapter also uses a Poisson distribution for modeling the request inter-arrival times.

4.5.3 *Design of Benchmarks*

We devise several benchmarks for experimental evaluation of the self-healing infrastructures: *mix+spike*, *popular+spike*, *unpopular+spike*, *mix+anomaly*, *popular+anomaly*, *unpopular+anomaly* and *mix+anomalyNet*. All benchmarks comprehend two periods:

- **Long period of normality** — a fault-free normal period;
- **Short faulty period** — a period during which the server is exposed to either high loads (client-workload overloading) or performance anomalies.

Each evaluation benchmark is structured into a single run with a timespan of approximately 90 hours. All benchmarks use 100 H.264 videos.

4.5.3.1 *Mix+spike, Popular+spike and Unpopular+spike Benchmarks*

The Mix+spike, Popular+spike and Unpopular+spike benchmarks combine periods of normality with periods of failure generated by server overloading conditions. These benchmarks are distinguished by the workload configuration they use.

The Mix+spike benchmark uses the Mix workload (presented in Section 4.5.2), which models popularity of videos according to a Zipf distribution. The Mix workload represents typical video-streaming workloads, specified according to previous studies. The remaining benchmarks adopt workloads associated to extremes of popularity of videos. These benchmarks are used for evaluation of the system behavior when the popularity of videos changes significantly.

The Popular+spike benchmark uses the Cached workload, which exercises the server by streaming the same file in all requests. This workload type is low disk-intensive, as the video content obtains the maximum benefit from temporal locality. By the contrary, the Unpopular+spike is a disk-intensive benchmark that uses the Disk workload to exercise the server by streaming each video at most once during the experiments, until all the other videos have been requested. Hence, this workload avoids temporal locality benefits.

The number of connections in all the three benchmarks varies sinusoidally in time between $[10\% - 50\%]$ (Type I) and $[10\% - 90\%]$ (Type II) of the server nominal capacity during periods of normality, as illustrated in Figure 24. During these periods, the selection of Type I and Type II loads is performed randomly. During overloading periods (spikes), the number of connections varies between $[10\% - 120\%]$ of the server nominal capacity, to induce overloading failures. The period of normality has a duration of 10 minutes and is followed by a overloading period with a duration of 1 minute. During both periods, the videos are requested according to the popularity distribution specified by the workload.

4.5.3.2 *Mix+anomaly, Popular+anomaly, Unpopular+anomaly and Mix+anomalyNet Benchmarks*

The Mix+anomaly, Popular+anomaly, Unpopular+anomaly and Mix+anomalyNet benchmarks combine normal service activity with periodic performance anomalies. Performance anomalies show up as exhaustion of resources, as the result of activation of software faults.

The space of potential faults responsible for performance anomalies is large. Studies characterizing these faults are not available or are incomplete since root cause analysis is very complex for this type of failures. It is common the use of techniques to tolerate these faults (e.g., using software rejuvenation) instead of fixing the software. For that reason, instead of reproducing the line (or lines) of faulty code, we reproduce their effects. Performance anomalies manifest in terms of system conditions unexpected for the actual workload. Thereupon, we perturb the main system resources to induce performance anomalies that are reflected in system, application and network parameters/metrics as signatures representative of performance failures.

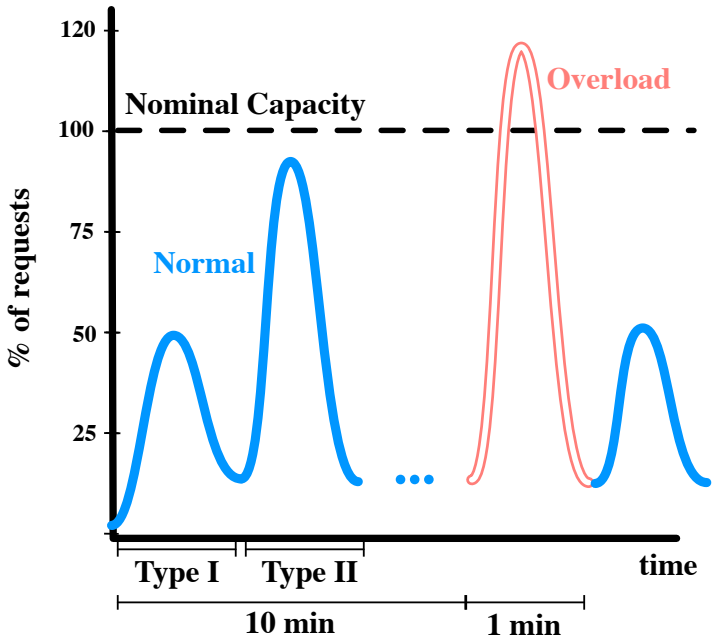


Figure 24: Structure of the Mix+spike, Popular+spike and Unpopular+spike benchmarks.

Performance anomalies might point to either *system anomalies*[4] or *server application anomalies*[5]. Accordingly, performance anomalies are injected at two levels: system and server application.

System anomalies are generated by creating performance irregularities that compromise the relationship between the server workload and the main system resources being consumed. During faulty periods, performance anomalies are generated by an independent process that selectively steals system resources to induce resource exhaustion unexplained by the server workload generated by client requests. Three types of performance anomalies are considered for injection using the Stress tool:

- **CPU Stress** - spawn fork processes, each spinning on *sqrt()* function;
- **I/O Stress** - spawn fork processes, each spinning on *sync()*, which writes any data buffered in memory out to disk;
- **Memory** - spawn fork processes, each spinning on *malloc()*.

The intensity of each fault injected is proportional to the number of times each function is invoked. The intensity of each fault is regulated randomly to induce conditions of performance degradation either with and without associated user-visible failures.

4 Environmental faults unrelated with the server application.
5 Faulty application behavior caused by not fully debugged server application code.

Thus, it is possible to evaluate the ability of the self-healing functionality to handle only performance anomalies associated to failures.

Figure 25 presents the structure of all *+anomaly benchmarks. They have the same structure of the *+spike benchmarks, but maintain the same load during periods of normality and faulty periods. In replacement of the overloading period devised for *+spike benchmarks, faults are injected in the *+anomaly benchmarks during an equivalent period of 1 minute, every 10 minutes.

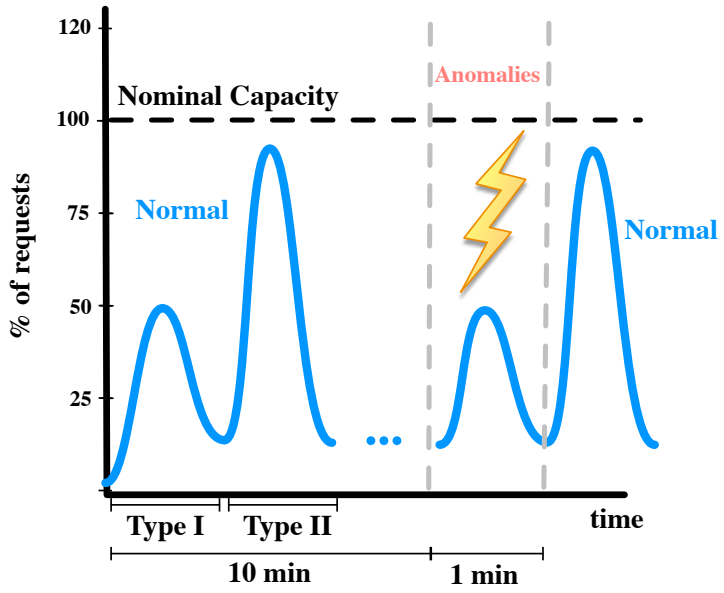


Figure 25: Structure of the Mix+anomaly, Popular+anomaly, Unpopular+anomaly and Mix+anomalyNet benchmarks.

Experimental tests are organized into scenarios. A scenario is delimited by the beginning of the period of normality and the end of the faulty period. Each scenario is associated to a single fault type: normal (no fault), CPU, Memory, I/O and Misc[6]. The association between faults and scenarios are performed sequently, in a way that one fault is only associated to a scenario n times when all the other faults have been already associated to previous scenarios $n-1$ times. This behavior is implemented by picking randomly from a list the fault type to be injected in a scenario, and only inserting it again in the list when all the other faults have been picked out.

Server application anomalies are induced by changing the video-streaming server code. We built an instrumented version of the original server application with CPU, memory and I/O faults equivalent to those used for inducing system anomalies. Thus, the stress functions are invoked within the server application, instead of being invoked by an external tool.

6 Misc faults are organic faults (not injected intentionally by the fault injection tool) activated during the experimental period.

4.5.3.3 *Mix+anomalyNet Benchmark*

The Mix+anomalyNet adds network faults to the list of faults covered by the Mix+anomaly benchmark. Network faults are injected in the form of packet losses and variance of packet delays, emulated using the Netem tool [Hemminger 2005]. The injection of network faults was calibrated through experimental analysis to induce network errors with several severity levels, giving a large amplitude of fault roughness in network failure scenarios.

Packet losses are emulated with a probability of occurrence between 10% and 50%, and a correlation between packet losses of 25% (i.e., the probability of a packet to be lost depends by a quarter on the last one). Correlation of packet losses emulates burstiness[7] of packet losses, a common phenomenon in the Internet [Jiang and Schulzrinne 2000]. The packet loss probability is modeled by (2).

$$\text{LossProbability(n)} = 0.25 * \text{LossProbability}(n-1) + 0.75 * \text{Random} \tag{2}$$

Packet delays are set to 100ms, with variance of 2000ms and a correlation of variance of 25%, and are emulated according to a normal distribution [Forbes et al. 2011].

4.5.4 *Training and Evaluation of Models Using Benchmarks*

Classification models used for failure prediction and failure diagnosis are learned using the mix+* benchmarks, because they have workloads closer to those found in real production systems. These benchmarks are denoted in this thesis as *learning benchmarks*. The other benchmarks are used exclusively for evaluation of models, when the workload varies in two opposite video popularity directions: popular and unpopular. These benchmarks are denoted in this thesis as *evaluation benchmarks*.

4.6 MONITORING

Monitoring is the foundation of failure prediction, diagnosis and repair activities. Self-healing systems implement the monitoring activity to:

- Obtain and prepare the log data to be provided to other self-healing activities;
- Perform failure detection through analysis of QoS metrics.

This section presents the monitoring activity incorporated in the self-healing video-streaming lifecycle devised for the Pure Streaming and HTTP Streaming infrastructures presented in this thesis. It presents the monitoring scope, log data gathered and evaluation results of the monitoring performance and overheads.

7 Temporal dependency between consecutive events.

4.6.1 *Monitoring Definition and Scope*

We define monitoring as the activity responsible for obtaining log data, preparing such data for analysis and performing failure detection. The log data contains values of metrics/parameters required for analysis. For succinctness, we refer to metrics/parameters only as metrics.

The log metrics are divided into:

- **System-level metrics**: system health, resource utilization levels and other global system metrics;
- **Application-level metrics**: application-level performance metrics relative to the video-streaming server;
- **Network-level metrics**: network performance metrics respecting the transmission of data between the server and players. These metrics include network interface statistics and TCP data transmission statistics;
- **QoS metrics**: metrics relative to the service quality provided to end-users.

The raw log data relative to these metrics have the following workflow:

1. Data preparation for analysis, by integrating, formatting and cleaning data;
2. Failure detection, through analysis of metrics representative of the service quality provided to end-users;
3. Data analysis by the failure prediction, failure diagnosis and repair activities;
4. Archiving of data for future analysis.

The monitoring activity should hold the following properties:

- **Timeliness**: video-streaming services are sensitive to performance variations. Hence, the entire monitoring lifecycle should be efficient enough to ensure short delays between the occurrence of errors and failures and their reflection on the data available for analysis;
- **Transparency**: monitoring probes should work with standard video players. That means that the monitoring activity should be transparent to players and codecs;
- **Coverage**: placement of monitoring probes should be devised to cover metrics representative of QoS and system, application and network performance.

The monitoring infrastructure should be designed to hold all these three properties. *Data gathering volume*, *sampling frequency* and *placement of monitoring probes* are main aspects of the monitoring apparatus that should be managed to ensure the aforementioned properties.

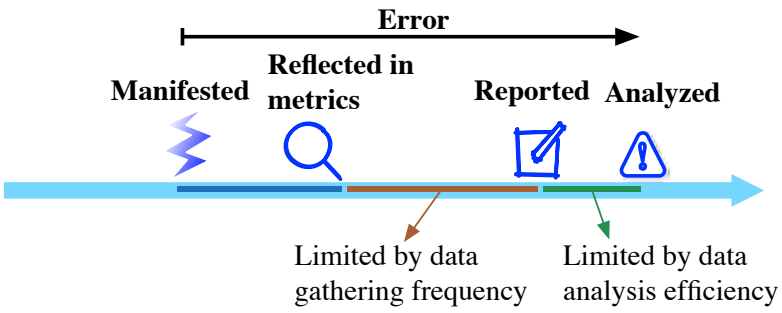


Figure 26: Breakdown of the monitoring delays into their contributors.

4.6.1.1 *Timeliness*

As illustrated in Figure 26, monitoring delays are decomposed into:

- The time required for error conditions to be reflected in metrics (constrained by the nature of the error);
- The time required for metrics to be reported — set by the data reporting frequency;
- The time required to analyze the reported metrics values. Efficient techniques for data preparation and failure detection are required to minimize the data analysis delay.

The monitoring delay will sum to delays added by other self-healing activities.

4.6.1.2 *Transparency and Coverage*

We consider the placement of monitoring probes at two levels: client-side and server-side. Client-side probes gather the data required for analysis of service quality provided to end-users. They present two challenges. Firstly, aggregation and analysis of log data from all clients can be unacceptably time consuming due to the volume of data involved, compromising the timeliness of failure detection and other self-healing activities. Secondly, client-side transparency is compromised, since all players are required to implement the monitoring functionality. This requirement is unviable in open systems accepting standard off-the-shelf player implementations. Server-side probes are placed to obtain server application metrics and system metrics. Network performance metrics can be also gathered server-side using TCP statistics and network interface statistics.

4.6.2 *Implementation of the Monitoring Activity*

Monitoring of the system performance in distributed systems is often done by off-the-shelf tools as Zabbix [Olups 2010] and Nagios [Barth 2008]. These tools monitor cluster machines, by periodically polling the servers to obtain data relative to service quality and performance metrics for analysis. These off-the-shelf monitoring tools are implemented as application-independent services running in the cluster. Some of them are also extensible to allow for development of customized server modules that provide application-level metrics for analysis.

Video-streaming players consume data in real-time and thus, impose strict time limits for the transmission of the video content requested. Therefore, performance degradation in these infrastructures should lead to responsive actions capable of restoring the normal performance levels within short periods of time. Hence, the monitoring solution chosen to integrate the self-healing lifecycle of video-streaming services should enforce these time constraints.

We decided for a monitoring solution built from scratch, instead of reusing a generic monitoring solution, for several reasons. Firstly, it is designed to provide guarantees of gathering, integrating, analyzing and providing monitoring data to self-healing activities within the time requirements imposed by video-streaming services. Secondly, it gathers application-level metrics and logs data in the format adopted by the self-healing infrastructure. Finally, instead of building a new monitoring service traversal to several servers, we integrate the monitoring activity with the other self-healing activities responsible for controlling each server. Thus, the server is promoted to an Autonomic Element with all self-healing dimensions incorporated.

4.6.3 *Design and Evaluation of the Monitoring Activity*

The monitoring activity integrated in our self-healing infrastructures is challenged by the coverage of metrics and the fault and design assumptions intrinsic to video-streaming services. Therefore, the following questions convey the design of the monitoring activity:

- What are the failure modes covered by the failure detector?
- Which QoS metrics are appropriate for failure detection in video-streaming services?
- How to monitor the service quality provided to end-users without compromising client-side transparency?
- Which metrics are required for the failure prediction, failure diagnosis and failure repair activities in the Pure Streaming and HTTP Streaming infrastructures?

- How to perform scalable and non-intrusive data gathering of metrics for analysis in video-streaming services?

Efficiency is an important characteristic of the monitoring apparatus. Therefore, the following questions addressing the monitoring performance should be answered through experimental evaluation:

- What is the expected monitoring overhead for the monitoring solution?
- What are the monitoring delays during periods of normalcy and during periods of server overload?

While the monitoring overhead represents the impact of the monitoring apparatus on the server performance, the monitoring delays determine whether the monitoring data can be timely provided for analysis, independently of the server condition.

4.6.3.1 *Failure Assumptions and Data Gathering Strategies for Failure Detection*

The failure model adopted in this thesis is agnostic of video encoding faults and client-side faults unrelated directly with the service. Hence, the service is considered correct when the server is accepting new requests (i.e., the server is responsive) and all data (i.e., video frames) transmitted to clients, for each request accepted, are transmitted on time for playback. Accordingly, incorrect service states are caused by server and server-side network faults leading to performance failures when providing video content to end-users.

As explained in Section 2.4, the assumption of in-order packet delivery guarantees at the transport protocol level, in video-streaming services, constrains the method and metrics adopted for measuring service quality. We assume the use of TCP for transmission of video-streaming data. This choice is justified by the universality of the service quality metrics used with this protocol and the widespread adoption of TCP for transmission of video-streaming data in modern Internet services.

Detection of hard failures is usually performed by an external agent that probes the service periodically to determine whether the service is responding to requests. On the other hand, detection of soft failures can be implemented either server-side — through comparison of video data rates transmitted by servers with data rates established by the video encoding for playback — or client-side — through analysis of player buffer states or connection establishment times.

Client metrics can be gathered from players or alternatively, by an external agent. In the latter case, the service is probed periodically by the agent using requests representative of the overall service quality. This approach is more appropriate for service monitoring than the analysis of the data gathered from all players. It is non-intrusive to end-users' players and has small overheads. Yet, it depends on the assumption of

probing representativeness. This assumption considers that service quality degradation affects equally the probing requests and all organic requests.

The analysis of server metrics for failure detection compares the transmission bitrate with the encoding bitrate of each video. This failure detection scheme enables the detection of soft failures transparently to end-users, for each request handled by the server. When compared with the use of client metrics for failure detection, it avoids: (1) the data gathering and data integration overheads required for analysis of all organic requests using client metrics; and (2) the representativeness assumption of probing requests.

4.6.4 *Performance Metrics and Parameters in Pure Streaming Services*

The monitoring activity should cover the entire request path. Client-side metrics can provide information for failure detection. On the other hand, the server-side metrics support the analysis of server performance problems even before they degrade the QoS seen by users.

As in any failure type, a performance failure starts with the fault activation, which gives rise to errors that propagate through the system up to end-users (Figure 27). In the error propagation process, performance degradation usually starts manifesting at the resource-level (e.g., CPU exhaustion). Then, resource problems turn out to application-level performance degradation (late transmission of video frames) before being reflected in the end-users' QoS (late reception of video frames).

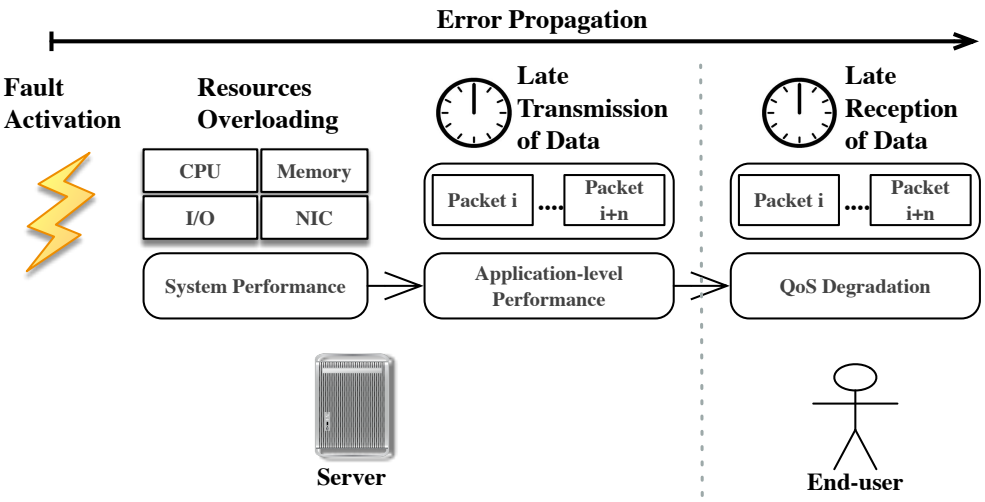


Figure 27: Propagation of Performance Errors.

Table 3 presents the system, network, server application and service quality metrics/parameters gathered by the monitoring activity.

Client	**Connection**		Encoding Bit-rate
			Establishing Time
			Buffering Time
			Dead Reason
System	**Global**	**Process**	**CPU +**
			Memory +
			Virtual Memory +
			I/O +
Application	**Interaction**		Desc-Setup
			Setup-Setup
			Setup-Play
	Scheduling		Frame Transmission Delay
	Aggregate		Number of Connections
			Throughput
Network	**Network Interface**		**Transmission +**
			Reception +
			Errors +
	TCP		**Transmission +**
			Reception +
			Errors +

Table 3: Metrics and parameters gathered by the monitoring activity. Groups of parameters are presented in bold.

4.6.4.1 *System and Network Metrics and Parameters*

System and network metrics/parameters are folded into groups, which are presented in bold in Table 3 for succinctness. As an example, the *CPU* group contains the *CPU Idle Time*, *CPU System Time*, *CPU User Time*, *CPU Waiting Time*, *CPU Nice Time* and *CPU Stolen Time* metrics. In the *network interface* class of metrics, the *transmission* group contains data transmission metrics, such as the *packets transmitted* and the *bytes transmitted*, and the *reception* group contains data reception metrics equivalent to the transmission metrics. The *error* group contains metrics representative of transmission and reception errors, overruns, collisions and packets dropped. The *TCP* class of metrics contains the statistics of TCP segments transmitted and received and also TCP errors, including, connection resets, retransmissions and errors on transmission and reception of TCP segments. A complete list of system and network metrics can be found in the SIGAR API documentation [sig 2012]. SIGRAR is used in the HTTP Streaming infrastructure for providing a comprehensive set of system and network metrics/parameters. The Pure Streaming infrastructure uses the same metrics gathered using the equivalent system commands.

Process-level metrics are gathered for the operating system process associated to the video server application. They expose the process state and resources being consumed by the server application (e.g., *CPU User Time* of the process). The rationale for the use of both global and process-level metrics is that, through analysis of both types of

metrics, it is possible to discern whether system performance problems are the reflex of server application problems or otherwise, are caused by other processes sharing resources with the server application process. As an example, a CPU utilization of 100% is caused by the server process when the equivalent process-level metric also reveals a similar CPU utilization level.

4.6.4.2 *Application-level Metrics*

Application-level performance degradation manifests at two request stages. In the first request stage, the connection establishment times increase, forcing users to wait more time than usual to start watching videos. In the second request stage, after connection establishment, the server send packets after their scheduled time. Accordingly, application-level performance is represented by two metrics: *interaction delays* and *scheduling delays*. Each of these metrics respects a request stage.

The interaction delays metric captures client-server interaction delays, observed when players issue commands to the server for establishing new sessions and also for stopping and resuming video playback. On the other hand, the scheduling delays metric captures delays on transmission of video fragments by the server to players, for the established sessions. The combination of both interaction delays and scheduling delays covers the overall application-level performance. Both metrics are gathered server-side using the Darwin Streaming Server monitoring module developed by us and through instrumentation of the server application.

4.6.4.3 *Interaction delays*

The interaction delays metric measures the time spent by the server handling the request at system and application levels, which includes the server queuing time and the server execution time of requests.

The request execution time is usually accessible from server modules. However, the time spent by each request in system and application queues is difficult to obtain. This problem creates the challenge of accounting the request reception delay [8] into the total request handling time. For guaranteeing transparency of the data gathering method, the total request handling time should be obtained without resorting to operating system or server application instrumentation.

We exploit the characteristics of the RTSP protocol to calculate the server request handling time. The use of several RTSP commands during the session establishment phase are exploited for that purpose. Hence, the delay between each pair of commands issued sequently by players is used as a client-server interaction performance metric in replacement of the request execution time. This metric will absorb both the request reception delay (queueing time) and the request execution time.

8 The time since the request arrives at the server until it is executed by the server application.

Figure 28 illustrates the process of calculation of the interaction delays metric. It shows the RTSP state machine and corresponding commands triggering transition between session states. Each interaction delay is calculated from the timespan between two sequential commands, issued during the session handshaking phase between the client and the server. Delays between DESCRIBE-SETUP, SETUP-SETUP [9] and SETUP-PLAY commands are averaged during each time interval to calculate the interaction delays metric. The DESCRIBE-SETUP delay, for example, is calculated as the difference between times $t2$ and $t6$ (the beginning of execution of two sequential RTSP commands), absorbing the:

- Relative request execution delay of the DESCRIBE command by the server — difference between the times $t2$ and $t3$;
- Relative request reception delay of the SETUP command by the server — difference between the times $t3$ and $t6$.

The relative values of interaction delays are similar to the relative values of request handling times. As an example, an increase of 5 seconds in the interaction delays metric is justified by an increase of 5 seconds in the server handling time, considering stable client and network delays. Thus, the use of the interaction delays as a metric representative of the server-side request handling performance requires the nullification of the effect of the network delay variance in its calculation. Thus, we calculate interaction delays only for synthetic requests issued by the probing agent in a controlled network environment. Therefore, the assumption of statistical invariance of both network and client-side delays allows considering the server the unique contributor of variance to interaction delays.

4.6.4.4 *Scheduling delays*

The scheduling delays metric represents delays on transmission of video-streaming packets (encapsulating groups of frames) by the server. Packet transmission delays are calculated relatively to their respective scheduled times, as illustrated in Figure 29. Pure Streaming servers schedule each packet p in advance, marking it to be sent at $Time(Schedule_p)$ to the respective client. When the server is experiencing degraded performance, it will likely start transmitting packets after their scheduling times. This scenario is illustrated for the delayed packet stored in the queue that is scheduled to be transmitted at the time $t0$, but it is actually transmitted later at the time $t1$.

Intuitively, the temporal distance between the transmission time and the scheduled time provides a simple indicator of the server performance for established client-server sessions. Accordingly, the severity of performance degradation is proportional to υ, cal-

9 SETUP of audio and SETUP of video channels belonging to the same session.

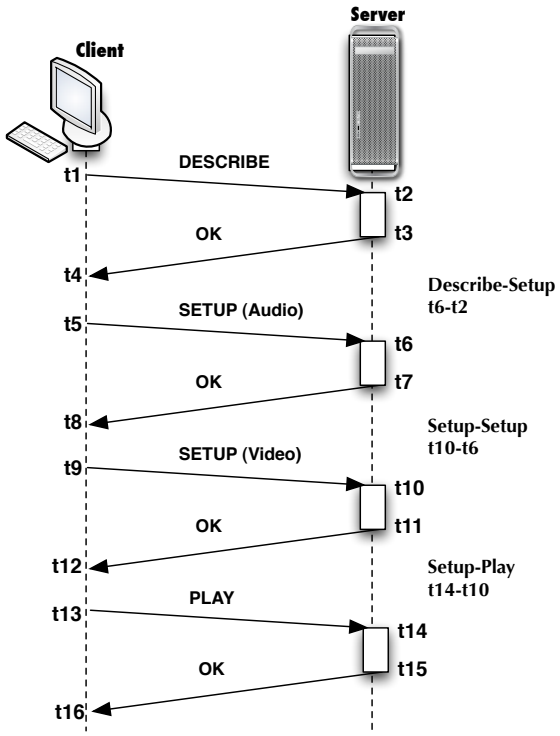


Figure 28: Calculation of interaction delays using the sequence of messages of the RTSP protocol.

culated in (3) as the mean difference between the scheduled time and the transmission time of all n packets transmitted by the server, during a given period of time.

$$v = \frac{\sum_{p=1}^{n} Time(Transmission_p) - Time(Schedule_p)}{n} \tag{3}$$

Similarly to the interaction delays, there is no direct correspondence between values of scheduling delays and QoS provided to end-users. Instead, these metrics expose the server application performance and are used by failure prediction and failure diagnosis techniques for modeling normal and anomalous system conditions.

4.6.4.5 *Service Quality Metrics*

Our failure detection approach uses QoS parameters obtained for synthetic requests issued by the probing agent. Synthetic requests are issued to fetch interaction delays and are used for failure detection, since they offer several advantages over organic requests, as such:

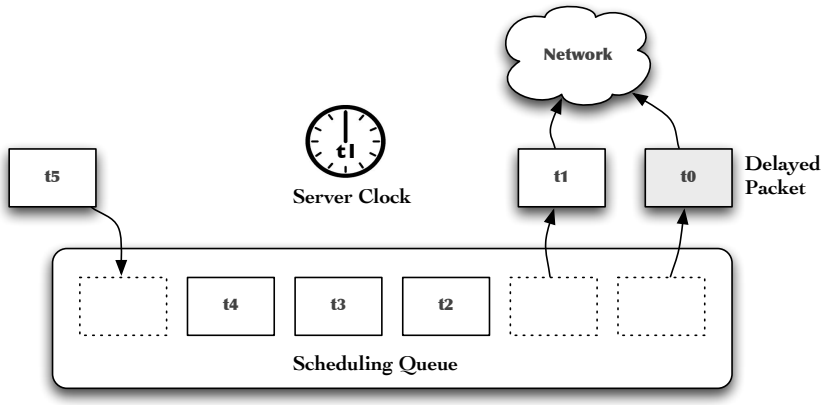


Figure 29: Scheduling delays during server performance degradation periods.

- **Manageable data volume** — service quality metrics and interaction delays are collected only for synthetic requests established and terminated at each n seconds;
- **Adoption of standard players** — players that do not require to be instrumented or be designed to provide service quality metrics;
- **Constant data gathering frequency** — data gathering frequency is ensured, even during periods without client-server interactions.

Generation of synthetic requests by the probing agent is constrained by:

- **Intrusiveness and time constraints** — the *number*, *length* and *encoding bitrate* of synthetic requests should be carefully specified to be low intrusive and to allow evaluation of service quality with enough frequency to ensure timely failure detection;
- **Isolation of server performance from the network and client interference** — synthetic requests are established between the probing agent and the server in a controlled environment. This assumption is required for calculation of interaction delays, because the server is seen as the unique responsible for performance degradation.

Synthetic requests should be representative of all organic requests. Representativeness means that the QoS of synthetic requests is consistent with the QoS of organic requests. QoS unbalancing between synthetic requests and organic requests is caused by differences on the impact of performance degradation on requests with different characteristics — e.g., different QoS levels observed for high bitrate and low bitrate

videos during periods of performance degradation. In these scenarios, synthetic requests will hardly represent the overall service quality provided by the server. Therefore, QoS probing is only viable when the impact of server performance on QoS is consistent in both synthetic requests and organic requests. Representativeness is a strong assumption of the monitoring infrastructure that should be validated experimentally for specific server implementations and configurations. We evaluate this assumption for our infrastructure in Section 4.6.6.

We use the Keynote StreamQ grade metric (presented in Section 2.4.3) for QoS evaluation. Thus, to calculate this metric, we gather the following service quality parameters over synthetic requests:

- **Connection time** — timespan between the moment the player sends a session request to the server and the moment it receives the confirmation of session establishment, by the end of the session handshaking phase;
- **Buffering time** — time required to fill the player buffer with enough data to start the playback, after the session handshaking phase;
- **Play time** — time spent playing the video;
- **Rebuffering time** — time required to refill the player buffer, when the video playback stalls due to the lack of buffering data.

The Keynote StreamQ grade metric establishes QoS degradation for frustration times (sum of the connection time with buffering and rebuffering times) higher than 6 seconds. Hence, we consider this value the threshold for a soft failure.

4.6.5 *Performance Metrics and Parameters in HTTP Streaming Services*

The SHStream infrastructure gathers the same system and network parameters used by the Pure Streaming infrastructure (shown in Table 3). SIGAR interfaces the operating system for gathering these parameters. More information about the system and network parameters gathered by SIGRAR can be found in [sig 2012].

Server application performance metrics and service quality metrics are specific to HTTP Streaming. Service quality metrics and parameters are also specified specifically for Progressive Download and Adaptive Bitrate services.

4.6.5.1 *Application-level Metrics*

SHStream gathers the following application-level performance metrics for each time period:

- **Average response time**. Average of the time since the server receives the request until the transmission of the first packet of the request-response to the client;

- **Number of degraded connections**. Number of request-responses with a transmission bitrate, in average, lower than their respective encoding bitrate;
- **Number of connections with negative gap**. Number of ABR request-responses transmitted after the playback time;
- **Standardized average of bytes written per session**. Average of bytes transmitted by the server, per request, standardized by the respective encoding bitrate;
- **Total bytes written**. Number of bytes transmitted to the network (payload data without protocol overheads);
- **Bytes read from disk**. Number of bytes read from disk by the server;
- **Number of active connections**. Number of requests being handled by the server;
- **Number of accepted connections**. Number of requests accepted by the server;
- **Number of connections recovered**. Number of requests redirected by the server to avoid congestion. This parameter is conditioned by the load admittance mechanism.

The *standardized average of bytes written per session* metric is a global server performance indicator. This metric is calculated in the left side of condition (4) and is lower than 1 during periods of service quality degradation. In that condition, *BitsTransmitted* represents the bits transmitted by the server to the player within each request since it starts at time t_0 until the present time t_i. When, in average, the sum of transmitted bits, for each active connection, since the beginning of transmission t_0 (i.e., time of the first byte transmitted), is smaller than the sum of the respective video encoding bitrate, during the same period of time, the service is in a degraded state. That means that, in average, the server transmitted less data to each player than the data required by it to play videos up to time t_i. The unit of time used to quantify data consumption is the second, as usually the encoding bitrate is quantified in multiples of bits per second (bps).

$$\frac{\sum_{j=1}^{nConnections} \frac{BitsTransmitted_j(t_0(j),t_i)}{(t_i - t_0(j)) \cdot EncodingBitrate_j}}{nConnections} < 1 \tag{4}$$

Application-level performance metrics are gathered using the mod_SHS module. Some of these metrics are already calculated internally by the Lighttpd and are grabbed by the mod_SHS module. The remaining metrics are calculated by the mod_SHS module, by accessing the Lighttpd internal state through instrumentation.

4.6.5.2 *Service Quality Metrics*

We use different metrics for detection of hard failures and soft failures in the HTTP Streaming infrastructure. Detection of hard failures is performed through analysis of the *number of failed connections* metric. This metric exposes the number of HTTP HEAD probing requests failed due to network server unreachability and HTTP 5xx errors.

Detection of soft failures in the SHStream infrastructure relies on metrics calculated server-side. Compared with the service monitoring approach using client-side metrics calculated for synthetic sessions, server-side metrics allow measurement of the service quality provided to all end-users, but disallows the evaluation of QoE.

The Keynote StreamQ metric adopted by the Pure Streaming infrastructure allows quantification of the user waiting time, one important metric to determine the QoE. By constrast, server-side metrics only suggest that end-users are either receiving or not the throughput required for maintaining playback continuity. However, both monitoring approaches are acceptable, since the main goal of failure detection in the self-healing infrastructure is the identification of failure conditions demanding recovery, caused by server and network problems.

Service quality metrics used for detection of soft failures are different for Progressive Download and Adaptive Bitrate services.

4.6.5.3 *Soft Failures in Progressive Download*

Soft failures are exposed by the *number of degraded connections* metric calculated in the mod_SHS module. This metric counts the number of requests with the ratio between the average server transmission throughput[10] and the corresponding video encoding bitrate inferior to 1. Accordingly, all requests satisfying the condition (5) are accounted by this metric.

$$\frac{\texttt{BitsTransmitted}(t_0, t_i)}{(t_i - t_0) \cdot \texttt{EncodingBitrate}} < 1 \tag{5}$$

The rationale of the *number of degraded connections* metric is intuitive. Client-side video playback consumes data at the same rate of the encoding bitrate. Therefore, the video encoding bitrate is the minimum server throughput required to guarantee continuity of video playback.

4.6.5.4 *Soft Failures in Adaptive Bitrate Services*

In Adaptive Bitrate streaming, each video segment delivered by one request respects a fixed and small period of playback time. Thus, one healthy server should ensure that each video segment is transmitted to the player before the previous segment has

10 Transmission bitrate provided for the request.

completed its playback. Accordingly, we consider that the player requests each video segment V_{i+1} at latest when the previous video segment V_i starts its playback. Thus, it is a fair assumption to consider that the server is failing to provide the service correctly when the *gap* defined as in (6) is negative.

$$gap = 10 - (R_{rec}(V_{i+1}) - R_{send}(V_{i+1})) \tag{6}$$

$R_{rec}(V_{i+1})$ represents the reception time of the next video segment and $R_{send}(V_{i+1})$, the transmission time of the request of the same video segment. We assume video segments with the duration of 10 seconds for the calculation of the gap. As a deduction, considering that the player issues the request for a specific video segment when it starts the playback of the previous one (worst scenario), the player receives the data for the requested segment after its playback time when the gap is negative.

The gap is calculated for segments of 10 seconds, but other segment sizes can be chosen. However, video segments with time lengths of 10 seconds are typical in ABR services. Smaller segment sizes have the disadvantage of the: (1) high number of I-frames, demanding more bits in the overall bitstream [Lederer et al. 2012]; and (2) small Groups of Pictures (GOP), providing a lower encoding performance and quality [Schonfeld 2011]. On the other hand, video segments larger than 10 seconds increase the adaptation time unnecessarily.

4.6.5.5 *Detection of Soft Failures*

Organic client requests are used in HTTP Streaming for detection of server-side performance failures. However, failing requests captured by the failure detector can be caused by:

- Performance problems originated client-side;
- Client-side network anomalies (e.g., last mile network failures).

A threshold established for the proportion of failed requests is required to separate server-side performance issues from client-side performance issues. In our experimental work, the detection of failures caused by server-side errors is performed based on the assumption that, during error-free periods, at least 95% of all requests handled by the server do not fail. This is an empirical value that do not compromise our experimental results and can be readjusted whenever it has been proven inappropriate in specific production services.

4.6.6 *Experimental Evaluation of the Monitoring Performance*

We evaluate experimentally the monitoring performance to determine the:

- Monitoring delays, including data gathering, data preparation and data analysis;
- Monitoring overheads, measured in terms of the reduction in the maximum number of requests handled simultaneously by the server when the monitoring activity is active;
- Representativeness of the QoS of synthetic sessions in the Pure Streaming infrastructure.

Experimental results provide a basis for interpretation of the results of other self-healing activities presented in the next chapters of this thesis. Monitoring delays are the most critical factor for the efficacy of the whole self-healing lifecycle. As the server load approaches maximum capacity, the monitoring delays could rise substantially due to scarcity of available system resources required to provide timely log data for failure detection, prediction, diagnosis and repair.

Since we ensure performance isolation between data gathering — provided by functionality attached to the server — and data analysis — performed in an independent virtual container or host machine — the variation in monitoring delays is explained only by variation in data gathering delays.

4.6.6.1 *Monitoring Delays*

Figure 30 presents the average monitoring delays for both Pure Streaming and HTTP Streaming infrastructures. Monitoring delays were obtained using the Mix+spike benchmark (Section 4.5.3.1) but limited to 180 simultaneous requests during the overloading period. The monitoring delays are presented for different number of connections (requests).

The Pure Streaming server reaches its nominal capacity at 110 connections, as evidenced in the graph, whereas the nominal capacity of the HTTP Streaming server is 140. In both infrastructures, the monitoring delays are maintained below 200 milliseconds, in average, up to the server nominal capacity. However, it is noticeable that in the Pure Streaming monitoring infrastructure, the monitoring delay increases significantly after surpassing the server capacity — monitoring delays reach approximately 1200 milliseconds in average (1400 milliseconds in the 95% percentile) for 140 connections.

Figure 31 presents the monitoring delays when the number of connections keeps rising, bringing also the HTTP Streaming server above its capacity. It is visible that during overloading states, the monitoring performance continues degrading in Pure Streaming — monitoring delays reach 15 seconds, in average, after 180 connections, increasing above 30 seconds in the 95% percentile. In the HTTP Streaming infrastructure, the monitoring delays keep at the same level, during normal and overloading periods.

The low monitoring delays in the SHStream infrastructure during overloading periods mean that:

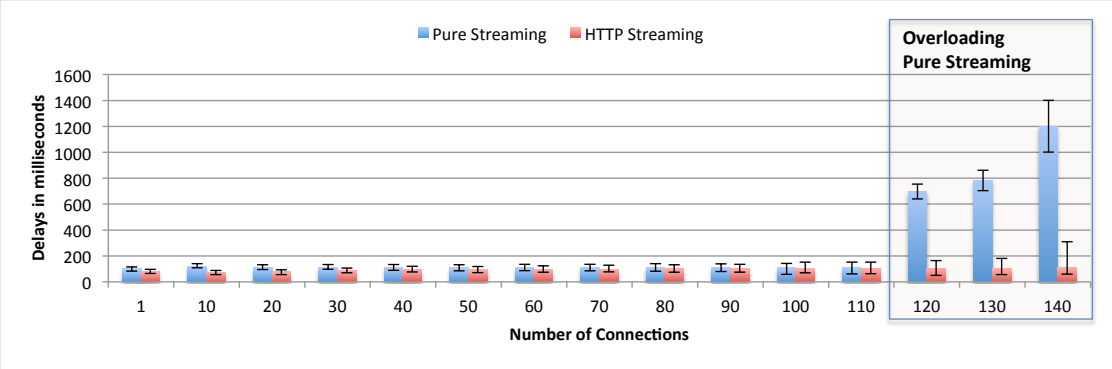


Figure 30: Average monitoring delays observed for Pure Streaming and HTTP Streaming. The error bars are presented for the 5% and 95% percentiles.

1. Server-side log data can be timely provided to the self-healing activities during overloading periods;

2. Container-based virtualization is effective in isolating the performance of the virtual container running the server from the performance of the self-healing activities running in the same virtual machine.

Although the monitoring performance is stable during overloading periods in the HTTP Streaming infrastructure, this result is specific to the server implementation. It shows that the application-level logs can be provided by the server's module during overloading periods. Consequently, since system metrics are gathered outside the virtual container (isolated from the server performance), the overall data gathering activity can be performed efficiently during overloading periods. Yet, the self-healing activities rely on the assumption of monitoring efficiency only during periods of normality.

The experimental results indicate that the data gathering process is efficient during normal conditions in both infrastructures, independently of the server load. Hence, proactive recovery can be ensured by both self-healing infrastructures, since log data are timely provided for failure prediction, failure diagnosis and repair during failure-free periods.

4.6.6.2 *Monitoring Overhead*

In the same way that the video server application can interfere with the performance of the monitoring activity, the monitoring performance overhead can also interfere with the server application performance. Monitoring overheads reduce the server capacity, due to consumption of resources by this activity stolen to the video server application. Therefore, with less resources available, the video server can handle a lower number of client requests simultaneously.

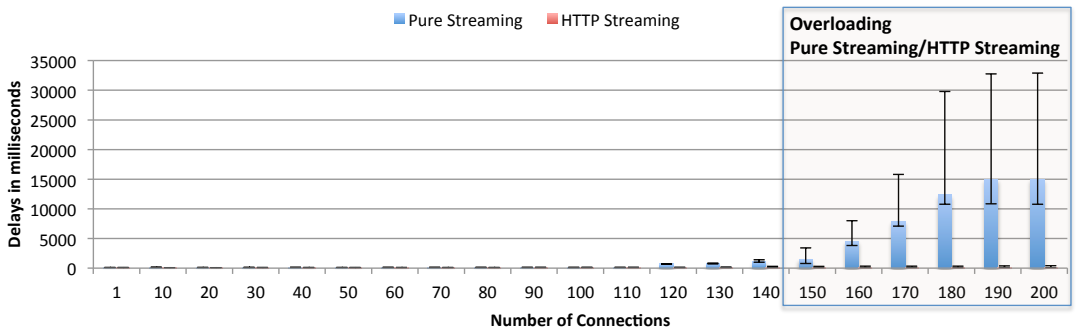


Figure 31: Average monitoring delays observed for Pure Streaming and HTTP Streaming, during normal and overloading periods. The error bars are presented for the 5% and 95% percentiles.

We evaluated the monitoring overhead during failure-free periods in Pure Streaming and HTTP Streaming servers, for several server load levels. The monitoring overhead is equivalent to the reduction in the maximum number of requests without degradation provided by the server after the activation of the monitoring activity.

Through experimental analysis, we observed that the maximum number of requests handled simultaneously by the HTTP Streaming infrastructure is the same either the monitoring is active or not. This observation was expected, seeing that the data gathering probes implemented by our Lighttpd module has low computational complexity and the HTTP HEAD requests issued by SHStream introduce relatively low performance penalties compared to video-streaming requests. By contrast, in the Pure Streaming infrastructure the server nominal capacity is decreased by 1 session. This capacity loss was expected, in view of the synthetic connections established by the probing agent for collecting service-level metrics.

4.6.6.3 *Representativeness of Synthetic Requests*

The QoS of synthetic sessions issued by the probing agent in the Pure Streaming infrastructure should be representative of the overall sessions being handled by the server. Using the results obtained for analysis of monitoring delays (Section 4.6.6.1) we observed that during overloading periods, the frustration time increases homogeneously over all synthetic and organic requests.

4.7 CHAPTER SUMMARY

This chapter presented the general self-healing approach, the self-healing infrastructures and the monitoring activity. It firstly described the self-healing problem space with the problems and assumptions in the basis of the development of the self-healing

activities presented in this thesis. Secondly, it presented the two self-healing infrastructures, one each for Pure Streaming and HTTP Streaming services, with the respective architecture, elements and activities. Thirdly, it explained the experimental methodology used to evaluate the self-healing activities presented in the next chapters of this thesis, along with the testbed and the benchmarks. Finally, the monitoring approach was described along with the QoS, system, network and server application metrics used by the monitoring activity and other self-healing activities presented in the next chapters of this thesis.

Results of the experimental evaluation of monitoring performance show that, during periods of normality, the monitoring delays in both the Pure Streaming and HTTP Streaming infrastructures are short enough to provide timely log data to the self-healing activities. Additionally, there are no monitoring overheads in the HTTP Streaming infrastructure, while the overheads in the Pure Streaming infrastructure are equivalent to 1 session of the server capacity, representing the synthetic session issued by the infrastructure to probe the QoS.

The infrastructures, monitoring activity, experimental methodology, testbed and benchmarks presented in this chapter are required by the following self-healing activities: failure repair (Chapter 5), failure prediction (Chapter 6) and failure diagnosis (Chapter 7). The next chapter addresses failure repair in video-streaming systems. It presents the way the repair activity integrates with the monitoring activity to achieve small recovery delays in the self-healing infrastructure.

5 FAILURE REPAIR

The real-time characteristic of video playback demands tight performance control over video-streaming systems to avoid playback interruptions. Performance control is implemented through efficient monitoring mechanisms that pursue timely detection of failures and efficient countermeasures to restore the service to appropriate levels with minimum impact on service quality.

This chapter addresses the problem of repairing video-streaming systems experiencing performance issues. Failure repair complements the monitoring activity in the self-healing infrastructures, to provide a failure recovery service that is not only able to gather log data and perform failure detection, but also to repair the service after each failure occurrence. This chapter also sets time requirements imposed by repair techniques to failure prediction and failure diagnosis techniques, addressed in Chapter 6 and Chapter 7, respectively.

5.1 FAILURE REPAIR APPROACH

Our self-healing infrastructures implement two classes of repair techniques that can be applied with both reactive recovery and proactive recovery strategies: *server migration* and *reboot*. Devising a solution that exploits these techniques to minimize the impact of recovery on the QoE of video-streaming services is the main objective of this chapter.

5.1.1 *Server Migration*

Self-healing systems should provide service continuity. The role of repair activities in the pursuit of service continuity is to reestablish the service with minimal downtime and preferably, without disrupting video-streaming sessions established between end-users' players and the server.

Video-streaming sessions are automatically resumed when the video players implement detection of service failures and automatic reconnection with video servers after failure occurrence. Therefore, the video players will reissue their connection requests, as soon as the service becomes available. Then, each player continues receiving video-streaming data, starting from the point in the video where it was left off when the service was interrupted. This behavior is common in Adaptive Bitrate streaming services, where the player issues independent requests for each small segment of the video. Thus, if one requests fails, the player reissues the request for the same segment. By contrast, the long request-responses of Progressive Download services require that

the server application and connection states have to be rescued server-side during the recovery process to ensure playback continuity.

Server migration techniques transfer the video-streaming service with the established client-server connections to a fallback host, when the active host starts showing signs of failure. This is an effective approach to counteract performance failures in video-streaming servers when:

1. Migration of server applications between hosts during failover is performed with small delays;
2. The state of the video-streaming server application and all client-server connection states can be reestablished after failover;
3. Faulty components are not migrated during failover, to avoid migration of errors to the target host;
4. Video-streaming data are transmitted progressively during long periods, as occurring in Progressive Download or Pure Streaming services.

Virtualization provides a facility for generic migration of virtual machines between hosts. By encapsulating processes, memory, connections and other structures reserved by the video-streaming application into a single checkpointable structure, it is possible to create an application-independent unit of migration between hosts.

5.1.1.1 *Server Migration Using Virtual Containers*

Container-based virtualization provides small server checkpoints, since it leaves the global operating system structures out of virtual containers. Virtual containers only embody structures required for their management, and private data of processes, such as the *address space*, *register set*, *opened files/pipes/sockets*, *IPC structures*, *current working directory*, *signal handlers*, *timers*, *terminal settings*, *user identities* (e.g., *uid* and *gid*) and *process identities* (e.g., *pid*, *pgrp*, *sid* and *rlimit*).

Failures caused by server performance problems and server-side network problems would potentially benefit from recovery using migration of virtual containers between hosts. However, only server performance problems originated outside the virtual container (at the operating system level or in other virtual containers) can be avoided by migrating the virtual container to another host. That means that migration is ineffective when errors originate within the virtual container or are propagated to the virtual container. Errors resulting from the activation of faults within the virtual container can be repaired by rebooting it.

5.1.1.2 *Checkpoint Replication Strategy*

We adopt the single checkpoint replication strategy, due to the large overheads of synchronous checkpointing (described in Section 3.3). Thus, the checkpoint should be

transferred to the fallback host within the time interval between the detection time of each soft failure and the further occurrence time of a hard failure.

Single checkpoint replication is unable to provide recovery guarantees, so it can be complemented with reboot techniques (detailed in Section 5.1.2). Therefore, when the time window available before the occurrence of hard failures is insufficient to rescue the server checkpoint, the service is reestablished using a rebooted server in the same host or in the fallback host, at the cost of disruption of client-server connections.

A proactive recovery strategy is also explored to increase the window of opportunity available for rescuing the server checkpoint before the occurrence of hard failures. In proactive recovery, the checkpoints are performed during the look-ahead time window provided by failure prediction (addressed in Chapter 6). Thus, both soft failures and hard failures can be anticipated and avoided. Failure prediction provides an opportunity to handle failures gracefully, without impacting the users' experience.

5.1.1.3 *Server Migration Technique*

In recovery scenarios, the reduction of virtual machine migration times should have priority over the reduction of service downtimes caused by the migration process. When the server starts experiencing performance problems, the virtual machine has to be transferred to the fallback host as fast as possible. Reduction of virtual machine migration times would increase the chance of moving the virtual machines' checkpoints to fallback hosts before:

- The occurrence of hard failures, in reactive recovery scenarios. Otherwise, hard failures occurring in the middle of this process will disallow the recovery of server application and connection states in the destination host;
- The occurrence of soft failures, in proactive recovery scenarios. In this scenario, the anticipation time provided by failure prediction is used to migrate the server before end-users start experiencing failures.

Notwithstanding live migration techniques (Section 3.3) introduce small downtimes when used during failure-free periods, they are inappropriate for recovery purposes, due to the large migration times imposed by progressive data synchronization between hosts [Clark et al. 2005][Che et al. 2010]. Instead, we use stop-and-copy migration to migrate the server from the active host to the fallback host.

Stop-and-copy migration simplifies the migration process and reduces the time necessary to rescue the virtual container, by suspending it (bringing it to a stable state) and copying its state (memory, network and processes) to the fallback host at once. Also, by stopping the service before starting the migration process, a larger amount of system and network resources will be available for supporting the migration process. Hence, the probability of successfully rescuing the checkpoint increases along with the reduction of the time required for its execution.

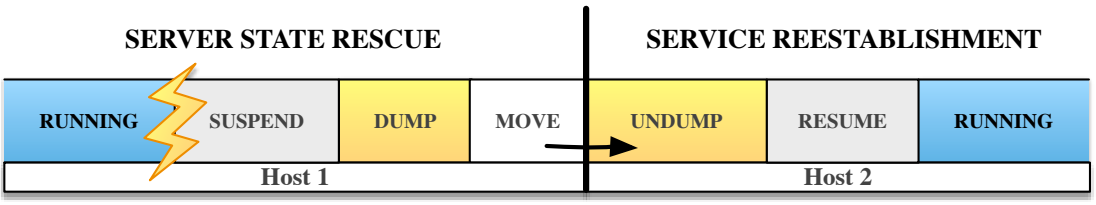


Figure 32: Migration phases and respective actions.

The stop-and-copy migration process is broken down into two main phases illustrated in Figure 32: *server state rescue* and *service reestablishment*.

The server state rescue phase represents a critical step to reestablish the service without disruption of client-server connections. This phase is responsible for rescuing the server's checkpoint to the fallback host. If the server fails to complete this phase before the occurrence of hard failures, the service cannot be resumed later in the fallback host, using the checkpoint. The server state rescue phase is composed by the *suspend*, *dump* and *move* actions. The suspend action moves virtual container's processes to a beforehand known state and stops the network interfaces. The dump action executes the checkpoint that saves the state of individual processes and the global state of the virtual container to a dump file. The move action transfers the dump file to the fallback host, completing the server state rescue phase.

During the service reestablishment phase, the checkpoint file is undumped in the fallback host and is further resumed to continue its execution. This phase involves reconstructing all in-memory structures required to reestablish the service from the checkpoint.

5.1.2 *Reboot-based Recovery*

Recovery using server migration techniques can ensure the continuity of video streaming connections established between clients and the server. However, disruption of client-server connections established for video-streaming requests are inevitable when hard failures arise abruptly without allowing migration of the server's checkpoint to the fallback host. In the same way, performance failures caused by server application faults cannot be recovered through server migration. The reason is that checkpointing techniques would replicate in-memory server states with errors to the fallback host, avoiding the recovery of the server.

Despite some previous work has addressed the problem of repairing applications at runtime by changing them dynamically [Fuad et al. 2006][Carzaniga et al. 2008][Portokalidis and Keromytis 2011], none of them is generic, provides full recovery of applications and guarantees application correctness after recovery. By contrast, reboot-

based techniques have shown a singular ability for overcoming transient failures, by renewing the system and/or server application.

Reboots are generic recovery solutions with low complexity that are implemented in our HTTP Streaming infrastructure for overcoming: (1) application-level failures; and (2) system-level failures that are not detected with the necessary anticipation to finish the checkpoint of the server application before the occurrence of hard failures.

5.1.2.1 *Impact of Reboots on the Service*

Reboot techniques are more disruptive than server migration techniques, because they force the termination of all video-streaming connections. Typical Progressive Download request-responses are long and the clients are bound to the server during the download of the entire video. Thus, after the execution of a reboot operation, the end-users may have to interact with the player application to reissue the requests again.

In typical Adaptive Bitrate streaming services, the impact of reboot on the service quality can be comparable to that of server migration. The reason is that requests can fail and be issued again without compromising the service quality. Each of the video segments can be requested again before its playback in case of failure, considering that reboots introduce small server downtimes.

5.1.2.2 *Reboot Granularity*

Reboots should be performed with the appropriate granularity to avoid ineffective recovery actions or recovery costs larger than required. An operating system reboot is expensive and its cost is proportional to the server downtime caused by the execution of several activities: stopping all services, restarting the operating system, starting the services again and warming-up the server. However, reboots can be performed at a finer granularity than the operating system. Often, errors accrue at the server application level and, in many cases, the kernel goes back to a consistent and clean state simply by killing and revoking the resources of the faulting process [Yoshimura et al. 2011]. Thus, whenever it is possible, only the server application should be rebooted, since it presents significantly smaller costs.

Even when rebooting the server application (or the VC where it runs) is enough to recover the service, the service downtime can be large enough to force the abandonment by end-users. The reason is that the service downtime includes the reboot time and the partial downtime incurred due to the limited server's capacity expected before the end of its warm-up period. Hence, at least during the warm-up period, the rebooted server requires the assistance of an alternative server to assume part of the load.

Operating system reboots are required when errors originate at — or are propagated to — the operating system level. This reboot granularity introduces larger server downtimes and server warm-up times than server application or VC reboots, as the entire

operating system should be rebooted and warmed-up along with the server application. To circumvent this problem, the service should be backed by alternative hosts during recovering periods.

5.1.2.3 *Challenges*

The most important challenges of the execution of server migration and reboot strategies concern the execution delays of these techniques and the service downtime caused by their execution. During the server migration process, the checkpoint rescue time should be minimized to increase the probability of being completed before the occurrence of hard failures. Also, the service downtime introduced by the recovery process should be minimized in both recovery strategies to avoid client-side interruptions of video playback and to minimize the periods of server unresponsiveness when players issue new video-streaming requests.

After rescuing the active server to the fallback host, the server should be warmed-up appropriately before receiving the entire active server's load. Thus, we avoid failures caused by handling loads in the fallback server larger than those supported by it before the end of the warm-up period. However, during the warm-up period, the remaining load not received by the fallback server should be handled by another server (e.g., the active server). So, the warm-up period is accounted as partial server downtime in the fallback host, which should be minimized.

5.2 RESEARCH QUESTIONS

The design of the repair solution is guided by the following research questions:

1. How to implement and integrate the server migration and reboot techniques into the SHStream infrastructure?
2. What is the downtime generated by server migration?
3. What is the look-ahead time before failure occurrence demanded by server migration for rescuing the checkpoint of the server state to the fallback host?
4. What is the breakdown of the server migration time into the times required to checkpoint the server application to the local disk, transfer the checkpoint to the fallback host over the network and start the video-streaming service in the fallback host?
5. What is the impact of server migration on the service quality?
6. Which reboot techniques are more efficient?
7. How to choose the appropriate reboot granularity?

8. What is the service downtime generated by reboots?
9. How to warm-up the server after executing the repair action without introducing significant server downtimes?
10. How to delimit the server warm-up period?
11. What is the length of the server warm-up period for server reboots and operating system reboots?

The experimental evaluation of the repair approach should answer these questions.

5.3 IMPLEMENTATION OF SERVER MIGRATION IN SHSTREAM

To reduce the virtual container migration time, one copy of its basic structure should coexist in the origin and destination hosts. That structure contains a stopped replica of the virtual container with its filesystem. Maintaining both replicas of the file system synchronized is essential to avoid unnecessary transference of files between hosts during migration. The video files stored inside the virtual container are examples of files that should coexist in both hosts before starting the migration, to reduce virtual container migration times.

In our implementation of stop-and-copy migration, we maintain a copy of the initial virtual container in the fallback host and only copy the in-memory structures (i.e., checkpoint data) during migration. Transference of checkpoints between hosts is performed using *rsync* [rsy], a popular tool for efficient incremental file transfer.

5.4 IMPLEMENTATION OF REBOOT-BASED RECOVERY IN SHSTREAM

Our recovery approach performs reboot of servers at two granularity levels: (1) *virtual container* and (2) *operating system*. We design a two-phase reboot strategy (Figure 33) that starts by rebooting the faulty VC. Then, a full operating system reboot is executed, if the faulty behavior persists after the VC reboot. This multi-phase reboot strategy aims to reduce recovery costs by avoiding full reboots when errors are confined to specific VCs. A VC reboot offers several advantages over a operating system reboot:

- It is less expensive, since it has smaller reboot delays;
- It reduces the server warm-up period significantly, since the kernel in-memory structures are isolated from the VC reboot process;
- It can be performed without additional resources provided by the actual host or other hosts selected to assist the reboot process.

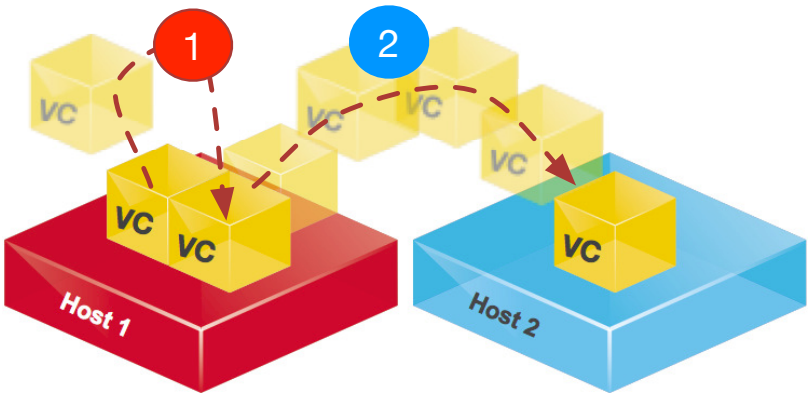


Figure 33: Two-phase reboot process.

Our solution combines the execution of VC and operating system reboots with a further server warm-up period, for recovering the server without compromising the service quality provided to end-users.

5.4.1 *Virtual Container Reboot*

Reboots at the virtual container level aim to refresh the execution context of the server application, using one of the following techniques:

1. *Restart the video-streaming application process* — restarts the operating system process attached to the server application;
2. *Restart the virtual container* — restarts the virtual container where the server application process is running along with all internal processes;
3. *Start a fresh replica of the virtual container* — stops the virtual container and starts a rebooted replica in the same or in another host. This technique requires a replica of a rebooted VC (a snapshot of a VC after a previous reboot) — henceforth presented as *secondary VC* — to replace the active VC — henceforth presented as *primary VC*.

We choose the last technique for recovering VCs because it is more efficient than the other techniques — as demonstrated by the experimental results presented in Section 5.8 — and also because it supports our server warm-up implementation.

5.4.2 *Operating System Reboot*

An operating system reboot is significantly more expensive than a VC reboot. However, it is an appropriate recovery technique when performance anomalies originate in the operating system.

Operating system reboots are accompanied by the instantiation of a VC replica into another host, to handle the server load during the reboot process. We use the term *primary host* to refer to the faulty host and *secondary host* to refer to an alternative host that assists temporarily the recovery process of the primary host.

The operating system reboot process has two phases. In the first phase, a replica of the primary VC (secondary VC) is instantiated into the secondary host. Then, the requests are redirected progressively until the secondary VC is warmed-up and takes the IP of the primary VC, similarly to a VC reboot. Finally, a full operating system reboot is executed. The second phase initiates when the primary host finishes the operating system reboot process. Then, the entire process followed in the first phase is executed again to move the server back to the primary host.

5.4.3 *Delimitation of the Server Warm-up Period*

Despite reboot techniques can be applied to any HTTP Streaming technology, only ABR services can potentially benefit from our warm-up approach. Since the size of ABR request-responses is small, video players have to issue requests with high frequency, allowing temporary redirection of requests between servers and performance control over each request delivered. Thus, these characteristics of ABR streaming proportionate:

- Progressive protocol-level redirection of requests to warm-up the secondary server during recovery;
- Analysis of variance of request-response delays[1] to delimit the server warm-up period. This process determines when the secondary server rebooted is ready to accept the full load of the primary server.

The server warm-up period should be respected to avoid failures caused by the transition from the primary VC to the secondary VC rebooted. The VC warm-up process starts with the creation of the secondary VC with its own IP address, in the same host of the primary VC. Afterwards, the web server running in the primary VC redirects part of the requests to the secondary VC, using the HTTP REDIRECT method (Figure 34). The number of redirected requests grows progressively to warm-up the server running in the secondary VC. Finally, the primary VC is destroyed and the secondary VC becomes the primary VC, by taking its IP address.

1 Difference between the time the client transmits the request until it receives the response.

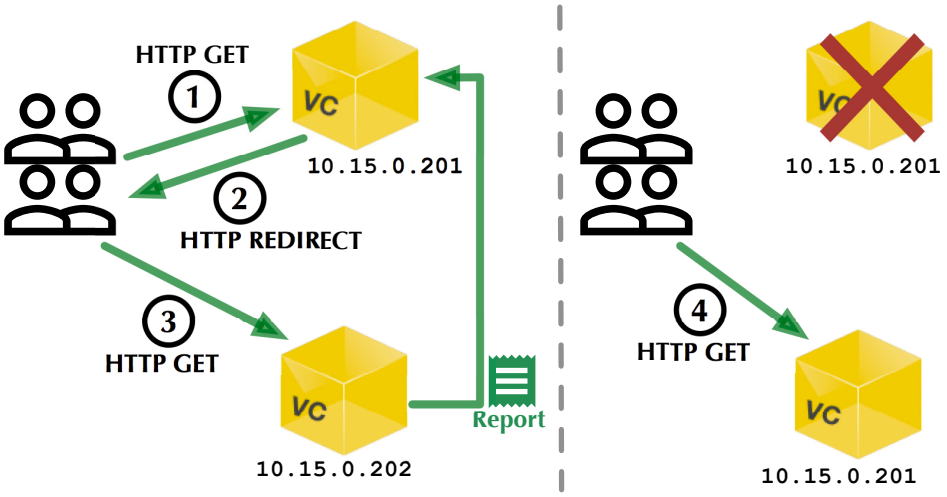


Figure 34: Progressive migration of requests to warm-up the server running in the secondary VC.

5.4.3.1 *Insight Behind the Server Warm-up Approach*

Figure 35 relates the running standard deviation [Knuth 1997] of request-response delays with the number of video requests handled by the server after the reboot, using a variant of the mix+spike benchmark without the overloading period. It is noticeable a pattern of large standard deviations of request-response delays during the server warm-up period. The variability of these values is dictated by the large delays of some request-responses.

Request-responses with large delays can be responsible for user visible failures during the server warm-up period. Figure 36 shows the gap between the length of each video segment streamed (10 seconds) and its download delay, calculated as in (6), for the same dataset of Figure 35. It exposes several requests with negative gaps during the server warm-up period, representing potential failures experienced by streaming users.

It is noticeable that the variability of request-response delays stabilizes after a certain number of requests have been handled by the server. We use that characteristic to control the transference of load between the primary server and the secondary server during the server warm-up period. Thus, the server warm-up period is delimited through analysis of variance of request-response delays. For that purpose, we formulate the hypothesis that the statistical distribution of request-response delays of the secondary VC is similar when the server is warmed-up, but different during the server warm-up period.

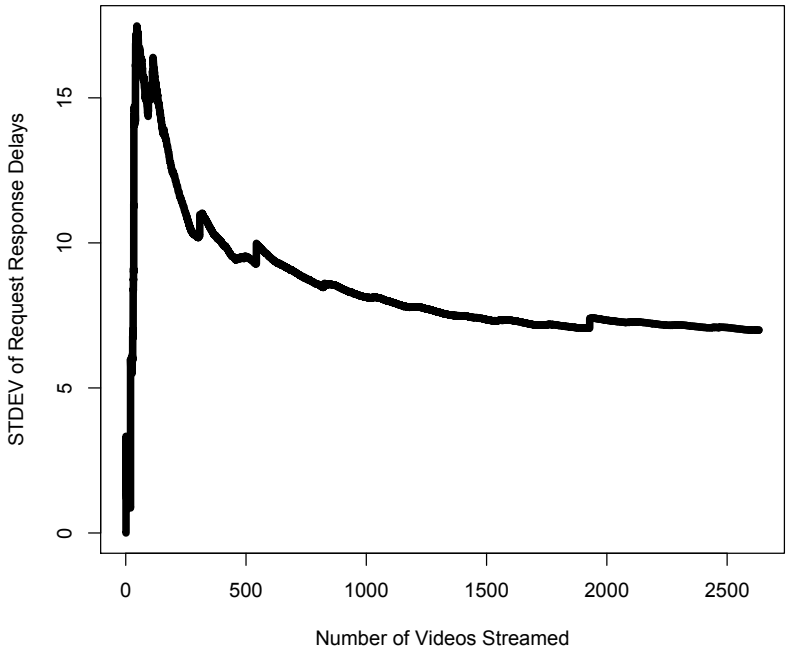


Figure 35: Relation between the standard deviation of request-response delays and the number of complete videos streamed after the reboot.

5.4.3.2 *Server Warm-up Approach*

During recovery, the primary server redirects the load progressively to warm-up the secondary server, and handles the remaining load. At each warm-up stage, the primary server redirects more requests to the secondary server (randomly selected) than in the previous stage, increasing in the proportion of L% of the primary server's capacity. The resulting load of the secondary server is only increased again after stabilization of the variance of request-response delays. This incremental process intends to avoid transient failures during the server warm-up period and to warm-up the secondary server with a realistic workload taken from the primary server.

Figure 37 shows the distribution of the load between the primary and secondary servers along time, during the warm-up period of the secondary server. The primary server controls the warm-up process, by redirecting requests and deciding when to increase the number of redirected requests. That decision is based on the analysis of server request-response delays gathered from the logs of the secondary server.

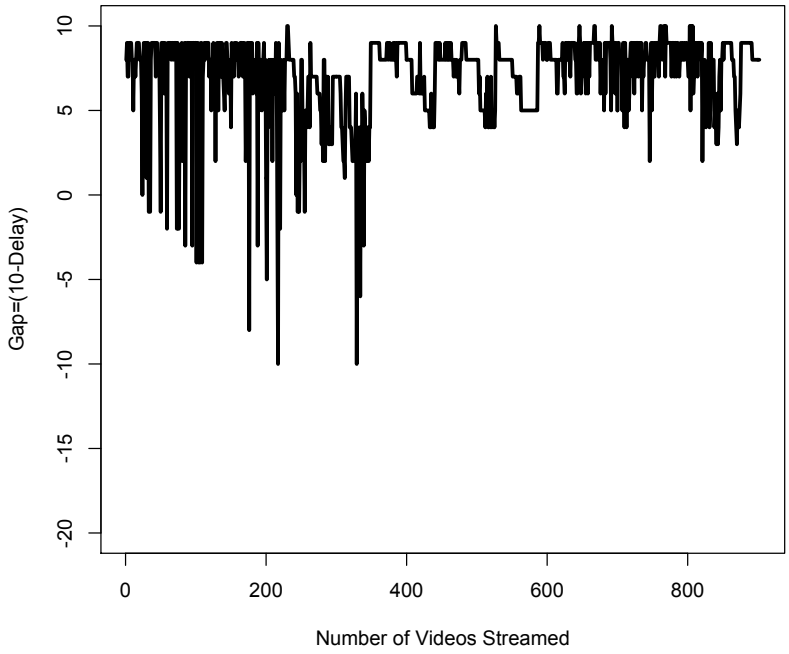


Figure 36: Difference between the length of video segments (10 seconds) and each request-response delay, after the reboot.

We use the Kruskal-Wallis method to compare statistical distributions of request-response delays. It is an efficient non-parametric method[2] used to test whether several groups of samples originate from the same distribution. The test statistic is given by (7), being n_i the number of observations of the group i with n samples, r_{ij} the rank of the observation j from group i, $\bar{r}_i$ and $\bar{r}$ calculated as in (8) and N the total number of observations.

$$K = (N-1)\frac{\sum_{i=1}^{g} n_i(\bar{r}_i - \bar{r})^2}{\sum_{i=1}^{g}\sum_{j=1}^{n_i}(r_{ij} - \bar{r})^2} \tag{7}$$

$$\bar{r}_i = \frac{\sum_{j=1}^{n_i} r_{ij}}{n_i} \qquad \bar{r} = \frac{1}{2}(N+1) \tag{8}$$

The evaluation of our server warm-up approach depends on the observation that:

2 Does not assume that the data are normally distributed.

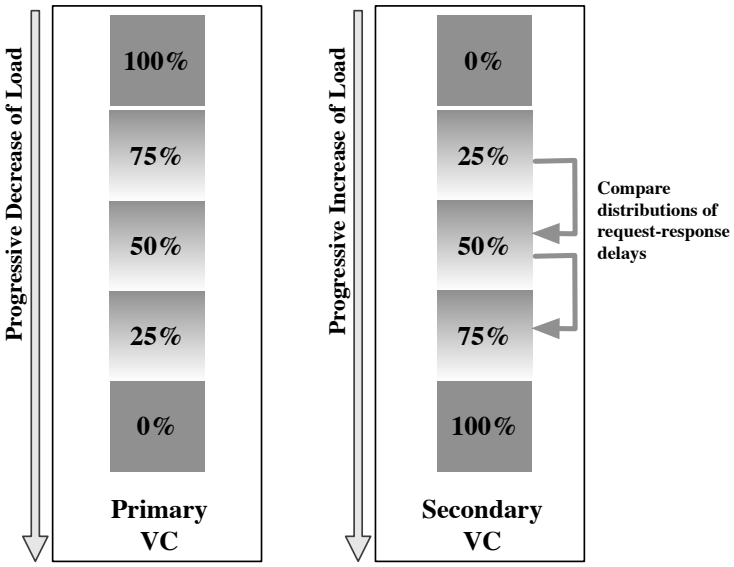


Figure 37: Distribution of load between the primary VC and secondary VC and comparison between the statistical distributions of request-response delays of different server load levels.

- For the lowest load level (25%), the Kruskal-Wallis test rejects the null hypothesis that the distributions of request-response delays, belonging to each pair of time-adjacent groups of request-response delays, are similar during the server warm-up phase;
- For higher load levels ($>$ 25%), the Kruskal-Wallis test rejects the null hypothesis that the distribution of each group of request-response delays, gathered during the server warm-up phase for the current server load, is similar to that of the last group of request-response delays belonging to a lower server load level already validated by the warm-up process.

When the null hypothesis is not rejected, the server is considered warmed-up for the provided load level and consequently, the load is increased in the secondary server. The server warm-up process finishes when more than 75% of the requests are being redirected to the secondary server.

5.4.3.3 *Data Gathering for Analysis*

We store the access log of the secondary server in a folder shared between both VCs, to provide data for analysis during the reboot of VCs (Figure 34). For operating system reboots, the content of the access log of the secondary server is synchronized between hosts using the rsync tool, at each 5 seconds.

5.5 SELECTION OF THE SECONDARY HOST

Operating system reboots and server migration techniques require the assistance of other hosts to receive the video server during the recovery period. Any host with available resources to handle the load of the faulty VC can be used to that end. Thus, the secondary host can be either one passive machine specifically dedicated to assist the recovery of other hosts or any of the active machines with available resources.

Container-based virtualization infrastructures provide statistics about utilization of system resources for each VC. These statistics are useful to select one of the active hosts as the secondary host. Host selection activities are performed through comparison of resource utilization statistics provided for the primary VC with equivalent statistics provided for VCs running in other hosts. In other words, any host with idle resources sufficient to afford the consumption of resources by the primary VC can assist its recovery.

OpenVZ, for example, measures the utilization and capacity of resources for each VC using *beancounters*. Beancounters represent the units of utilization of resources and are presented as such:

```
vzctl exec 101 cat /proc/user_beancounters

uid  resource         held    maxheld    barrier      limit
101: kmemsize       803866    1246758    2457600    2621440
     lockedpages         0          0         32         32
     privvmpages      5611       7709      22528      24576
     shmpages           39        695       8192       8192

         [...]
```

The *held* column represents the current resource utilization, the *maxheld* the maximum resource utilization since the last VC reboot and the *barrier* and *limit* represent the capacity of the given resource — the distinction between the last two metrics is specific to each resource.

The OpenVZ statistics provided for each VC enable the selection of eligible secondary hosts based on resources not consumed by their VCs. One straightforward method for selecting the secondary host is to find a host where the *maxheld* value of all resources in the primary VC fits the available resources of one VC running in the secondary host, as formulated in (9). Thus, a replica of the primary VC can be instantiated in the secondary host to assist the reboot of the primary host.

$$\mathrm{maxheld}_{\mathrm{primary}} < \mathrm{limit}_{\mathrm{secondary}} - \mathrm{maxheld}_{\mathrm{secondary}} \tag{9}$$

The host selection process depends on a data sharing mechanism that provides the faulty servers with resource utilization statistics of the other hosts. That mechanism can be implemented by a reporting service accessible to all hosts to report resource

utilization statistics periodically. The design of that service is out of the scope of this thesis.

5.6 EXPERIMENTAL METHODOLOGY

This section presents the experimental methodology followed to evaluate our repair approach. It presents the experimental testbed, benchmarks and evaluation metrics, used to answer the research questions stated in Section 5.1.

5.6.1 *Evaluation Benchmark*

We use two benchmarks for the evaluation of server migration techniques: *popular benchmark* and *unpopular benchmark*. These benchmarks have the same specification of the *popular+spike* and *unpopular+spike* benchmarks presented in Section 4.5.3, respectively, but:

- Do not have the overloading period (the spike period);
- The virtual container is migrated every 10 minutes;
- The number of requests ranges from 1 up to the maximum number of simultaneous requests supported by the server.

These benchmarks allow the evaluation of server migration with workloads positioned in the extremes of temporal locality: a cache-intensive workload and a disk-intensive workload. Despite the low probability of being observed in real production services, these workloads are required for knowing the worst-case recovery behaviors associated to the limits of the recovery performance.

The mix benchmark is used for evaluation of reboots. This benchmark is a variant of the mix+spike benchmark without the overloading period. As explained in Section 4.5.3, the workload associated to this benchmark is designed to approximate real workloads, which makes it adequate to understand the server behavior during the warm-up phase. The performance of server reboots are evaluated by rebooting the server every 10 minutes.

5.6.2 *Server Migration in HTTP Progressive Download*

We use the testbed described in Section 4.5 for evaluation of server migration between hosts in HTTP Progressive Download services. The experimental evaluation process addresses the analysis of the impact of server migration on the service provided to end-users, measured as such:

Algorithm 1 Sequence of instructions executed to evaluate server migration.

```
loop
    sleep(600)
    ssh(host1, suspend|dump|rsync)
    ssh(host2, undump|resume)
    sleep(600)
    ssh(host2, suspend|dump|rsync)
    ssh(host1, undump|resume)
end loop
```

- Time required to execute each server migration step;
- The impact of server migration on the service quality metrics.

Migration of virtual containers between hosts is controlled by Secure Shell (SSH) commands issued by a shell script. Algorithm 1 describes the sequence of instructions executed by the shell script during the experimental tests.

5.6.2.1 *Evaluation Metrics*

Experimental evaluation of server migration is done using two types of metrics: *recovery delays* and *service quality*. Recovery delays metrics represent the server downtime and are divided into:

- *Server state rescue time* — the time spent rescuing the checkpoint. It sums the time required for checkpointing the virtual container locally to the time required for transferring the checkpoint data to the destination host;
- *Service reestablishment time* — the time required to reestablish the service in the destination host.

Service quality metrics are gathered at every time interval, for evaluating the impact of server migration on the service quality experienced by end-users. We consider the following service quality metrics:

- *Number of degraded connections* — measures the number of soft failures manifested as playback interruptions. This metric counts the number of connections with transmission bitrates smaller than their encoding bitrates (the minimum bitrate required to play videos without interruption of playback);
- *Standardized average of bytes written per session* — reveals the global server condition, represented by the average of the transmission bitrate of all sessions, standardized by the respective encoding bitrate.

Service quality metrics are calculated in the monitoring activity and are described in detail in Chapter 4.

5.6.3 *Reboot in ABR Streaming*

Our reboot approach is evaluated in scenarios in which: (1) the server is rebooted in the same host, using one of the techniques presented in this chapter; and (2) a rebooted replica of the virtual container is instantiated in the secondary host.

5.6.3.1 *Evaluation Metrics*

Reboot is evaluated for its efficacy and efficiency using the following metrics:

- *Total recovering time* — includes the reboot and server warm-up times;
- *Total reboot delay* — time required to reboot the server, virtual container or operating system;
- *VC suspend delay* and *VC start delay* — time required to suspend and instantiate a VC in the same or in another host, respectively;
- *Number of failed requests served by the secondary server* — represents the efficacy of the server warm-up approach;
- *IP takeover time* — time required by the secondary VC to take the IP of the primary VC.

Despite not impacting the service directly, the total recovering time has a direct impact on the risk of failure. Typically, performance anomalies lead to the increase of the severity of failures along the time. Hence, the occurrence of hard failures could compromise the service quality provided to some users, since the server is involved in the recovery process. In that scenario, the warm-up phase in the secondary VC is interrupted, forcing the secondary VC to take the IP of the primary VC and handle its entire load without being warmed-up. So, to minimize the risk of service failures caused by a server not correctly warmed-up, the recovery time should be minimized as well.

The number of failed requests in ABR streaming services is equivalent to the number of requests with negative gaps, defined as in (6). These requests represent potential interruptions of video playback, seeing that the gap represents the time distance between the reception of one video segment and its playback time, in the worst-case scenario. Therefore, to avoid failures, the downtime created by the recovery process should be smaller than the gap of each video segment obtained without reboots. However, since the gap varies for each request, we consider enough to have service downtimes significantly smaller than the time length of segments (10 seconds) to reduce the chance of being larger than the gaps.

The service downtime is the sum of the time required to suspend the primary VC with the time required by the secondary VC to take the IP of the primary VC —

and vice-versa in the case of operating system reboots. The other recovery-related operations do not contribute to the service downtime, because they are performed while the primary server is providing the service.

5.7 EXPERIMENTAL RESULTS FOR PROGRESSIVE DOWNLOAD

This section presents the experimental results of the evaluation of server migration and reboot as recovery strategies for performance failures in video-streaming services. It starts by presenting the server state rescue time and service reestablishment time for server migration, broken down into the migration phases depicted in Figure 32. Then, it presents the impact of server migration on the service quality provided to end-users. Finally, it presents the server warm-up time, the total recovery time and the downtime generated by reboot techniques.

During the experimental tests, the migration of virtual containers is performed 100 times in both directions — from the primary host to the secondary host and backwards. Reboot techniques are executed 100 times.

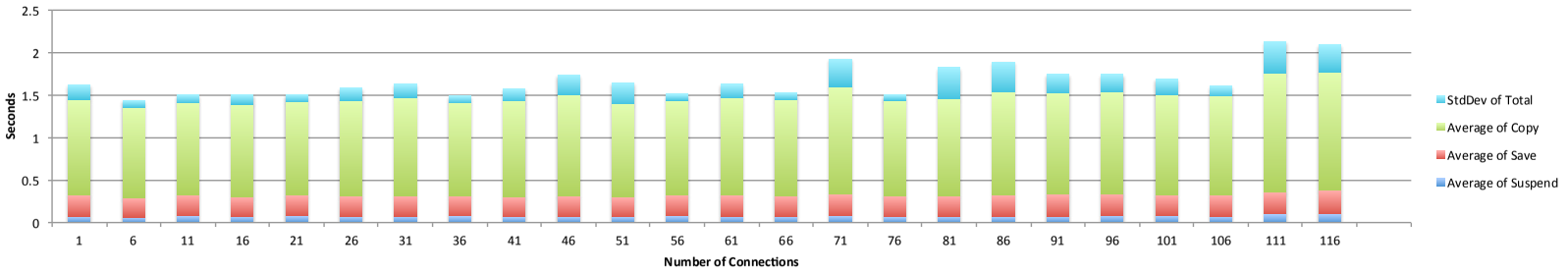
5.7.1 *Analysis of Server Migration Times*

Figure 38a and Figure 38b show the average and standard deviation of the server state rescue time for the popular benchmark and unpopular benchmark, respectively. The server state rescue time is presented for a progressive number of connections, up to the maximum server capacity observed for the respective workload type. The values are broken down into the average time to suspend the VC, save the VC checkpoint to the disk (dump) and copy the checkpoint to the secondary host.

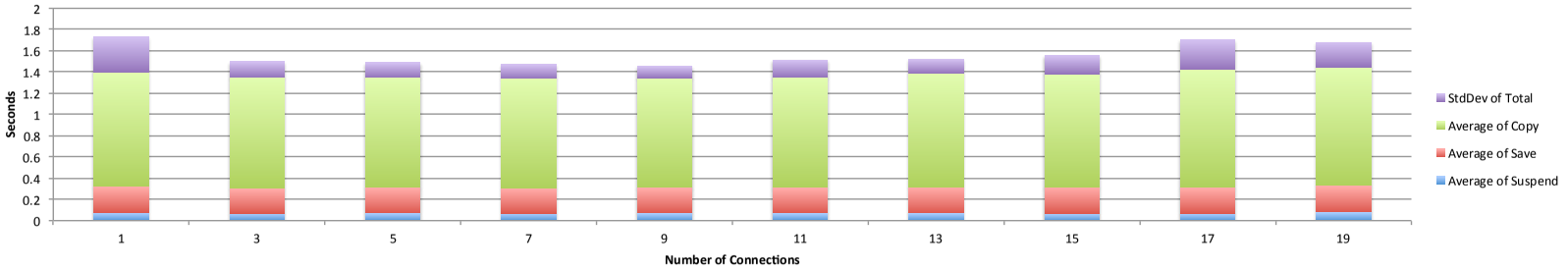
It is noticeable that the virtual container's checkpoint can be rescued in less than 1.5 seconds, in average, for most server load levels and both benchmarks, increasing slightly when the server approximates its maximum capacity. It is also observable that a large portion of the server state rescue time (more than 1 second), during server migration, is spent copying data to the destination host.

The maximum number of connections afforded by the server using the unpopular benchmark (19 connections) is significantly smaller than in the popular benchmark counterpart (116 connections). This observation is expected because the unpopular benchmark does not benefit from temporal locality, since all requests are issued for exclusive video content. That means that the content is fetched from the disk without benefit from caching, creating a disk bottleneck.

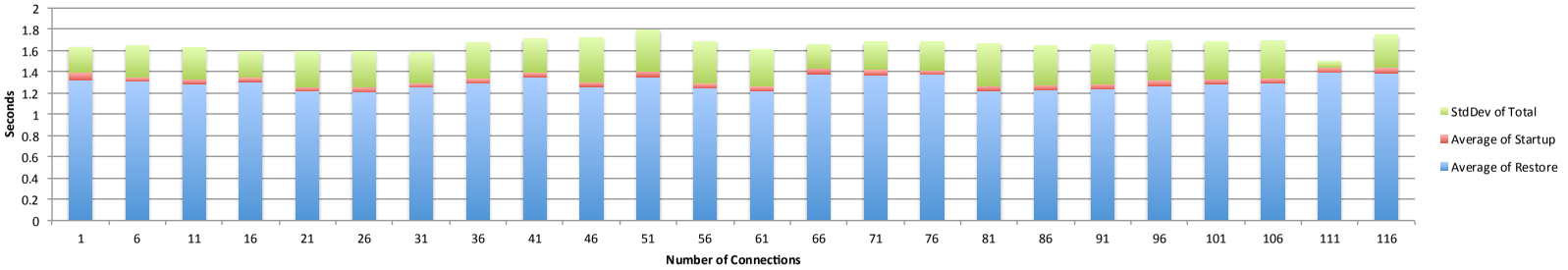
The time required to reestablish the service in the destination host is presented in Figure 38c and Figure 38d. The service reestablishment time is broken down into the checkpoint restore time (checkpoint undump time) and the VC startup time (VC resume time). Both benchmarks have service reestablishment times below 1.4 seconds,



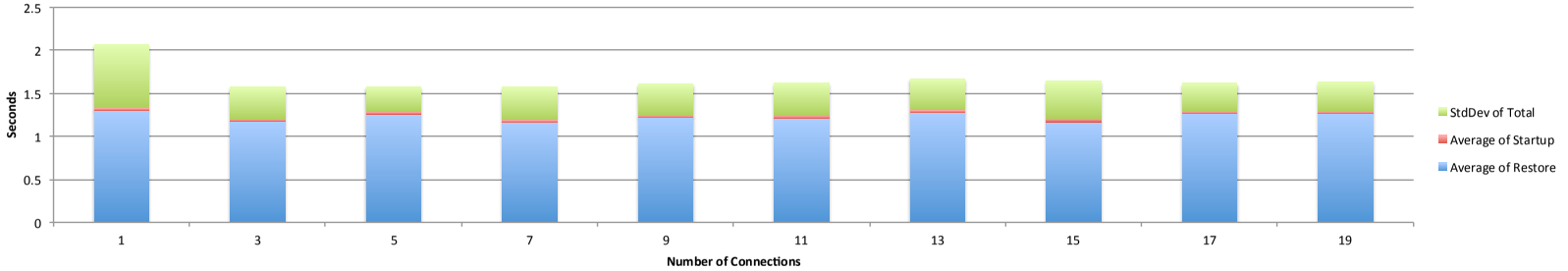
(a) Breakdown of server state rescue times into phases in the popular benchmark



(b) Breakdown of server state rescue times into phases in the unpopular benchmark



(c) Breakdown of service reestablishment times into phases in the popular benchmark



(d) Breakdown of service reestablishment times into phases in the unpopular benchmark

Figure 38: Breakdown of server state rescue times and service reestablishment times during server migration

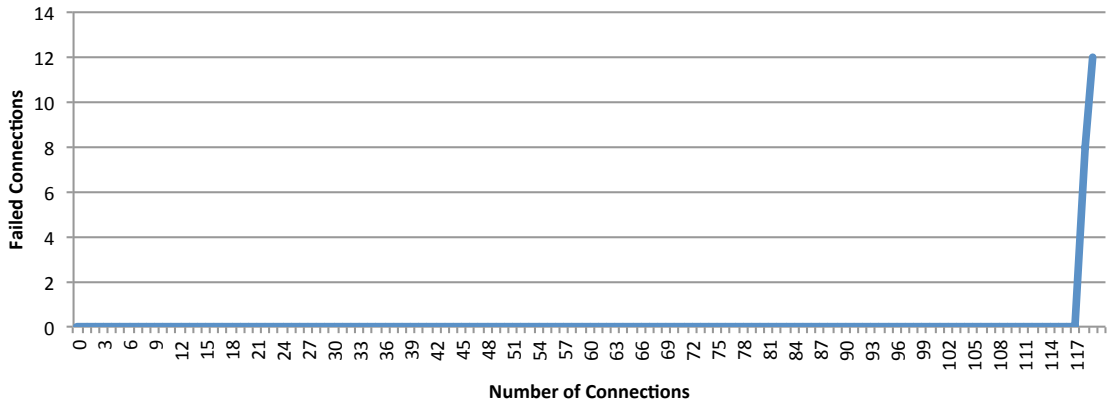


Figure 39: Maximum number of degraded connections of all runs, during the server migration, using the popular benchmark.

in average. Almost all of that time is spent restoring the VC state from the checkpoint file.

The total standard deviation is represented on top of each stacked bar in all graphs of Figure 38, for both server state rescue times and service reestablishment times. It is noticeable that the standard deviation is relatively small and thus, there is low variability around the average values presented.

5.7.2 *Analysis of the Impact of Server Migration on Service Quality*

As described before, the impact of migration of virtual containers on the service quality provided for ongoing video-streaming request-responses is evaluated in terms of the number of degraded connections and the standardized average of bytes written per session.

Figure 39 and Figure 40 show the maximum number of degraded connections (presented as failed connections) of all runs, using the popular benchmark and unpopular benchmark, respectively. Failed connections induced by the popular benchmark are only visible when the server approximates its maximum capacity. By contrast, the number of failed connections induced by the unpopular benchmark increases significantly after 11 connections (approximately half of the server capacity).

The global service levels provided by the server are represented by the standardized average of bytes written per session in the Figure 41 and Figure 42, for the popular benchmark and unpopular benchmark, respectively. Values higher than 1 represent average transmission bitrates higher than the respective encoding bitrates.

The results of the standardized average of bytes written per session for the popular benchmark are consistent with the failures exposed by the number of degraded connections. The exception is the migration of the server when it is near full capacity.

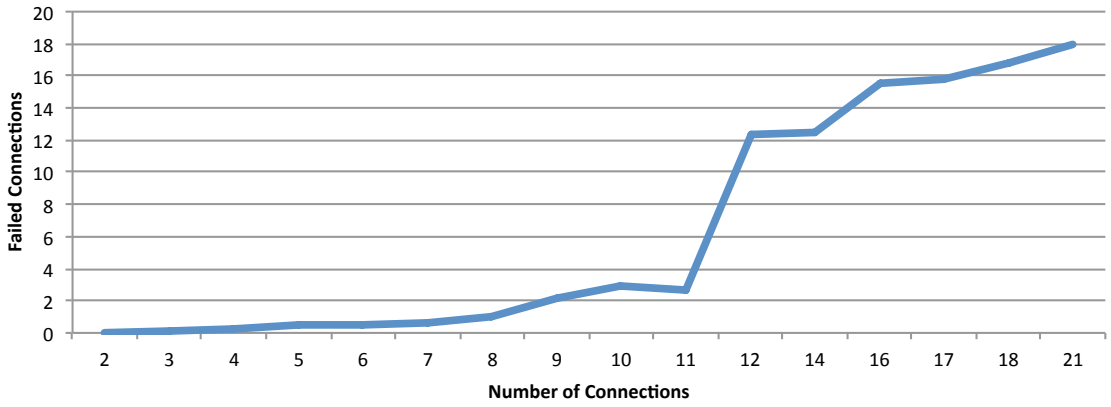


Figure 40: Maximum number of failed connections of all runs, during the server migration, using the unpopular benchmark.

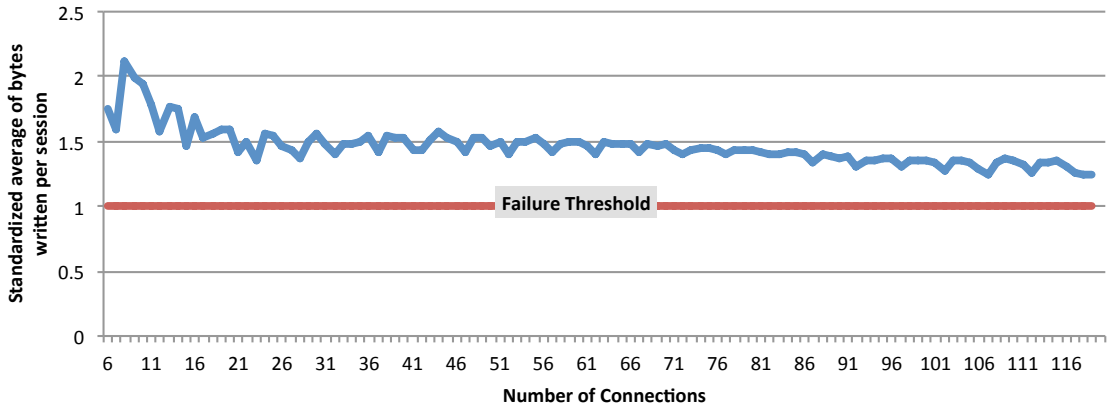


Figure 41: Standardized average of bytes written per session after recovery, using the popular benchmark. The presented values are normalized to the encoding bitrate.

In that scenario, the number of degraded connections can expose some failures unreflected by the standardized average of bytes written per session metric. This result is likely to occur, since the number of degraded connections reveals the maximum failures of all runs. Thus, it is likely that it includes sporadically connections that have had no time to buffer sufficient data to maintain playback continuity.

The results of the unpopular benchmark are also consistent with those obtained from the analysis of individual connections: above 11 connections, the aggregate bitrate transmitted by the server is insufficient to guarantee continuity of playback by players.

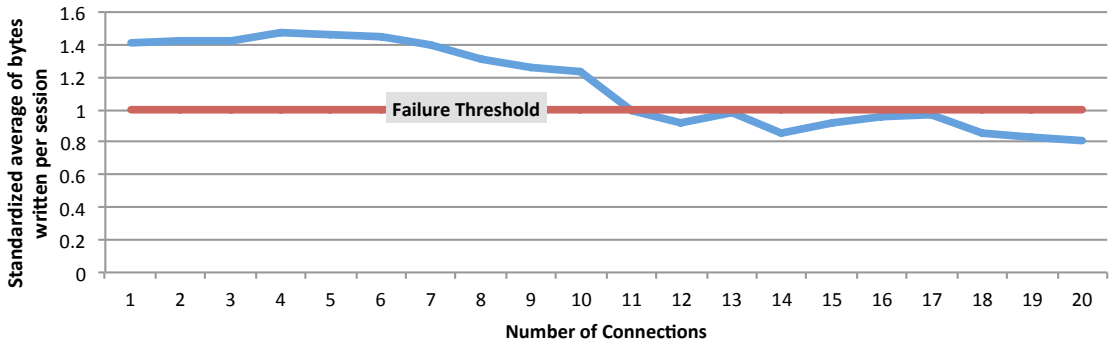


Figure 42: Standardized average of bytes written per session after recovery, using the unpopular benchmark. The presented values are normalized to the encoding bitrate.

5.7.3 *Discussion of Results*

Experimental results presented in this section uncovered several important insights about the use of server migration for recovery of video-streaming servers. We summarize the experimental results as follows:

1. The virtual container state can be rescued in approximately 1.5 seconds, in average;
2. The average service downtime visible by players trying to establish new connections during server migration is below 4 seconds;
3. Transference of checkpoints between hosts is responsible for most of the server state rescue times;
4. Service quality degradation is expected for ongoing video-streaming request-responses during server migration, but only for extreme disk-intensive workloads and server loads above approximately half of the server capacity.

Experimental results show that container-based virtualization allows migration of video-streaming servers with small service downtimes and without service quality interference on ongoing request-responses. The exception is the disk-intensive workload incorporated in the unpopular benchmark. This workload induces service quality degradation when the server surpasses roughly half of its nominal capacity. This phenomenon is mainly explained by the warm-up period that avoids the server to attain its maximum throughput during an initial period after server migration.

Despite the small server state rescue times observed, these values can be reduced even more using faster networks connecting hosts, as the transference of VC checkpoints between hosts dominates the server state rescue time. Additionally, server

warm-up approaches can be necessary to avoid service failures in disk-intensive workloads. We evaluate a server warm-up approach in Section 5.8.

Experimental results taken for the recovery process respect the scenario where the active host is in a healthy state. However, during periods of performance degradation, the time necessary to rescue the server can be larger, depending on the resources available to support the server migration process. Performance anomalies, in particular, impact the server performance in unpredictable ways, avoiding determining, in advance, the time and system resources available for rescuing the checkpoint to another host. However, by stopping the VC during the migration period, the host has more resources available, increasing the likelihood of successfully rescuing the checkpoint to the fallback host. Even though, the unpredictability of the host condition disallows providing recovery guarantees when applying server migration to overcome performance anomalies.

Server migration techniques can be combined with reboot techniques to provide recovery guarantees. Therefore, in case the server migration fails to complete, the server is rebooted to circumvent failures. However, when the reboot action is required, the service is harmed in two ways. Firstly, all ongoing streaming connections are destroyed. Secondly, the rebooted server instance has no time to warm-up, limiting its capacity during a certain period of time. Reboot techniques are evaluated experimentally in Section 5.8.

5.8 EXPERIMENTAL RESULTS FOR ABR STREAMING

This section reveals the experimental results of the evaluation of reboot techniques in ABR streaming services.

5.8.1 *Comparison between Reboot Techniques*

We start by evaluating the techniques presented in Section 5.4 to reboot the server application: (1) *restart the video-streaming application process*; (2) *restart the virtual container*; and (3) *start a fresh replica of the virtual container*. We execute the reboot process 100 times to have statistical significance.

Table 4 presents values of the 5th, 50th and 95th percentiles of the execution delays, for each reboot technique responsible for service downtimes. The technique that starts a fresh replica of the virtual container presents the smallest downtime of all the reboot techniques, reaching 1.4 seconds in the 95th percentile (the median is 0.9 seconds). The technique that restarts the video-streaming application process presents the second-best performance, with 3.7 seconds in the 95th percentile. Finally, the worst downtime is achieved by the technique that restarts the virtual container, reaching 12.3 seconds in the 95th percentile.

	Percentiles in Seconds		
Technique	**5th**	**50th**	**95th**
Restart the video-streaming application process	2.4	3.1	3.7
Restart the *virtual container*	10.2	11.5	12.3
Start a fresh replica of the *virtual container*	0.8	0.9	1.4

Table 4: Downtime generated by each reboot technique.

5.8.2 *Server Warm-up Time*

Figure 43 and Figure 44 show the p-values of the Kruskal-Wallis test, using groups of 20 samples. Assuming a significance level of 95%, the null hypothesis is rejected for p-values lower than 0.05. Accordingly, the load of the secondary server is increased when the p-value increases above 0.05. The presentation of p-values is segmented by the proportion of the number of requests transferred to the secondary server relative to the primary server's capacity (represented by N).

Figure 43 shows that the server warm-up time is approximately 178 seconds for a operating system reboot. This value contrasts with the 70 seconds of server warm-up time required for a VC reboot shown in Figure 44. The smaller warm-up times observed for VC reboots are expected, since the VC state are renewed in the same host, preserving the operating system kernel structures and caches.

We observed that all requests handled during the operating system and VC reboot activities have positive gap values. That means that all requests were processed without failures.

5.8.3 *Recovery Time and Downtime*

Figure 45 presents the time required to execute each recovery step. It is required 72 seconds to recover the VC and, if the failure condition persists, 253 seconds to continue the service into the secondary host after reboot the VC. The entire reboot lifecycle is completed after 434 seconds, plus the time required to perform a operating system reboot in the primary host. The service downtime, represented by the time required by the secondary VC to suspend and take the IP of the primary VC, is less than 2 seconds (rounded to 2 seconds).

5.8.4 *Discussion of Results*

Reboot proved to be an efficient alternative repair approach to server migration for ABR streaming. This recovery strategy can prejudice considerably the service quality in Progressive Download services, since it can interrupt long ongoing request-

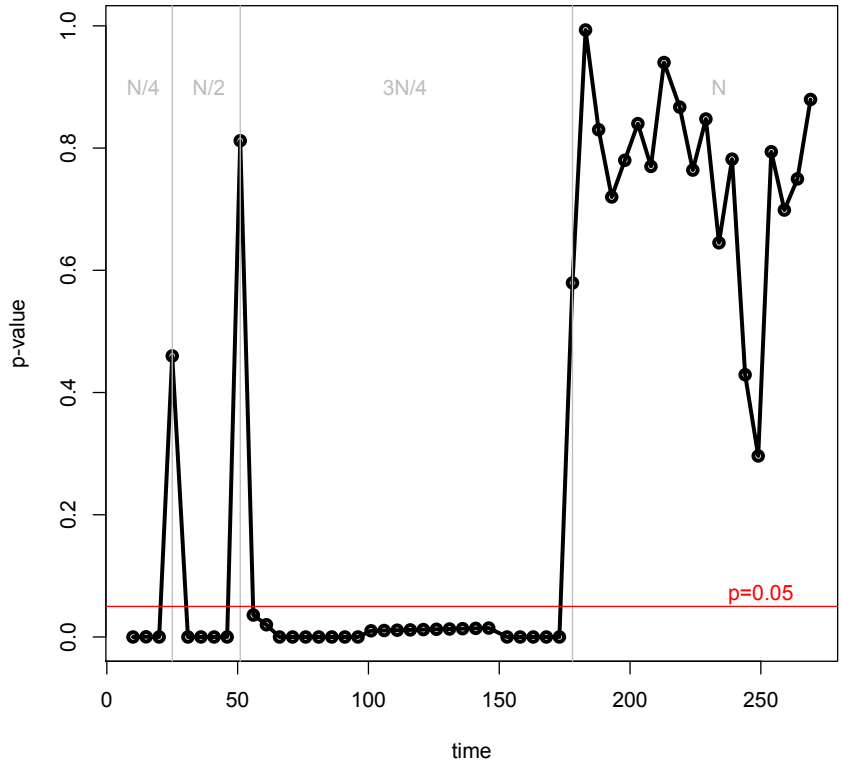


Figure 43: p-values of the Kruskal-Wallis test for the operating system reboot. N represents the primary server's capacity.

responses. However, video segments in ABR streaming can be downloaded again by players in case of failure. This is possible because players are in control of the service quality provided and because the small size of video segments permits the download of each segment more than once before playback, in case of failures.

We observed that the reboot technique that starts a fresh replica of the virtual container revealed to be approximately 3 times more efficient than the restart of the video-streaming application and approximately 10 times more efficient than the restart of the virtual container. This technique provides service reestablishment times lower than 1.4 seconds, in 95% of cases. Additionally, it can be used to reboot the server not only in the same host, but also in the secondary host, seeing that the rebooted replica has been stored in the local disk of the host where it will be started.

Experimental results have shown that our reboot-based approach for recovery of performance anomalies in video servers can be executed with server downtimes smaller than 2 seconds — the time required by the secondary VC to suspend and take the IP

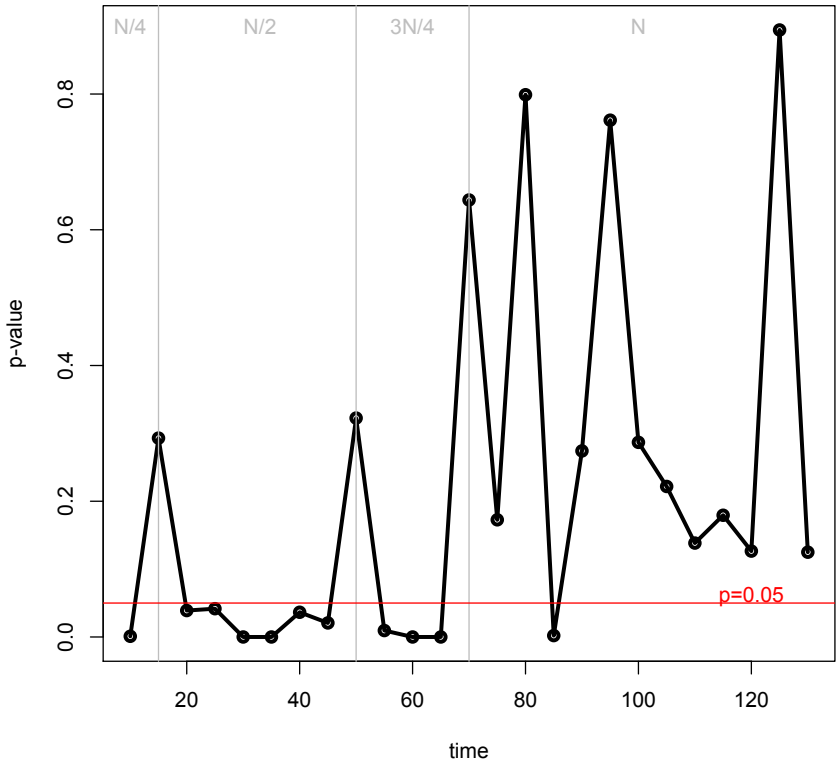


Figure 44: p-values of the Kruskal-Wallis test for the VC reboot. N represents the primary server's capacity.

of the primary VC. Downtimes of this order are small, considering that, in the worst-case scenario, the request has to be downloaded in less than the playback time span of the video segment (10 seconds in our configuration) to avoid playback interruptions. Therefore, the server downtime could shorten the maximum download time of the video segment by less than 2 seconds.

Our recovery approach assumes that performance anomalies will not affect the ability of the server to handle requests, before and during the recovery process. That means that during 72 seconds for VC recovery and 253 seconds for operating system recovery, in the worst-case scenario — when the server is running at full capacity — the primary server should be able to handle the requests not redirected to the secondary server. This is an important assumption because, most of the time spent by the recovery process, is spent warming-up the server. Otherwise, replacing the primary VC by the secondary VC without warming-up the server, when the primary server is experiencing severe failures, can impact less the service quality than maintaining the

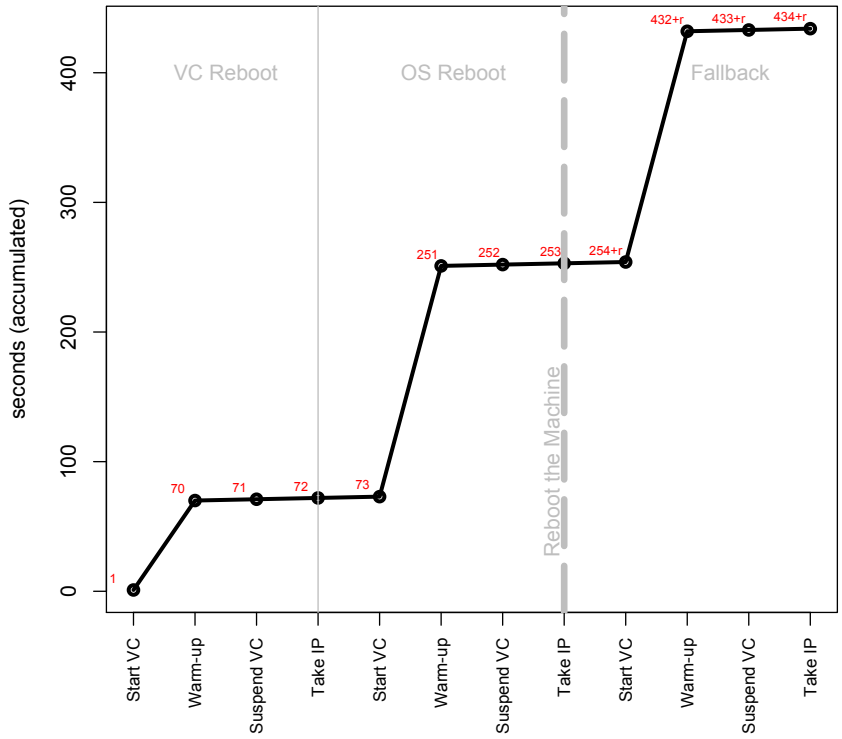


Figure 45: Cumulative time required to recover the server by executing a VC reboot and, if necessary, an operating system reboot.

primary server active during the warm-up period. On the other hand, by redirecting part of the primary server's load to the secondary server during the warm-up phase, the service quality provided by the primary server could return to normalcy — e.g., the utilization of an exhausted resource can decrease below its limit. For the aforementioned reasons, the server warm-up process should be tried before replacing the primary VC by the secondary VC. Then, if failures still occur during the warm-up phase, the secondary VC replaces the primary VC immediately.

5.9 CHAPTER SUMMARY

This chapter presented and evaluated two recovery approaches for performance anomalies in video-streaming services: server migration and reboot. The implementation of these approaches is integrated in the SHStream infrastructure (HTTP Streaming infrastructure) presented in Section 4. SHStream adopts container-based virtualization for performance isolation between the self-healing functionality and the video-streaming

server, within the Autonomic Element. This virtualization approach is exploited for recovery, in order to attain: (1) efficient checkpointing and resumption of virtual containers during server migration between hosts; (2) efficient server reboots; and (3) reduction of the server warmup time.

Experimental evaluation of server migration in Progressive Download services, using workloads with popular videos (high temporal locality), showed short service downtimes without impacting the service quality provided to end-users. On the other hand, service quality degradation was observed during server migration for workloads with unpopular videos (disk-intensive workloads with low temporal locality), when the server load exceeds approximately half of its capacity. That phenomenon is explained not only by the service downtime, but also by the lack of appropriate warm-up after server migration. Experimental analysis also showed that the transference of checkpoint data during server migration is the main contributor of service downtime. Thus, the impact of server migration on the service quality can be reduced using faster networks.

Server migration has shown a promising technique for recovering performance failures in Progressive Download streaming services. Yet, only the failures originating outside the virtual container where the server application is running are recoverable by this technique. Also, the primary host participates in the process of transferring the checkpoint data to the secondary host. Consequently, since performance anomalies generate nondeterministic service behaviors, we are unable to provide completion guarantees for the transference of the checkpoint before the occurrence of hard failures in the primary host.

Reboot techniques are more adequate than server migration for ABR streaming services, since they have low complexity and can be applied to these services without service quality degradation. These techniques are also required for Progressive Download services, when the internal state of virtual containers is compromised or when the recovery process fails rescuing the server checkpoint to the secondary server. In these scenarios, the reboot is disruptive for client-server connections, but can reestablish the service with short service downtimes.

Experimental analysis has shown that by applying reboot techniques without warming-up the server is often insufficient to avoid service quality failures. Therefore, we propose an approach that uses the variance of request-response times to delimit the server warm-up period during the recovery period. The experimental results demonstrated the efficacy of our server warm-up approach, whenever the failure conditions allow the faulty server instance to handle part of the load during the warm-up period.

Failure prediction is addressed in the next chapter. This activity cooperates with the repair activity for enabling proactive recovery of video-streaming systems. Proactive recovery provides guarantees of successfully rescuing server checkpoints during server migration, by proportionating a time window of opportunity to rescue the server's checkpoint to an alternative host even before the occurrence of soft failures. Plus, the

classification outcome provided by the failure diagnosis activity presented in Chapter 7 will decide which repair action to execute.

6 FAILURE PREDICTION

Repair techniques that deal with performance failures in video-streaming services can be applied to reactive recovery and proactive recovery scenarios. In reactive recovery scenarios, the repair techniques are triggered after failure detection, which presents two main problems: (1) soft failures are not anticipated, resulting in an exponential decay in the quality of experience [Fiedler et al. 2010]; and (2) during server migration, the anticipation time window[1] can be insufficient to rescue the server checkpoint.

Proactive recovery is an attractive alternative to reactive recovery. It aims to restore the service after manifestation of the error, but before end-users start noticing failures. Failure prediction supports proactive recovery by signaling service failures in advance. Thus, it enables the execution of repair actions before errors start impacting the service quality. Also, it provides the enlargement of the time window of opportunity required to rescue server checkpoints during server migration and thus, yields higher guarantees of service continuity.

Failure prediction is backed by models that allow detection of abnormal system conditions preceding failures. The accuracy of these models is challenged by the representation of failure patterns that allow the disambiguation between effective system error conditions preceding failures from other temporary conditions unrelated to system failures, such as, server and network performance fluctuations [Dean and Barroso 2013].

This chapter presents the methodologies and algorithms used by our approach for prediction of performance failures in video-streaming services. It addresses several issues related to the creation, maintenance and evaluation of prediction models for Pure Streaming and HTTP Streaming technologies, using online learning and offline learning algorithms. The failure prediction approach is evaluated experimentally for performance failures caused by performance anomalies and workload-related failures.

6.1 FAILURE PREDICTION APPROACH

Failure prediction assumes the existence of patterns of alarm, hinting performance failures occurring in the future. Thus, failure prediction is defined as a *pre-failure pattern* detection problem.

Pre-failure patterns become visible after fault activation (Figure 46), which leads to errors that force the server to expose an abnormal behavior. We hypothesize that these errors are capturable by system, application and network metrics, interpreted as per-

1 Time gap between the moment the soft failure is detected until the occurrence of a hard failure.

formance signatures named as pre-failure patterns. Failure prediction aims to capture pre-failure patterns before errors become exposed to end-users as service failures.

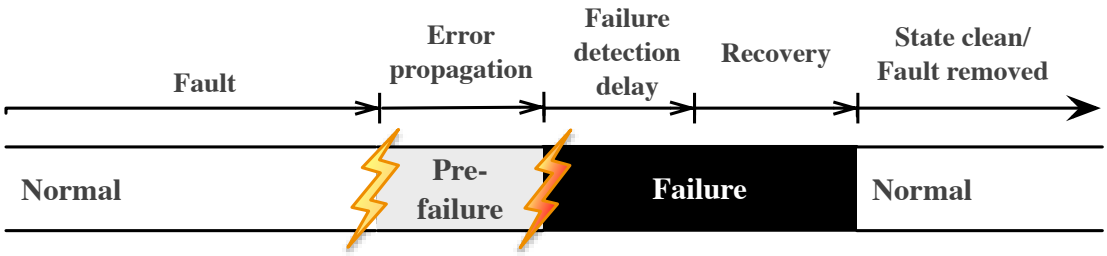


Figure 46: Illustration of the positioning of pre-failure patterns on the failure recovery cycle.

Failure prediction models are two-class classifiers that discriminate normalcy patterns from pre-failure patterns. These classifiers, after being learned using different types of machine learning algorithms, are applied to online classification of new log data. Later, the log data and respective classification are stored along with the observed service state (failure or non-failure) observed by the end of the look-ahead time period of prediction. The service state attests the failure prediction correctness, necessary for evaluating the models' classification performance and also for training failure prediction models.

Transferability of models between systems is limited by their dependence on hardware and software configurations [Powers et al. 2005]. Thus, every host has its own set of models for prediction, which can be replaced anytime by new models in face of major software updates and hardware upgrades.

6.1.1 *Formalization of the Approach*

Our failure prediction approach is formalized as follows. Being $F = \{normal, pre\text{-}failure, failure\}$ the list of service states and M the space of vectors of values corresponding to application, system, service and network metrics and parameters, each model is trained incrementally to discriminate the subspace $S1$ of M associated to the service state $f = \{normal\}$ (normalcy patterns) from the subspace $S2$ of M associated to the service state $f = \{pre\text{-}failure\}$ (pre-failure patterns), being $f \in F$. The latter state is observed between periods where $f = \{normal\}$ and periods where $f = \{failure\}$. In other words, pre-failure patterns are vectors of the multidimensional space M that are observed between fault activation and the exposition of the corresponding service failures to end-users.

6.1.2 *Failure Prediction Hypotheses*

Failure prediction is supported by several hypotheses set out for pre-failure patterns. Pre-failure patterns should:

1. Be exhibited in an unambiguous way before the failure — pre-failure patterns should precede service failures and should be exclusive to faulty periods;
2. Be represented by the features (metrics and parameters) covered by monitoring — the availability of features absorbing these patterns and the selection of representative features compel the performance of failure prediction models;
3. Be representable by models able to classify them with an acceptable margin of error — selection of algorithms for learning models is a determinant activity to attain high prediction performance;
4. Provide an acceptable failure anticipation time — the look-ahead time[2] provided by prediction should suffice to apply countermeasures to restore the service to normal levels.

These hypotheses pose the most important challenges of our failure prediction approach. The validity of these hypotheses is a necessary condition to obtain accurate failure prediction models using our approach.

6.2 DERIVED METRICS

The failure prediction activity uses metrics data gathered by the monitoring activity. These data are represented as periodic snapshots, which are unable to expose the tendency of metrics values along the time. To circumvent this limitation, we calculate second-order metrics, designated as derived metrics, to represent the temporal tendency of metric values.

As we are doing short-term prediction of failures (in the order of seconds), the temporal tendency of metrics values will likely help improving the model performance. As an example, a CPU utilization close to 100% may indicate a possible failure in the near future if the utilization of this resource is increasing but not if the tendency is for decreasing utilization.

By adding one derived metric per each metric selected for the model, it is possible to determine the tendency of metrics values. Each derived metric is calculated as the angle between the first and the last values of a given time window, as shown in Figure 47. The angle $\theta \in [-90, 90]$ degrees measures the slope of the tangent line calculated for each metric. Intuitively, the strength of each metric tendency is proportional to $|\theta|$.

2 Time between the observation of the pre-failure pattern and the failure occurrence.

Positive values of θ represent a tendency to increase the metric value, whereas negative values of θ represent a tendency to decrease its value.

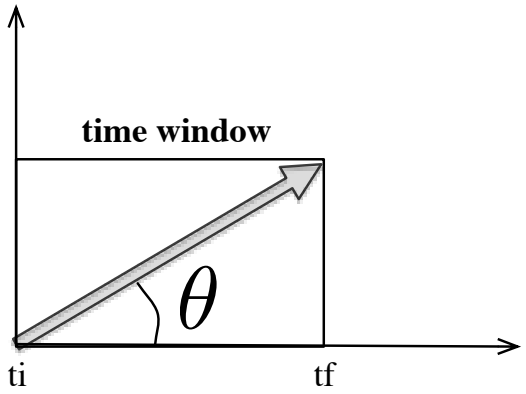


Figure 47: Derived metric calculated as the angle formed by the first and the last value of a given time window.

6.3 EVALUATION OF PREDICTION MODELS

The success of any failure prediction approach depends on the accuracy of its prediction models, which depends on:

- **Coverage of metrics and parameters** — all metrics and parameters capable of identifying pre-failure patterns should be included in the model training dataset;
- **Data gathering frequency** — this parameter limits the look-ahead time of prediction. Smaller data gathering intervals allow smaller delays between the reflection of errors in metrics and their analysis in the failure prediction process. In other words, the sooner the metrics measurements representative of pre-failure patterns are gathered, the larger the look-ahead time of failure prediction will be;
- **Adjustment of models to learning data** — models should fit the training data and thus, correctly classify the pre-failure patterns included in the learning dataset with small errors;
- **Model generalization** — new log data resulting from measurements taken from production services (i.e., data not included in the learning dataset) should be accurately classified.

Accurate prediction models should not only be able to correctly predict events registered in the model training dataset, but also to correctly predict events in new log measurements (unseen data). However, the models that best adjust the learning dataset can

be those that least adjust to future events requiring classification. This phenomenon occurs because the reduction of prediction errors over training datasets can be done at the cost of increased model complexity, which are in the origin of the *overfitting* phenomena [Hawkins 2004]. An overfitted model can perfectly model events registered in the learning dataset but lacks prediction generalization to other events unseen in the learning dataset. For that reason, the models' accuracy has to be evaluated using both training data (historical data) and unseen data.

6.3.1 *Metrics for Measuring the Performance of Prediction Models*

Failure prediction performance are commonly measured by four metrics:

- *True Positives (TP)* — the number of pre-failure patterns correctly classified;
- *False Positives (FP)* — the number of normalcy patterns classified as pre-failure patterns;
- *True Negatives (TN)* — the number of normalcy patterns correctly classified;
- *False Negatives (FN)* — the number of pre-failure patterns classified as normalcy patterns.

These metrics provide the elements for reasoning about the model prediction performance. However, it is complex to interpret the models' performance and compare several models using each of these metrics individually. To simplify the interpretation of results, several metrics imported from the *information retrieval* domain [Manning et al. 2008] are commonly used to evaluate models: *recall*, *precision* and the *f-measure* metrics.

Recall (10) is a measure of completeness or quantity. This metric is also known as the *sensitivity* or *coverage* of the model. It represents the fraction of failure events captured by the classifier.

$$\mathtt{Recall} = \frac{TP}{TP + FN} \tag{10}$$

Precision (11), also called *positive predictive value*, is a measure of exactness or quality that captures the true positive rate of failure events.

$$\mathtt{Precision} = \frac{TP}{TP + FP} \tag{11}$$

The models' classification performance can be evaluated and compared using the recall and precision metrics. However, using a single metric to represent the models'

performance simplifies the comparison of performance between models. F-measure is a metric that combines recall and precision, by calculating their weighted harmonic mean (12).

$$F-measure = \frac{2 \cdot Precision \cdot Recall}{(Precision + Recall)} \tag{12}$$

Notwithstanding the simplicity of the f-measure metric for evaluation of models — since it exposes a single value representative of the classification performance — the analysis of recall and precision metrics individually would be required. The reason is that false positives (captured by the precision metric) can have a larger negative impact on the service provided to end-users than true positives (captured by the recall metric). As an example, models with a large number of false positives can have prohibited costs when the repair actions triggered to recover the system are expensive (e.g., repair generates large service downtimes). In such cases, f-measure would be less representative of the model performance, as high precision values would be preferable to high recall values.

Both precision and recall metrics are robust to unbalanced datasets [Chawla 2005], which makes them good metrics for evaluation of failure prediction performance. Unbalanced datasets hold a disproportionate number of instances respecting each classification outcome. In failure prediction problems, it is common to have more log instances gathered during periods of normalcy than log instances gathered during faulty periods. This is because the service is running most of the time without failures.

6.3.2 *Model Evaluation Process*

As explained before, prediction models should be evaluated during the training phase with historical data, and later with new log data. Models with poor classification performance predicting historical events recorded in the training dataset should be discarded before being used in production services. Further, failure prediction models that start showing unacceptable prediction performance with new log data should be removed from the prediction process or replaced by others. The model evaluation scheme depends on the learning algorithm type. Basically, learning algorithms can be decomposed into: *offline learning* (also known as *batch learning*) and *online learning* algorithms.

Models created with offline learning algorithms remain unchanged after their creation. *Cross-validation* techniques [Kohavi 1995] are commonly employed to evaluate these models with historical data, after being created. Afterwards, *per-instance validation* evaluates continuously the models' prediction performance with new gathered data, after each prediction, as soon as the information about the prediction correctness becomes available.

Compared with offline learning algorithms, online learning algorithms assume that the historical datasets are unavailable for learning or that they have insufficient historical data to create accurate models. Hence, models are trained progressively with data gathered iteratively from logs. Due to the incremental nature of this learning approach, only per-instance validation is appropriate for the evaluation of prediction models.

6.3.2.1 *Cross-validation*

Cross-validation techniques determine how predictions performed by a given model will generalize between different subsets of the learning dataset. This form of validation measures *bias* and *variance* errors in the learning dataset. Bias errors respect modeling errors on each subset of the learning dataset. On the other hand, variance errors are taken for model predictions using different learning subsets. In other words, variance errors measure the prediction error for a given dataset using different realizations of the model — each one excluding a learning subset of the dataset. To determine the variance, the entire model building process is repeated several times, one for each subset.

The bulls-eye diagram in Figure 48 illustrate the bias and variance concepts. The center of the target is a model that perfectly predicts the correct values. The entire model building process is repeated to get a number of separate hits on the target. Each hit means one individual model realization, given the chance variability in the training data. Thus, different realizations will result in a scatter of hits on the target. Intuitively, low bias errors are achieved when the hits are close to the center of the target and low variance errors are achieved when all hits are close together.

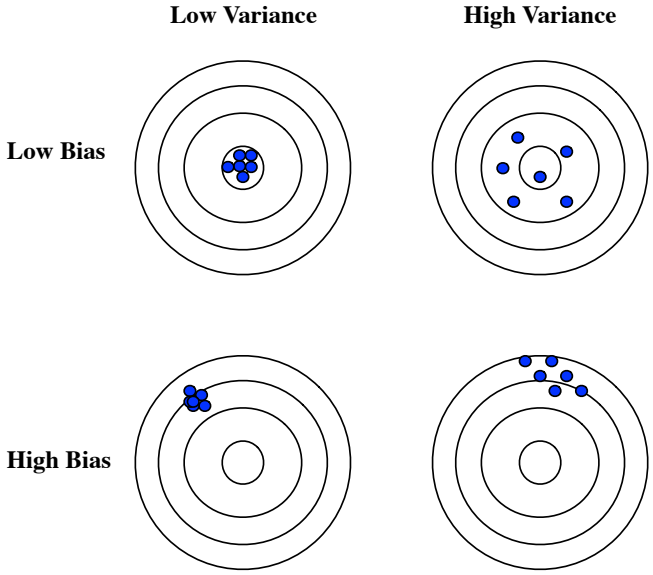


Figure 48: Illustration of bias and variance.

In *K-fold cross-validation*, the logged data instances are randomly segmented into k independent equal-sized folds, as illustrated in Figure 49. The evaluation process repeats k times, one for each model realization and, at each time, a different fold is used for evaluation. The remaining folds are combined and used for training. *Ten-fold validation* (i.e., k-fold cross-validation with $k = 10$) is the most common configuration for cross-validation. Performance metrics used for evaluation expose bias from each model realization and variance between model realizations.

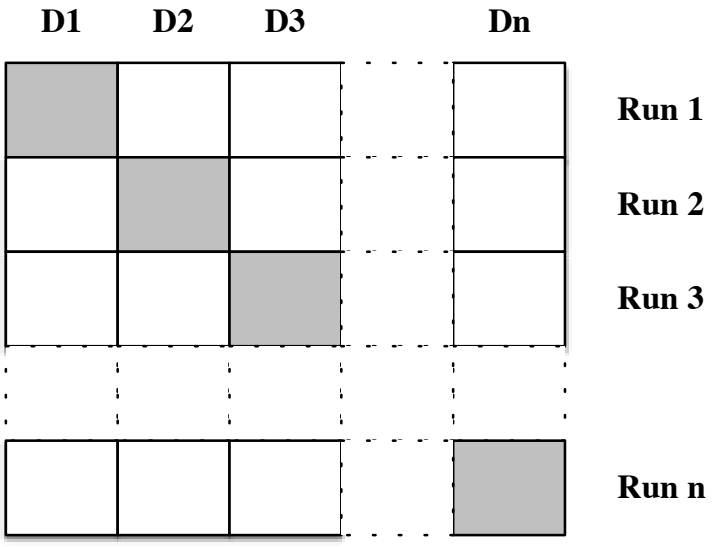


Figure 49: At each run, the data segment colored as gray is used to evaluate accuracy and the others are used to train the classifier.

Bias and variance are two main prediction error terms. Considering $\widehat{f}(x)$ a model estimation of $f(x)$, the expected squared prediction error is determined as in (13).

$$Err(x) = E\left[(Y - \widehat{f}(x))^2\right] \tag{13}$$

The error decomposition into bias and variance components is formulated in (14).

$$\begin{aligned} \mathbf{Err}(\mathbf{x}) &= \left(E[\widehat{f}(x)] - f(x)\right)^2 + E\left[\widehat{f}(x) - E[\widehat{f}(x)]\right]^2 + \sigma_\epsilon^2 \\ \mathbf{Err}(\mathbf{x}) &= Bias^2 + Variance + IrreducibleError \end{aligned} \tag{14}$$

The third term of (14) represents the irreducible error, which cannot fundamentally be reduced by any model, as it represents the noise term. Bias and variance terms can be reduced to 0, given the true model and infinite data to calibrate it. Since real models are imperfect and data is finite, there is a trade-off between minimizing both bias and variance.

6.3.2.2 *Per-instance validation*

Classification models require continuous evaluation after being settled for prediction in production environments. As well, in online learning scenarios, only a limited amount of data can be available for learning initially. Thus, the learning process runs continuously and simultaneously with classification of log instances. Hence, the models' performance metrics should be recalculated after each classification.

6.4 FAILURE PREDICTION METHODOLOGY

The failure prediction activity rolls out into several phases:

1. **Data preparation** — prepare the log data to optimize the classification performance;
2. **Feature selection** — select relevant features for building accurate prediction models. When the learning algorithm performs feature selection intrinsically, this phase is avoided;
3. **Model learning** — train prediction models using log data;
4. **Model evaluation** — this phase starts after model creation (cross-validation) and afterwards, to determine the model classification error iteratively with new log data (per-instance validation);
5. **Classification** — apply prediction models to classification of log instances into normalcy patterns and pre-failure patterns.

Some machine learning algorithms demand the preparation of data and selection of relevant features (first two phases). These activities will be further described.

6.4.1 *Data Preparation*

The monitoring activity is responsible for providing log data for the model training and classification activities. However, these data should be prepared to maximize the models' accuracy.

We consider two data transformations: *rescaling of feature values* and *data discretization*. Rescaling changes the range of each feature to the interval $[0, 1]$, so that each feature value x is rescaled to x' using the formula (15).

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{15}$$

Not all learning algorithms allow both continuous and discrete feature variables. The values of continuous feature variables have to be discretized before being used by some learning algorithms. We use equal-width binning to divide the range of possible values of each feature into N subranges of the same size, so that each range is represented by an integer.

6.4.2 *Feature Selection*

Only a subset of the features (metrics and parameters) included in the learning dataset has discriminatory power for prediction. Some features are irrelevant and add complexity to prediction models when considered by learning algorithms. The role of feature selection is to identify the list of features relevant for prediction models, in the list of metrics provided by the learning dataset. Feature selection can be performed intrinsically by learning algorithms or otherwise, it may require an independent approach.

6.4.2.1 *Curse of Dimensionality Problem*

The *curse of dimensionality* [Witten et al. 2011] is a phenomenon in high dimensional data spaces that prejudices the prediction performance. Dimensionality represents the number of features in the data space. When the dimensionality increases in data spaces, the available data become sparse. Consequently, the amount of data needed to achieve statistical significance in the model training process grows exponentially with the dimensionality. Additionally, high dimensional spaces lead to complex models having large computational and time costs, and low discriminative performance. Therefore, the reduction of dimensionality is desirable to build powerful and efficient models.

By reducing the number of features, the models' complexity is reduced as well. Thus, it is expected a speed up in the model learning process and an increase in the model's generalization capability. Also, the interpretability of models is improved, so that the visual inspection of models becomes possible.

6.4.2.2 *Feature Selection Algorithm*

We adopt the Linear Forward Selection (LFS) algorithm [Gutlein et al. 2009] for feature selection. This algorithm is known to be fast and accurate for scoring subsets of features according to their predictive ability. The LFS algorithm uses a *wrapper* method for selection of metrics during the feature selection process. Wrapper methods use learning algorithms and resultant models — one model for each feature — to score features according to their predictive ability. After ranking features by significance, the resultant list R is used by LFS to include stepwisely each candidate feature $r \in R$, ordered by its significance, into the subset of features $\vec{M}$ chosen for the model. This process

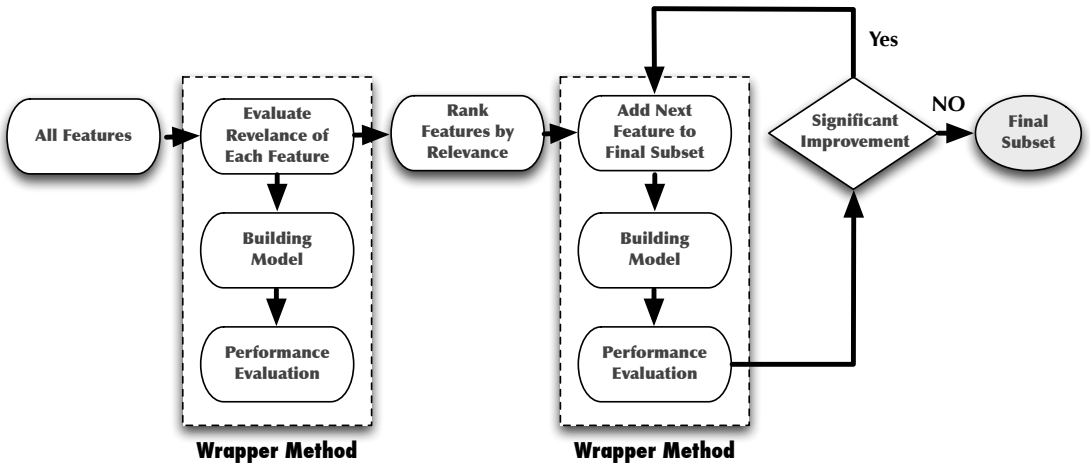


Figure 50: The process followed by the Linear Forward Selection algorithm.

continues until none of remaining features of R reveal significant improvement when added to the model, as described in Figure 50.

Despite the LFS algorithm suits most types of problems, there are other alternative feature selection approaches that can be explored. The Principal Component Analysis (PCA) procedure [Guyon and Elisseeff 2003] and the Mutual Information Criteria [Peng et al. 2005] are two popular alternatives to LFS for feature selection.

6.4.3 *Training, Evaluation and Exploitation of Models*

The activities associated with training, evaluation and exploitation of models are specific to each class of algorithms and will be addressed in the next sections for the Pure Streaming and HTTP Streaming infrastructures.

6.5 FAILURE PREDICTION IN PURE STREAMING

This section presents the research questions, the machine learning algorithms, the model classification strategy and the results of the experimental evaluation of our failure prediction approach for Pure Streaming services.

6.5.1 *Research Questions*

The evaluation of our failure prediction approach in Pure Streaming services is performed to answer the following fundamental questions:

1. Are performance failures preceded by server states exposed in the form of recurrent and unambiguous pre-failure patterns to be captured by prediction models?

2. Which type of classification algorithms shows the best failure prediction performance in video-streaming servers?
3. What is the relationship between the prediction look-ahead time and the classifiers' performance?
4. What is the relationship between the failure severity and the classifiers' performance?
5. Does the inclusion of past data in the learning process increases the classifiers' performance?
6. What is the performance difference between the prediction of workload-related failures and performance anomalies?

The first research question is of the highest importance in our research. When pre-failure patterns are exposed by metrics to be represented by models, the remaining work resides in selecting algorithms and configurations able to create models with the best prediction performance. The remaining questions focus on the breakdown of results into several aspects of analysis.

6.5.2 *Batch Learning Algorithms*

Failure prediction focuses on training, evaluation and exploitation of prediction models, created using *supervised learning* algorithms [Hastie et al. 2001]. These algorithms train models from a known set of input data and known responses to the data, with the purpose of obtaining accurate predictive models that generate reasonable predictions for the response to new data. Predictive models are trained using vectors of feature values labelled with one of the following pattern types: normalcy patterns or pre-failure patterns.

To obtain accurate models for classification of service states, it is mandatory to select appropriate learning algorithms. Our failure prediction infrastructure incorporates several popular algorithms in the machine learning and data mining fields. It is unfeasible to evaluate our approach using all algorithms published in the literature. Instead, we consider reasonable to choose one algorithm representative of each class of algorithms encountered in the literature to ensure diversity of prediction models.

Learning algorithms can be grouped into *decision trees*, *probabilistic models*, *discriminant functions* and *ensembles of models* [Hastie et al. 2009]. To represent each of these groups, we choose the following algorithms:

- C4.5 learning algorithm [Quinlan 1993] to build decision trees;
- Tree Augmented Naïve Bayesian Networks (TANs) [Friedman et al. 1997] to build probabilistic classifiers;

- Support Vector Machines (SVMs) [Hearst et al. 1998], with a *polynomial kernel* to create discriminant functions;
- Bootstrap Aggregating (Bagging) [Breiman 1996a] to build ensembles of models.

We evaluate the algorithms integrated in our failure prediction approach in terms of their prediction performance (using the metrics presented in Section 6.3), look-ahead time provided by prediction and interpretability of models.

6.5.2.1 *Look-ahead time*

The look-ahead time provided by prediction limits the effectiveness of repair actions. The execution time of proactive recovery actions should be smaller than the look-ahead time provided by failure prediction, to avoid the occurrence of service failures. The larger the look-ahead time provided by prediction, the wider the set of repair actions eligible to recover the service before end-users start experiencing failures. However, the experimental results of the application of container-based virtualization techniques to recovery of video-streaming servers (Chapter 5) have shown that look-ahead times in the order of a few seconds is sufficient to recover the service before the occurrence of failures.

6.5.2.2 *Interpretability*

Interpretability is a desirable quality of any failure prediction approach, as it allows human inspection of prediction models. Black box modeling approaches lack interpretability, leading to obscure decisions taken from classification outcomes given by models with structures invisible to human operators.

Interpretable models also help dealing with the *automation irony* [Russell et al. 2003], a phenomenon where the reduction of reliance on human decisions leaves systems vulnerable to wrong decisions made by systems automatically. Interpretability helps turning on transparent the set of rules used by models to predict failures and their relationship with the observed features values. Additionally, fault removal activities carried out during software maintenance can be facilitated by the analysis of the model structure, model classification outcomes and features values observed during faulty periods.

From the list of algorithms considered for creating failure prediction models, only C4.5 decision trees and Tree Augmented Naïve Bayesian Networks produce interpretable models. The structure of these models is formed by a tree — also representable by classification rules — that relates features in a way that permit human inspection.

6.5.2.3 *C4.5 Trees*

The C4.5 algorithm creates decision trees models that are are robust to noise, avoid overfitting (perform feature selection intrinsically) and produce interpretable models. C4.5 is an evolution of ID3, an algorithm to build decision trees using the concept of *information entropy*. During the tree building process, the C4.5 algorithm selects, for each node of the tree, the attribute that most effectively splits the training set of samples (learning instances) into subsets enriched in one class or the other. It uses Hill-Climbing Search [Hastie et al. 2001] based on the Normalized Information Gain (difference in entropy) to search the space of decision trees.

The Hill-Climbing Search is an algorithm that iteratively attempts to find better solutions for a given problem, by incrementally changing a single element of the solution. If the change (a new node split) produces a better solution — determined in the C4.5 by the Normalized Information Gain — an incremental change is made to the new solution. This process is repeated until no further improvements can be found.

The Normalized Information Gain is computed as described in (16), being S the attribute space, E the entropy, $values(A)$ the set of possible values of an attribute A and S_v the subset of S when A has value v.

$$Gain(S, A) = E(S) - \sum_{v \in values(A)} \frac{|S_v|}{|S|} E(S_v) \tag{16}$$

Decision tree models are representable by decision rules, formed through interpretation of each tree path. One tree path represents the set of nodes, from the tree root to the tree leaf, followed by one rule used to make a decision. To exemplify, Figure 51 shows one decision tree that determines when to play and when not to play a game of golf. The decision tree was created by the C4.5 algorithm with 14 samples. Each tree leaf shows the number of samples that incorporates the respective tree path.

6.5.2.4 *Discriminant Functions*

In the group of discriminant functions, we opt by Support Vector Machines (SVMs) with a polynomial kernel, due to their good prediction performance when applied to prediction in several domains [Sapankevych and Sankar 2009][Kim 2003][Thissen et al. 2003]. SVMs construct N-dimensional *hyperplanes* that optimally separate samples into categories.

SVMs select a small number of critical boundary samples called *support vectors* from each class (possible classification outcome) to create a linear discriminant function — optionally extended to a non-linear function to form quadratic, cubic, and higher-order decision boundaries — that separates classes as widely as possible. The maximum margin hyperplane that separates two classes is represented as in (17), where i reference

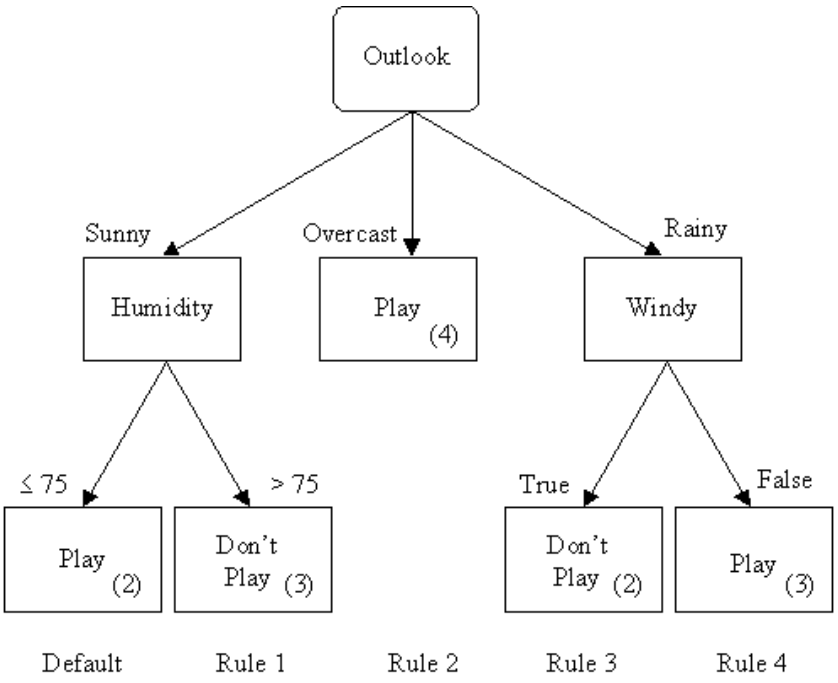


Figure 51: Decision tree model created with the C4.5 algorithm.

each support vector, a the test instance, $a(i)$ the support vectors, and finally b and α_i are parameters that determine the hyperplane that separates two classes.

$$x = b + \sum_i \alpha_i Y_i a(i) \cdot a \tag{17}$$

Figure 52 illustrates an example with three candidate hyperplanes H1, H2 and H3. We can observe that H1 is unable to separate classes, H2 does so but with a small margin and H3 separates them with the maximum margin. Thus, the H3 hyperplane has the largest distance to the nearest training data point of any class. Samples in the margins are called support vectors.

In failure prediction problems, samples are mapped into the model space and are classified to belong to a class based on which side of the hyperplane they fall on. Larger margins between support vectors will give higher confidence on the predictions, as long as they represent lower generalization errors of the classifier.

6.5.2.5 *Probabilistic Classifiers*

Naïve Bayes is a popular probabilistic classifier that is known to be accurate. Naïve Bayes classification is performed over the probability of each class value C — in failure

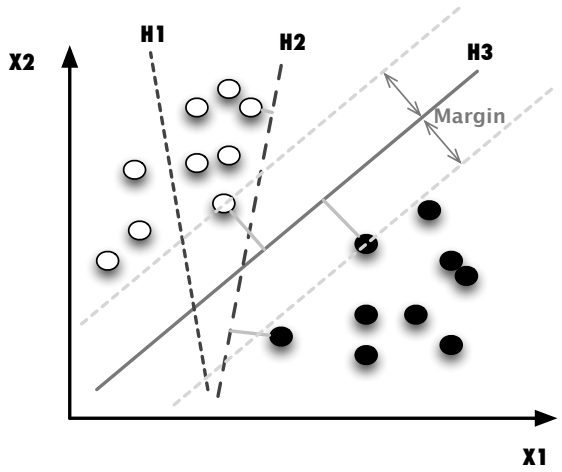


Figure 52: Separation of hyperplanes using SVMs.

prediction, class values would be either normalcy pattern or pre-failure pattern — conditional on several feature variables F_i, as formulated in (18).

$$p(C|F_1, ..., F_n) = \frac{p(C)p(F_1, ..., F_n|C)}{p(F_1, ..., F_n)} \tag{18}$$

Despite the good classification performance of Naïve Bayes, it has a fixed structure, encoding the assumption that features are conditionally independent. The independence assumption is unrealistic in several problems.

Tree Augmented Naïve Bayes (TANs) [Zheng and Webb 2010] improve Naïve Bayes by weakening the feature independence assumption, through the approximation of dependencies among features in a tree structure. The dependence between features is illustrated in the network presented in Figure 53.

TANs comprise a subclass of Bayesian Networks [Pearl 1988] that create probabilistic models which combine knowledge engineering and statistical induction. They are characterized as networks of nodes with probabilistic semantics, where nodes represent features connected by annotated directed edges encoding a joint probability distribution, in such a way that there are no cycles (Directed Acyclic Graph). The Maximum Weighted Spanning Tree [Friedman et al. 1997] that maximizes the likelihood of the training data is used to perform classification.

TANs have been shown to outperform Naïve Bayes and other bayesian approaches (e.g., Generalized Bayesian Networks) in both cost and accuracy for classification in a variety of contexts [Friedman et al. 1997]. Additionally, they have been successfully applied to performance diagnosis and management in three-tier web services [Cohen et al. 2004] and other Internet services [Tan and Gu 2010].

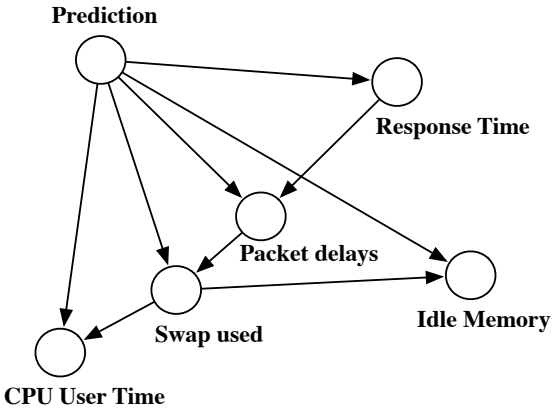


Figure 53: Example of a TAN structure for failure prediction, representing the dependencies between features in a tree structure.

6.5.2.6 *Ensembles*

Ensembles group several models that are generated and combined to solve a particular computational intelligence problem. By using several models in the classification process, it is expected an improvement over the performance provided by a single model and a reduction of the likelihood of an unfortunate selection of a poor one. The prediction performance of a single model can be improved by exploiting diversity of models when making decisions, using outcomes of several models (e.g., through Majority Voting [Ruta and Gabrys 2005]) built using different data or algorithms.

We choose the Bootstrap Aggregating (Bagging) algorithm [Breiman 1996a] as the representative of ensemble learning algorithms. Bagging produces models with high classification performance, even when using learning datasets with noise — within some reasonable levels. This algorithm achieves resiliency to noise by exploiting it to produce more diverse classifiers [Dietterich 2000].

Bagging is a robust meta-learning algorithm that operates with other algorithms to create several classification models. It performs *bootstrapping* (random sampling) on training data for model diversity. Classification is performed by amalgamating the outputs of all models into a single prediction outcome using Majority Voting, as illustrated in Figure 54. By randomly choosing samples for training models, Bagging expects to improve the accuracy of models and reduce variance and overfitting.

Bagging generates m new training sets S_i, by *sampling with replacement* [3] from the standard training set S with size n, uniformly. Sampling with replacement ensures that the samples can be repeated in different training sets. S_i is known as a *bootstrap sample* because, with a large enough n, each S_i is expected to have $\approx 63.2\%$ of the unique samples of S, calculated as the fraction $1 - 1/e$, being the rest duplicates. Each

3 All samples are eligible to incorporate any S_i.

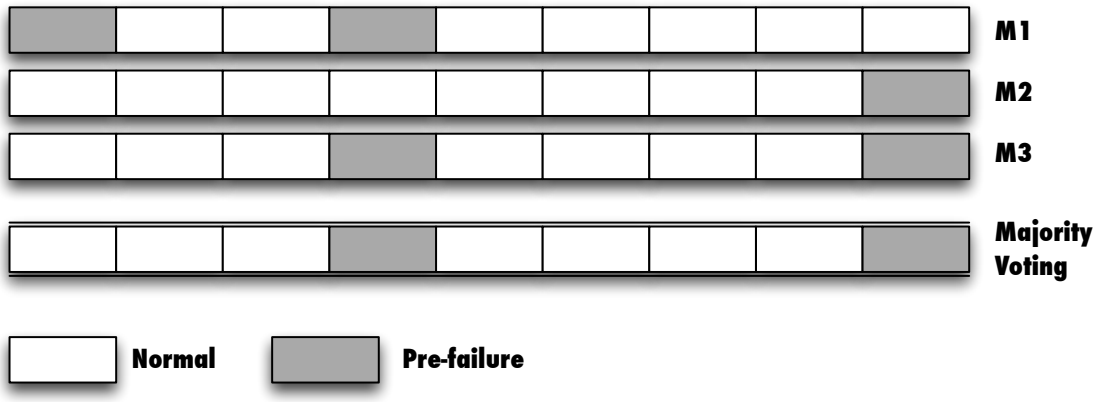


Figure 54: Majority voting in failure prediction.

training set S_i is used to train one model M_i that will be included in the ensemble of models.

6.5.2.7 *Learning Algorithms Implementations*

We use the Weka implementation of the algorithms [Hall et al. 2009] to build failure prediction models. It includes the Sequential Minimal Optimization (SMO) — an algorithm for solving the optimization problem that arises during the training of SVMs — and the J48 open-source implementation of C4.5 decision trees. The Bagging meta-learner is used with ensembles of C4.5 models.

6.5.3 *Implementation of Failure Prediction*

Learning datasets include log instances associated to periods of normalcy (normalcy patterns), periods of pre-failure (pre-failure patterns) and periods of failure (failure patterns). Figure 55 illustrates these three periods. Prediction models are created using log instances gathered during periods of normalcy and periods of pre-failure. During periods of pre-failure, there are two types of log instances: *client-workload overloading* and *performance anomalies*.

Several model configurations can be explored for failure prediction. We start by exploring the *model-pairing* configuration (Figure 56) using distinct models for binary classification of performance anomalies and client-workload overloading failures. A failure is predicted in the model-pairing configuration when, at least one of the models, predicts the corresponding failure. In another configuration of models, we explore the use of a *single-model* configuration (Figure 57) to classify log instances into pre-failure patterns and normalcy patterns, without distinguishing between client-workload over-

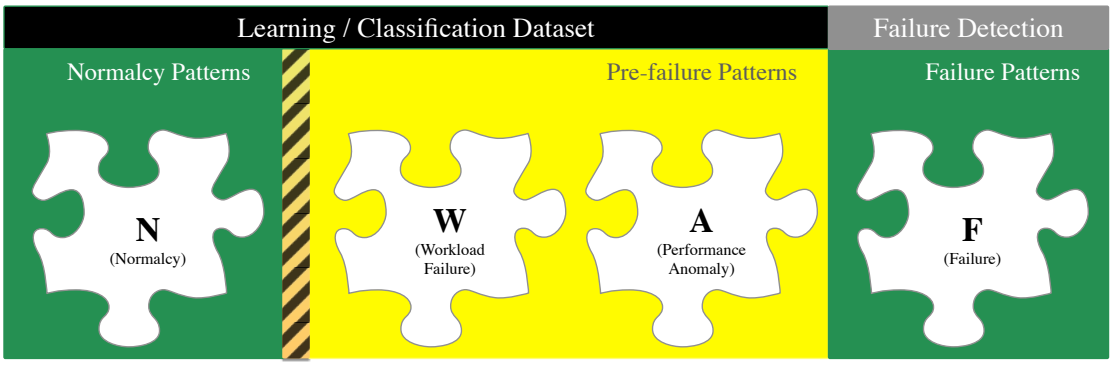


Figure 55: Classification of log instances into the different types of patterns.

loading and performance anomalies. This configuration avoids separation of instances associated to each fault type.

6.5.4 *Experimental Design and Preliminary Analysis*

This section presents the experimental methodology to evaluate the failure prediction performance of our approach in Pure Streaming services. The experimental process can be summarized as follows:

1. Run the mix+spike and mix+anomaly benchmarks, using a load generation tool. Each benchmark runs for 90 hours, during which metrics measurements are collected, at every second, for further analysis;
2. Integrate, clean and find relevant features on the logged data;
3. Build models to evaluate the impact that class unbalancing in the learning dataset has on prediction performance;
4. Build models to evaluate the classification performance improvement resulting from the inclusion of past data in the learning process — including metrics measurements gathered at time $t - i$ in the failure prediction process, being t the time of the last measurement and i a positive value;
5. Build models for the model-pairing and single-model configurations;
6. Breakdown of failure prediction results per each learning algorithm, failure severity level, look-ahead time and failure type;
7. Determining the impact of derived metrics on the failure prediction performance.

The classification performance of failure prediction models is evaluated using standard ten-fold cross-validation (described in Section 6.3.2).

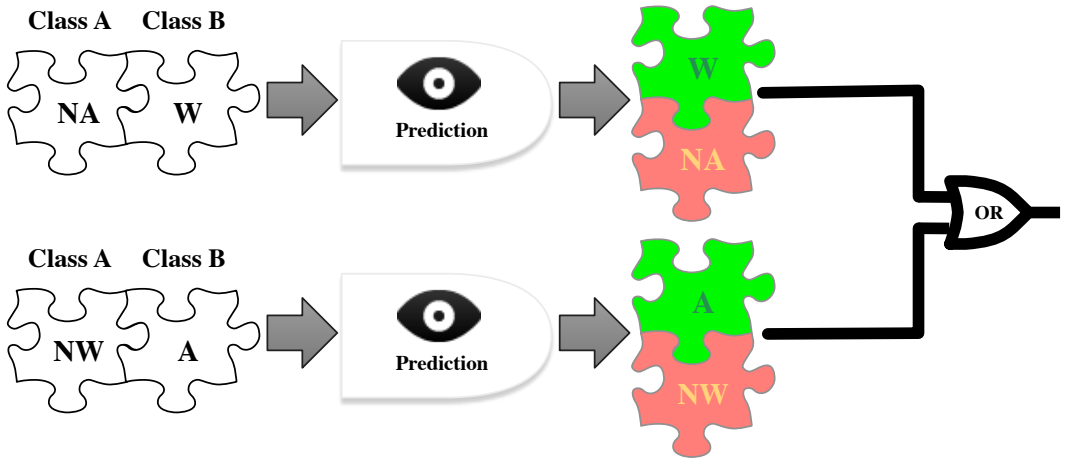


Figure 56: Failure prediction in the model-pairing configuration, using two models. One of these models separates pre-failure patterns associated to workload-related failures (W) from the patterns (N) and (A). Similarly, the other model separates pre-failure patterns associated to performance anomalies (A) from the patterns (N) and (W).

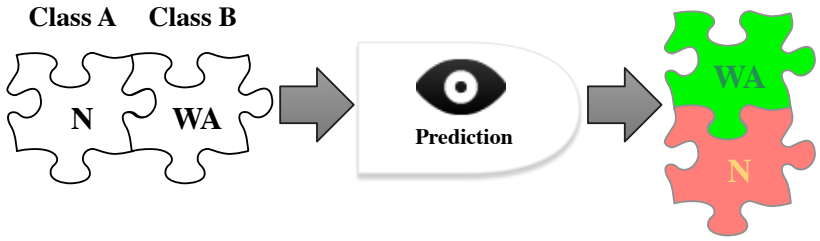


Figure 57: Failure prediction in the single-model configuration. Pre-failure patterns are separated from normalcy patterns, independently of their fault types.

6.5.4.1 *Prediction Performance Metrics and Semantics*

Section 6.3.1 presents several common metrics used to measure the classifiers' performance. From these, the recall and precision metrics are the most relevant, for the reasons already described.

Each event accounted by recall and precision represents one period of the benchmark. Benchmarks are structured into periods of normalcy with a duration of 10 minutes, each followed by a period of failure with a duration of 1 minute. Hence, a true positive is accounted the first time the failure is predicted before failure occurrence within the same period. On the other hand, one false positive is accounted when, within a period of normalcy, there is at least one erroneous prediction of failure.

6.5.4.2 *Preliminary Analysis of the Impact of Class Unbalancing*

The number of learning instances gathered during normal server periods can exceed several orders of magnitude the number of learning instances associated to failure periods, as most of the time the system is healthy. Learning datasets with this configuration may interfere with performance of classifiers. The reason is that some algorithms underperforms classification of the minority classes in class unbalanced datasets, giving higher significance to majority classes, when they hold a significantly higher number of learning instances compared with minority classes.

We evaluated the impact of compensation of class unbalancing on the failure prediction performance, using a cost model that benefit learning instances associated to pre-failure periods in the learning process. This is done by attributing them a weight inversely proportional to the number of occurrences of its class in the dataset. However, the experimental results obtained using the mix+spike and mix+anomaly benchmarks have shown that compensation of class unbalancing is unable to improve the classifiers' performance. Based on the results, we decide to perform experimental evaluation of models without compensation of class unbalancing.

6.5.4.3 *Preliminary Analysis of Prediction Using Past Data*

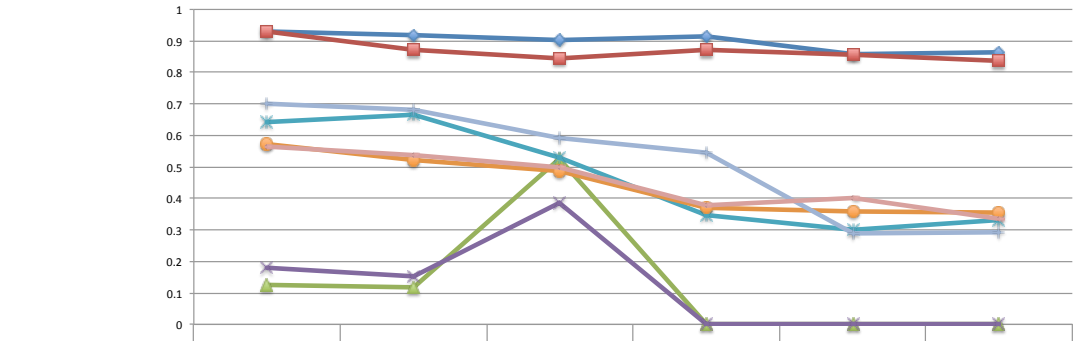
The failure prediction approaches presented in this chapter infer the server state F_t from patterns exposed by specific features values M_t, observed at the current time t. We consider an alternative configuration that combines both M_t and the features values observed in the past $M_{t-1}, M_{t-2}, ..., M_{t-n}$ to infer the server state F_t.

The rationale for the use of the sequence of features values observed along the time within the time window $[t-n, t]$ is that they expose a tendency. This approach comes at the cost of an increase in the number of features values equal to $|M_t| \times (n+1)$, resulting from the multiplication of the number of features $|M_t|$ by the number of past periods $n+1$ considered for each feature. Consequently, the data volume and processing costs would increase proportionally to the number of past events considered.

The experimental results obtained using the mix+spike and mix+anomaly benchmarks for different values of n, have shown that past features values do not improve the prediction performance of models. This observation can be explained by the redundancy of past values with the derived metrics, which already provides the tendency of metrics values along time. In face of the results, we avoid past features values in the learning process.

6.5.5 *Experimental Results in the Model-Pairing Configuration*

This section presents the results of the application of binary classification to failure prediction using the model-pairing configuration.



	5 Seconds	10 Seconds	15 Seconds	20 Seconds	25 Seconds	30 Seconds
Bayes Net Soft Failure	0.929	0.92	0.902	0.913	0.858	0.864
Bayes Net All Failures	0.929	0.871	0.844	0.871	0.855	0.835
SMO Soft Failures	0.124	0.116	0.527	0	0	0
SMO All Failures	0.179	0.153	0.386	0	0	0
Bagging Soft Failures	0.642	0.664	0.531	0.347	0.3	0.333
Bagging All Failures	0.574	0.523	0.488	0.371	0.358	0.355
J48 Soft Failures	0.7	0.682	0.59	0.547	0.289	0.293
J48 All Failures	0.564	0.539	0.5	0.377	0.403	0.335

Look-ahead Time

Figure 58: Recall using different learning algorithms in the model-pairing configuration.

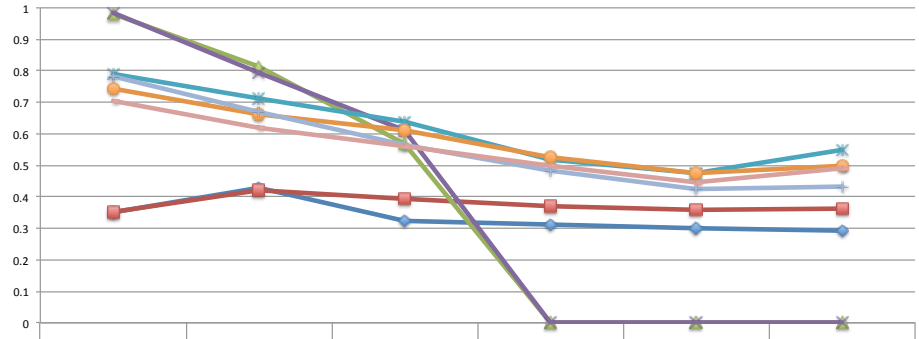
6.5.5.1 *Breakdown of Results by Classification Algorithm*

Figure 58 and Figure 59 present the classification recall and classification accuracy, partitioned by learning algorithm. The results are presented for:

- Look-ahead times that range from 5 to 30 microseconds;
- *Soft failures*, defined as frustration times higher than 6 seconds, according to the Keynote StreamQ grade metric (Chapter 4);
- *All failures*, comprising both hard failures (e.g., server crash leading to termination of connections) and soft failures.

Experimental results have shown that bayesian networks present the highest recall for look-ahead times up to 25 seconds. Recall values reach approximately 93% in both *soft failures* and *all failures* configurations, for look-ahead times of 5 seconds. Raising the look-ahead time to 10 seconds, only the *all failures* configuration decreases to approximately 87%, contrasting with *soft failures*, which suffers a decrease below 1%. Variations are small for larger look-ahead times, except for *soft failures*, which drops approximately 5% after 20 seconds. The best performance achieved by bayesian networks is counterposed with the low precision values, which never surpasses 43% in the entire range of look-ahead times.

The SVMs (SMO) results exhibit the opposite behavior of bayesian networks, showing low recall but high precision values for small look-ahead times. Recall values



	5 Seconds	10 Seconds	15 Seconds	20 Seconds	25 Seconds	30 Seconds
BayesNet Soft Failures	0.352	0.43	0.325	0.311	0.302	0.292
BayesNet All Failures	0.352	0.421	0.393	0.372	0.359	0.362
SMO Soft Failures	0.982	0.813	0.568	0	0	0
SMO All Failures	0.983	0.795	0.612	0	0	0
Bagging Soft Failures	0.792	0.714	0.637	0.517	0.474	0.551
Bagging All Failures	0.743	0.66	0.613	0.526	0.474	0.5
J48 Soft Failures	0.783	0.669	0.563	0.484	0.423	0.434
J48 All Failures	0.703	0.618	0.559	0.498	0.446	0.489

Look-ahead Time

Figure 59: Precision using different modeling algorithms in the model-pairing configuration.

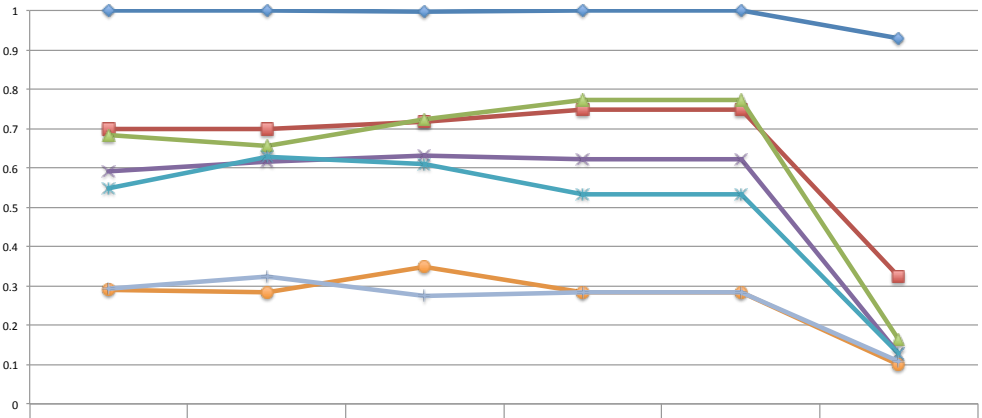
start at 12%, approximately, for *soft failures* with look-ahead times of 5 seconds and vary with time very irregularly, dropping to zero above 15 seconds. This behavior shows that models created with this algorithm have limited performance capturing pre-failure patterns.

Decision trees (J48) present the best balance between recall and precision. It starts with a recall of 70% and a precision of approximately 78%, for *soft failures* with look-ahead times of 5 seconds. Both metrics drop sharply for look-ahead times above 10 seconds. The precision of decision trees is slightly improved when combined with the Bagging meta-learner (approximately 79% for soft failures). However, the recall of *soft failures* is lower in Bagging (approximately 64%) than in decision trees.

Summing up, it is noticeable three global patterns from the analysis of look-ahead times and failure types. Firstly, it is visible a global decrease of recall and precision over look-ahead times varying from 5 to 20 seconds. Secondly, the prediction performance is smaller in the *all failures* configuration than in the *soft failures* counterpart. That means that models are less accurate in predicting hard failures. Finally, the algorithm with best recall has lower precision and vice-versa. Due to the equilibrated performance achieved by J48, we consider this algorithm in the further analysis.

6.5.5.2 *Breakdown of Prediction Performance by Failure Severity*

Failure severity is defined in terms of the frustration time associated to each Keynote StreamQ grade for soft failures and observation of connection failures for hard failures.



	>6 seconds	>9 seconds	>12 seconds	>15 seconds	>18 seconds	Conn Failure
Failure Time	1	1	0.998	1	1	0.928
5 Seconds	0.7	0.7	0.717	0.748	0.748	0.324
10 Seconds	0.682	0.655	0.722	0.771	0.771	0.164
15 Seconds	0.59	0.615	0.631	0.621	0.621	0.128
20 Seconds	0.547	0.628	0.609	0.534	0.534	0.128
25 Seconds	0.289	0.283	0.348	0.283	0.283	0.1
30 Seconds	0.293	0.323	0.274	0.285	0.285	0.108

Failure Severity

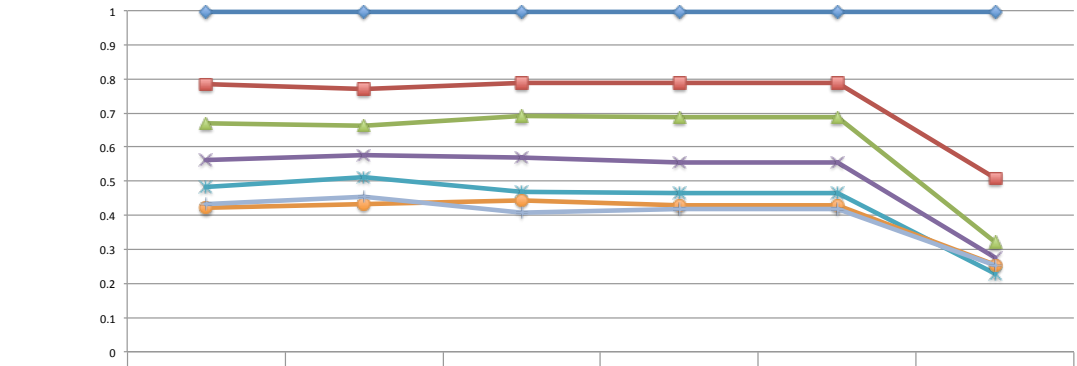
Figure 60: Recall values for different failure severity types and look-ahead times in the model-pairing configuration.

Accordingly, the lower limit of the frustration time for the least severe failure is 6 seconds, increasing in intervals of 3 seconds above 6 seconds (i.e., above 9, 12, 15 and 18 seconds) along with the severities. Hard failures are more aggressive than soft failures. They are manifested by client-server connection breaks occurring when users are watching or waiting for watching videos.

Figure 60 shows recall values for different look-ahead times and failure severity types. Recall is shown for failures classified at the failure time (without anticipation) and look-ahead times ranging from 5 to 30 seconds. In the former case, the feature values will determine whether the server is faulty or not, but after failure occurrence. That means that the server is capturing failure patterns, instead of pre-failure patterns.

At the failure time, recall is approximately 100% for all severities of soft failures, meaning that the model perfectly captures the pattern associated to a failure state, when prediction is performed without anticipation. Recall is positioned around 70% for soft failures with frustration times higher than 6 seconds and look-ahead times up to 10 seconds. Plus, recall increases slightly with increasing failure severity for soft failures, but falls significantly for hard failures (presented as connection failures).

Figure 61 presents the precision for different look-ahead times and failure severity types. The precision of all soft failures with a look-ahead time of 5 seconds is stable around 78%. For larger look-ahead times, the precision is stable along different failure



	>6 seconds	>9 seconds	>12 seconds	>15 seconds	>18 seconds	Conn Failure
Failure Time	0.998	0.998	0.995	0.995	0.995	0.996
5 Seconds	0.783	0.77	0.788	0.788	0.788	0.509
10 Seconds	0.669	0.664	0.693	0.688	0.688	0.32
15 Seconds	0.563	0.576	0.568	0.554	0.554	0.276
20 Seconds	0.484	0.51	0.47	0.465	0.465	0.229
25 Seconds	0.423	0.432	0.442	0.428	0.428	0.255
30 Seconds	0.434	0.454	0.406	0.419	0.419	0.252

Failure Severity

Figure 61: Precision values for different failure severity types and look-ahead times in the model-pairing configuration.

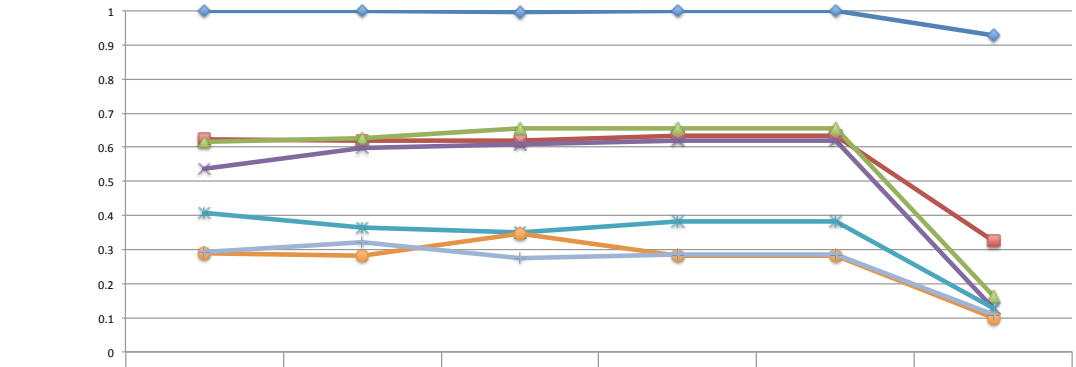
severity types, but falls roughly 10% per each additional 5 seconds of look-ahead time, up to 25 seconds.

6.5.5.3 *Impact of Derived Metrics on Failure Prediction Performance*

Figure 62 and Figure 63 present the recall and precision values, respectively, obtained by removing the derived metrics from the learning and classification processes. It is observable that the failure prediction performance is higher when the derived metrics are included. Recall values decay more than 10% when the derived metrics are removed from the learning and classification processes, in several failure severities and look-ahead times. The largest drop in precision is 3.9% and 6.5%, for look-ahead times of 5 seconds and 10 seconds, respectively. For larger look-ahead times, the precision drop is close to these values.

6.5.5.4 *Breakdown of Prediction Performance by Cause of Failure*

As described in Section 4.2.1, server overloading can occur as a result of workload type changes that obfuscate load control mechanisms or other related problems responsible for server overloading. On the other hand, performance anomalies occur due to server faults unrelated with server loads generated by client workloads. Figure 64 and Figure 65 present the breakdown of predicted failures into workload-related failures (presented as *client-workload overloading*) and *performance anomalies*, respectively.



	>6 seconds	>9 seconds	>12 seconds	>15 seconds	>18 seconds	Conn Failure
Failure Time	1	1	0.998	1	1	0.928
5 Seconds	0.624	0.618	0.619	0.633	0.633	0.324
10 Seconds	0.616	0.625	0.655	0.657	0.657	0.164
15 Seconds	0.538	0.597	0.607	0.619	0.619	0.128
20 Seconds	0.409	0.365	0.35	0.384	0.384	0.128
25 Seconds	0.289	0.283	0.348	0.283	0.283	0.1
30 Seconds	0.293	0.323	0.274	0.285	0.285	0.108

Failure Severity

Figure 62: Recall values for different failure severity types and look-ahead times without derived metrics.

Prediction of soft failures caused by workload-related failures has recall values of 73.5% and 71.6% for look-ahead times of 5 seconds and 10 seconds, respectively. Recall drops significantly for larger look-ahead times. Plus, the precision is 82.2% for look-ahead times of 5 seconds, dropping considerably for larger look-ahead times.

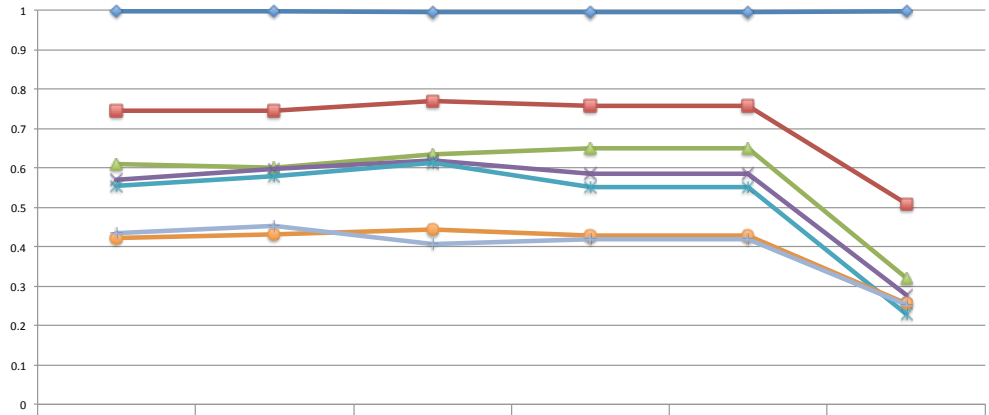
Prediction of performance anomalies exhibits lower performance than prediction of workload-related failures. The recall of performance anomalies for soft failures is 66.5% and 64.8% for look-ahead times of 5 seconds and 10 seconds, respectively, falling significantly above 10 seconds. The precision is 74.4% for look-ahead times of 5 seconds, falling significantly above 5 seconds.

6.5.6 *Experimental Results in the Single-Model Configuration*

We further present the experimental results obtained for the single-model configuration. They are examined for different classification algorithms and failure severities.

6.5.6.1 *Breakdown of Results by Classification Algorithms*

Figure 66 shows the recall of each learning algorithm. All learning algorithms, except the SMO, exhibit recall values above 95% for soft failures and look-ahead times of 5 seconds. J48 has the highest performance, with 98.6% of recall, followed by Bagging



	>6 seconds	>9 seconds	>12 seconds	>15 seconds	>18 seconds	Conn Failure
Failure Time	0.998	0.998	0.995	0.995	0.995	0.996
5 Seconds	0.744	0.744	0.768	0.756	0.756	0.509
10 Seconds	0.61	0.599	0.633	0.65	0.65	0.32
15 Seconds	0.568	0.598	0.619	0.585	0.585	0.276
20 Seconds	0.554	0.579	0.612	0.551	0.551	0.229
25 Seconds	0.423	0.432	0.442	0.428	0.428	0.255
30 Seconds	0.434	0.454	0.406	0.419	0.419	0.252

Failure Severity

Figure 63: Precision values for different failure severity types and look-ahead times without derived metrics.

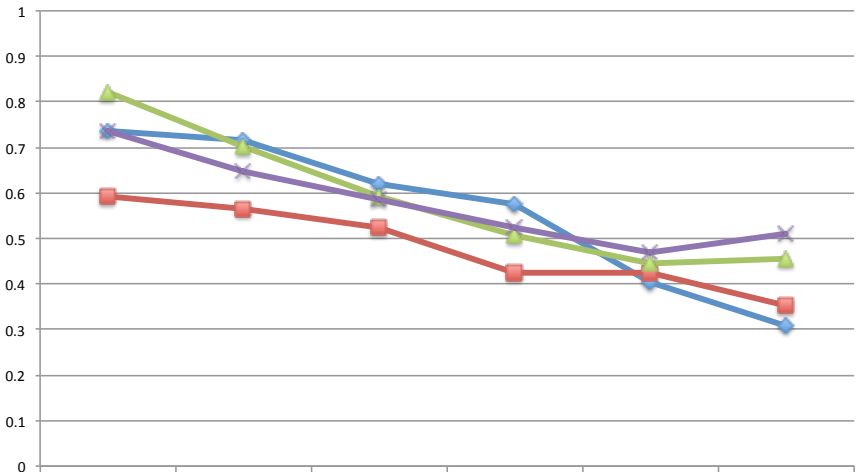
and bayesian networks with 97.8% and 95.2%, respectively. SMO presents the worst performance, with 13.1% of recall.

Figure 67 presents the precision of each learning algorithm. Bayesian networks present the lowest precision (36.5%) of all algorithms in the *soft failures* configuration, following the pattern of the model-pairing configuration. On the contrary, SMO shows the highest precision (99.7%) of all algorithms. J48 achieved the second highest precision of all algorithms, with 99.1% in the soft failures configuration. It is followed by Bagging, with 98.5% for the same configuration.

Following the same pattern of the model-pairing configuration, the recall and precision values decay with the increase of the look-ahead time. Also, both metrics have lower values in *all failures* configuration than in the *soft failures* counterpart.

6.5.6.2 *Breakdown of Prediction Performance by Failure Severity*

In the model-pairing configuration, the J48 algorithm is used in the analysis of the prediction performance for several failure severities. The reason behind this choice is the highest prediction performance demonstrated by this algorithm in the experimental evaluation. Likewise, we use the same algorithm, for the same reasons, in the single-model configuration.



	5 Seconds	10 Seconds	15 Seconds	20 Seconds	25 Seconds	30 Seconds
Recall Soft Failures	0.735	0.716	0.62	0.574	0.403	0.308
Recall All Failures	0.592	0.566	0.525	0.426	0.423	0.352
Precision Soft Failures	0.822	0.702	0.591	0.508	0.444	0.456
Precision All Failures	0.738	0.649	0.587	0.523	0.468	0.51

Figure 64: Recall and Precision of workload-related (client-workload overloading) failures.

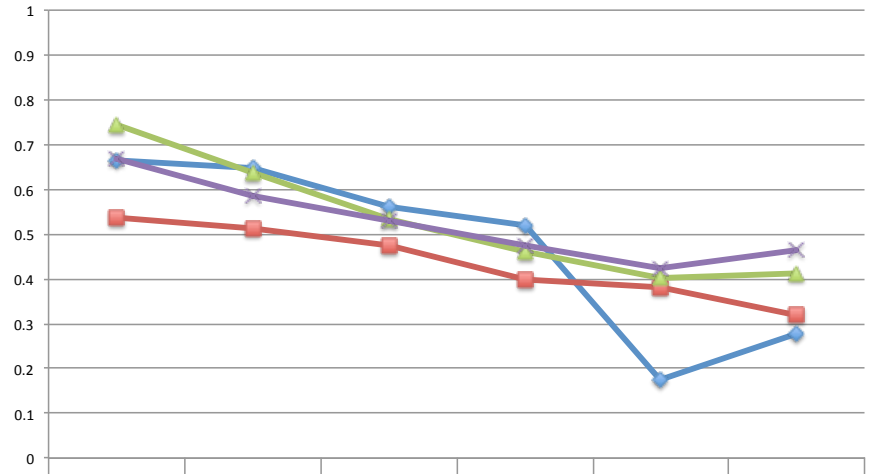
The breakdown of recall and precision into failure severities is presented in Figure 68 and Figure 69, respectively. Similarly to the results observed for the model-pairing configuration, both recall and precision are roughly stable on all severities of the *soft failure* configuration. As well, these metrics have significantly lower values in the prediction of hard failures than in the prediction of any other severities established for soft failures.

Failure patterns obtained from feature values after failure occurrence are classified with recall and precision of 100% or very close to 100%. These results are similar to those obtained for the model-pairing configuration.

6.5.7 *Discussion of Results*

The experimental results presented before are summarized as follows:

- Failure prediction models created for the single-model configuration have significantly better performance than those created for the model-pairing configuration;
- Bayesian networks (TANs) and Support Vector Machines have low performance, exposed either in terms of recall or precision, making these algorithms less attractive for creating failure prediction models;



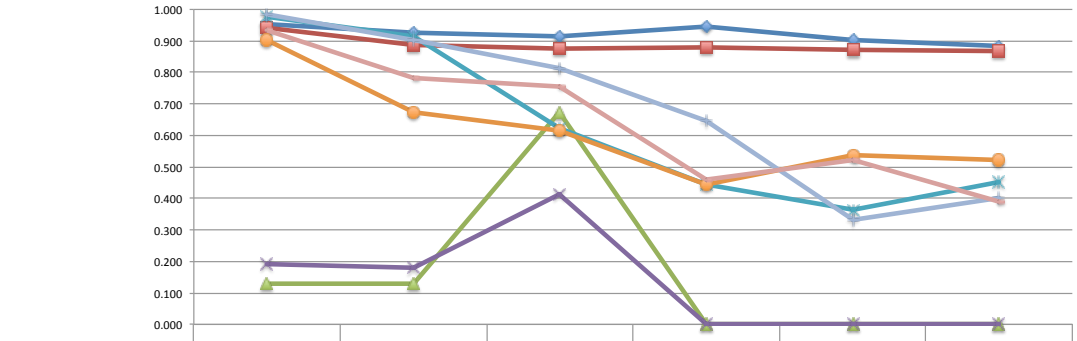
	5 Seconds	10 Seconds	15 Seconds	20 Seconds	25 Seconds	30 Seconds
Recall Soft Failures	0.665	0.648	0.561	0.52	0.175	0.278
Recall All Failures	0.536	0.512	0.475	0.398	0.383	0.318
Precision Soft Failures	0.744	0.636	0.535	0.46	0.402	0.412
Precision All Failures	0.668	0.587	0.531	0.473	0.424	0.465

Figure 65: Recall and Precision of performance anomalies.

- Decision trees (J48) present the best equilibrium between recall and precision, achieving recall of 98.6% and precision of 99.1%, for look-ahead times of 5 seconds;
- Ensembles of decision tree models are unable to improve the performance of single decision tree models;
- Prediction of performance anomalies is less accurate than prediction of workload-related failures;
- Generally, recall and precision values decrease with the increase of the look-ahead time;
- The prediction performance is stable over failure severities for soft failures, but drops considerably for hard failures;
- Derived metrics increase the failure prediction performance;
- When the server state is inferred from feature values after failure occurrence (without anticipation), both recall and precision are 100% or close to 100%.

The discussion of results is conveyed by the following vectors of analysis:

- The impact of decisions made using inaccurate failure predictions on the service performance;



	5 Seconds	10 Seconds	15 Seconds	20 Seconds	25 Seconds	30 Seconds
Bayes Net Soft Failure	0.952	0.928	0.916	0.947	0.901	0.884
Bayes Net All Failures	0.941	0.887	0.874	0.881	0.873	0.867
SMO Soft Failures	0.131	0.129	0.675	0	0	0
SMO All Failures	0.191	0.181	0.413	0	0	0
Bagging Soft Failures	0.978	0.915	0.622	0.443	0.364	0.453
Bagging All Failures	0.904	0.672	0.615	0.444	0.539	0.523
J48 Soft Failures	0.986	0.901	0.814	0.646	0.332	0.401
J48 All Failures	0.935	0.782	0.755	0.458	0.523	0.39

Look-ahead Time

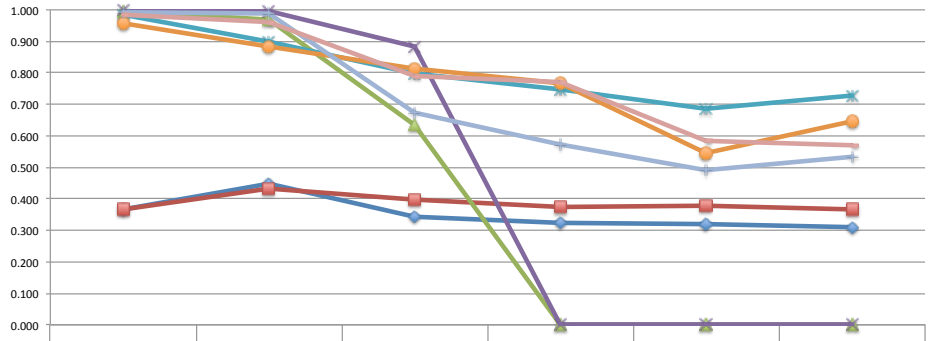
Figure 66: Recall using different learning algorithms in the single-model configuration.

- Breakdown of prediction performance into learning algorithms;
- Breakdown of prediction performance into failure severities;
- Comparison of the look-ahead time provided by prediction with that required for proactive recovery;
- Breakdown of prediction performance into workload-related failures and performance anomalies;
- Failure prediction efficiency.

6.5.7.1 *Impact of Failure Prediction Errors on the Service Performance*

Recall and precision have different service performance costs when prediction models are employed to decide whether the system requires recovery or not. These costs depend on the characteristics of the repair actions executed. Low-cost repair actions can absorb the impact of false positives (exposed by precision), giving preference to models with high recall values, even though they sacrifice precision to some extent. On the other hand, repair actions with significant performance costs demand predictors with high precision levels to minimize the execution of unnecessary repair actions.

Despite the small percentage of false positives absorbed by the precision metric for decision trees, the consequently unnecessary execution of repair actions can contribute to significant performance penalties, if expensive repair techniques are em-



	5 Seconds	10 Seconds	15 Seconds	20 Seconds	25 Seconds	30 Seconds
BayesNet Soft Failures	0.365	0.448	0.343	0.322	0.319	0.310
BayesNet All Failures	0.365	0.431	0.397	0.376	0.377	0.366
SMO Soft Failures	0.997	0.968	0.635	0	0	0
SMO All Failures	0.999	0.997	0.885	0	0	0
Bagging Soft Failures	0.985	0.899	0.798	0.747	0.686	0.728
Bagging All Failures	0.956	0.882	0.812	0.767	0.545	0.645
J48 Soft Failures	0.991	0.987	0.673	0.571	0.491	0.532
J48 All Failures	0.986	0.962	0.791	0.772	0.586	0.57

Look-ahead Time

Figure 67: Precision using different learning algorithms in the single-model configuration.

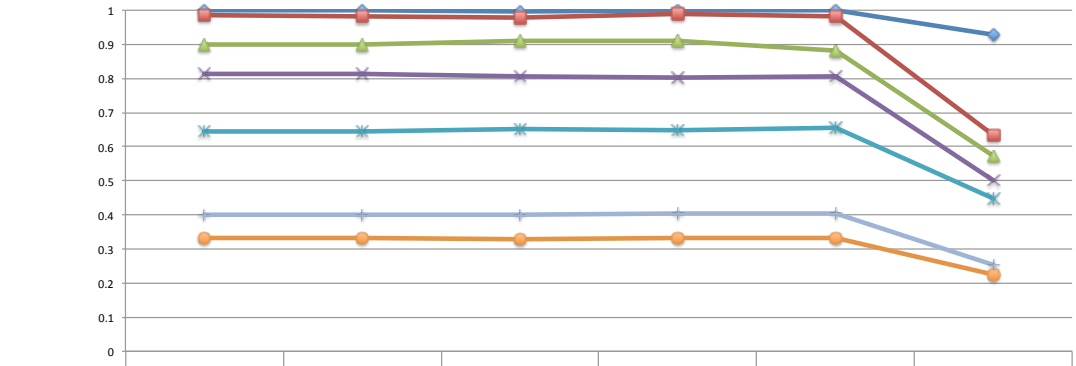
ployed. However, cheap repair techniques as those explored in Chapter 5 can avoid user-visible failures caused by the downtimes resulting from their execution. These techniques allow service recovery in approximately 3 seconds, in average, when the server is migrated between hosts and roughly 2 seconds when the server is restarted with a fresh state in the same or another host. Service downtimes of that order can usually be tolerated by client-side buffering.

6.5.7.2 *Breakdown of Prediction Performance into Learning Algorithms*

The highest recall and precision are achieved using the single-model configuration with the C4.5 decision trees and Bagging algorithms. TANs present high levels of recall but poor precision, indicating a high number of false positives. The worst classification performance is achieved by the SVM algorithm.

We attribute the low performance of SVMs to the sensitivity of this algorithm to outliers. Outliers are expected in the learning dataset due to the discrimination between pre-failure patterns and normal patterns. Pre-failure patterns represent feature values observed several seconds before failure occurrence. Thus, the time margin that separates the normal patterns from pre-failure patterns can be blurred and indistinct. This results in features values of pre-failure patterns misclassified as normal patterns and vice-versa.

The good performance of the C4.5 algorithm is justified by its robustness to outliers [Sheng et al.], since it prunes subtrees from decision trees to prevent overfitting



	>6 seconds	>9 seconds	>12 seconds	>15 seconds	>18 seconds	Conn Failure
Failure Time	1	1	0.998	1	1	0.928
5 Seconds	0.986	0.983	0.98	0.988	0.983	0.633
10 Seconds	0.901	0.9	0.909	0.91	0.88	0.572
15 Seconds	0.814	0.814	0.805	0.801	0.806	0.501
20 Seconds	0.646	0.644	0.652	0.648	0.656	0.448
25 Seconds	0.332	0.332	0.329	0.332	0.334	0.225
30 Seconds	0.401	0.401	0.401	0.403	0.404	0.253

Failure Severity

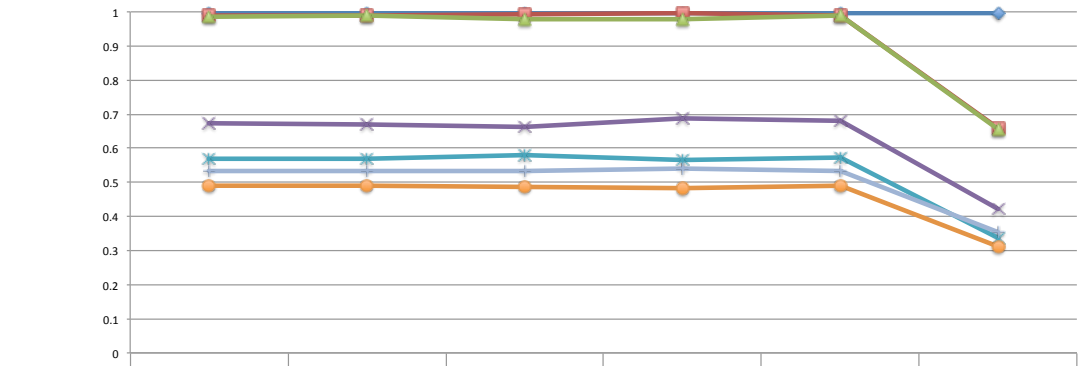
Figure 68: Recall values for different failure severity types and look-ahead times in the single-model configuration.

to noise in the data. We consider this characteristic essential to attain high levels of performance in our approach. The Bagging algorithm also creates classifiers robust to noise, since it trains individual models with log instances randomly chosen. Therefore, the likelihood for overfitting is reduced. However, it was unable to improve de performance of individual decision trees models, indicating the efficacy of the C4.5 algorithm dealing with outliers.

6.5.7.3 *Breakdown of Prediction Performance into Failure Severities*

Two failure conditions were considered for experimental analysis: hard failures and soft failures. Both failure conditions respect performance failures associated to a specific severity. Hard failures are more severe than soft failures. They represent fail-stop failures usually preceded by soft failures. On the other hand, soft failures are classified in several severity levels indexed by the Keynote StreamQ Grade metric (Chapter 4).

Experimental results demonstrated that the failure prediction performance is homogeneous on all soft failures severities. Yet, soft failures are significantly more predictable than hard failures. This is explained by the instability of the server behavior before the occurrence of hard failures. Hard failures caused by performance problems are usually preceded by degradation of service quality manifested as soft failures. However, the timespan between the moment the service quality starts degrading until the eminence of hard failures is often nondeterministic. Fault types and complex in-



	>6 seconds	>9 seconds	>12 seconds	>15 seconds	>18 seconds	Conn Failure
Failure Time	0.998	0.998	0.995	0.995	0.995	0.996
5 Seconds	0.991	0.99	0.993	0.997	0.99	0.658
10 Seconds	0.987	0.989	0.977	0.979	0.99	0.656
15 Seconds	0.673	0.671	0.664	0.686	0.68	0.421
20 Seconds	0.571	0.57	0.579	0.567	0.573	0.337
25 Seconds	0.491	0.491	0.488	0.483	0.489	0.312
30 Seconds	0.532	0.532	0.533	0.539	0.534	0.354

Failure Severity

Figure 69: Precision values for different failure severity types and look-ahead times in the single-model configuration.

teractions between system resources, operating system, applications and the network explains the difficulty of modeling pre-failure patterns that could hint to hard failures.

We consider that the low prediction performance of hard failures is tolerated by the high prediction performance of soft failures. As long as it is assumed that hard failures caused by performance problems manifest early as soft failures, it is possible to anticipate failure occurrences only by predicting soft failures.

6.5.7.4 *Prediction Look-ahead Times Required for Proactive Recovery*

The larger the look-ahead time provided by failure prediction, the broader the set of repair techniques that can be employed to restore the service without end-users noticing a performance failure. As previously shown, the look-ahead time of 5 seconds is enough to execute the repair techniques addressed in this thesis. However, prediction performance results are also presented for larger look-ahead times, for completeness.

6.5.7.5 *Breakdown of Failure Prediction Performance into Failure Causes*

The breakdown of failure prediction performance into failure causes has shown that workload-related failures are more predictable than performance anomalies, even though these differences are unsubstantial. This condition is explained by strong regularities when the server is overloaded by client workloads. That means that the overloading states have a regular reflection on the feature values used for classification. On the

contrary, performance anomalies would likely generate more irregular server behaviors with lower repeatability than workload-related problems. That means that the server states captured by performance metrics are less consistent between failure occurrences, generating variance that blurs the distinction between normalcy patterns and pre-failure patterns. This makes identification of pre-failure patterns less obvious for this type of failures.

The differences of performance between failure causes are specific to the model-pairing configuration, where log instances are classified by independent models, each bound to a failure cause. In the single-model configuration, these failure types are classified together, avoiding the evaluation of failure types individually. However, the high levels of the global recall and precision achieved for both failure causes are only attainable when each failure cause individually exhibits similar values for the same metrics.

6.5.7.6 *Prediction Efficiency*

In online failure prediction, it is mandatory to perform efficient classification of log instances. Short classification delays avoid the reduction of the anticipation time provided by failure prediction, which is required to execute proactive recovery actions.

In our experiments, the classification of log instances is performed by prediction models with latencies in the order of a few microseconds. So, the classification overhead gives a negligible contribute to the total failure prediction delay, which sums up the failure prediction latency to the monitoring latency (including data gathering, data preparation and failure detection). Overheads in the order of microseconds are unnoticeable for look-ahead times measured in the order of seconds.

6.6 FAILURE PREDICTION IN HTTP STREAMING

The idiosyncrasies of each video-streaming technique mold the server behavior during normal and faulty periods. As explained in Section 2, Pure Streaming servers implement a rigorous transmission rate control mechanism. Data transmission rates are regulated by the server with the support of the RTP protocol, to ensure: (1) data transmission at the same rate the data will be played out at the receiver; and (2) timely delivery of video-streaming fragments for playback. By contrast, in HTTP Streaming, the data are downloaded over HTTP by players similarly to typical web objects — at the maximum speed allowed by the server and network conditions. Hence, the data flow between the server and players is independent of the data received or played out by players.

The differences between the logic behind HTTP Streaming and Pure Streaming systems will likely lead to different normalcy patterns and pre-failure patterns. This section evaluates the failure prediction activity in the HTTP Streaming infrastructure. This

activity implements learning, classification and evaluation of prediction models built using several online learning algorithms [Witten et al. 2011].

6.6.1 *Research Questions*

The evaluation of our failure prediction approach in HTTP Streaming services is performed to answer the following fundamental questions:

1. Do online learning models accurately classify pre-failure patterns in HTTP Streaming services?
2. What is the performance overhead of the SHStream infrastructure?
3. How effective is the load control approach in avoiding server overloading?
4. Which online learning algorithms have the best prediction performance?
5. What is the breakdown of failure prediction performance by fault types?
6. How many learning instances are required until stabilization of the models' classification performance?

Answering these questions will unveil the ability of SHStream to predict performance failures.

6.6.2 *Online Learning Algorithms*

Online learning algorithms train models iteratively with the arrival of each log instance. These algorithms allow prediction even when only a small number of log instances are available to train models, allowing further incremental learning. Additionally, this learning scheme is also known to have low memory and CPU footprints. Online learning counterposes with batch learning, which aggregates all log instances and then train models at once, avoiding further updates.

By continuously training prediction models, it is possible to increase gradually the coverage of patterns associated to each classification category. Therefore, models evolve iteratively, without requiring an initial amount of training data enough to cover all normalcy patterns and pre-failure patterns. This is an important characteristic, since performance anomalies are usually rare events with low repeatability. Thus, the ability of learning algorithms to enrich models with new pre-failure patterns associated to performance anomalies is an important feature of the self-healing infrastructure.

Different online learning algorithms will likely produce models with different accuracies. With the purpose of continuously improving the prediction performance, SHStream is designed to accept new online learning algorithms dynamically to train new models iteratively and thus, exploring different approaches to find better models.

SHStream is evaluated in this chapter using *data stream mining*, a recent class of data mining techniques that overcomes traditional batch learning limitations to fulfill the online learning requirements of dynamic systems [Fayyad et al. 1996][Kuncheva 2004b][Bifet and Kirkby 2009], namely:

1. **Incremental learning** — learn models using a single block of data at the time;
2. **Single pass through data** — only one pass through the data is required for learning;
3. **Limited time and memory** — instances are processed in a small and constant time, using an approximately constant amount of memory;
4. **Any-time learning** — if stopped before its conclusion, the algorithm should provide the best possible answer.

Learning algorithms with the above characteristics can accurately train models iteratively using large streams of data, concurrently with the classification activity. So, enrichment of models using new learning instances can be performed anytime without rebuilding models from scratch. We evaluate three types of online algorithms in our framework: *decision trees*, *probabilistic classifiers* and *ensemble algorithms*.

6.6.2.1 *Decision Trees*

Decision trees are popular learners in both batch learning and online learning scenarios. They are powerful, interpretable and efficient classifiers — with n learning instances and m attributes, the average cost of basic decision tree induction is $O(m \cdot n \cdot \log n)$.

Traditional decision trees algorithms are non-parametric (without assumptions about the underlying data) and powerful learning approaches used for classification in batch learning scenarios. C4.5 is one of the most popular methods for building decision trees in batch learning (Section 6.5.2).

Hoeffding Trees promise performance levels similar to decision trees in batch learning scenarios, with the advantage of fulfilling the online learning requirements. The performance of Hoeffding Trees has been shown comparable with traditional decision trees, Naïve Bayes, k-NN, and ensemble algorithms [Bouckaert 2003][Babcock et al. 2002] but much faster and less memory consuming handling extremely large datasets than these approaches.

Very Fast Decision Tree (VFDT) is a state of the art algorithm for creating Hoeffding Trees proposed by Domingos and Hulten [Domingos and Hulten 2000]. VFDT builds a

three iteratively, by splitting each node when the number of learning instances learned satisfies the *hoeffding bound* formulated in (19).

$$\epsilon = \sqrt{\frac{R^2 \cdot \ln(\frac{1}{\delta})}{2 \cdot n}} \tag{19}$$

The hoeffding bound defines the split confidence, by stating that with probability $1 - \delta$, the true mean of a random variable with range R does not differ from its estimated mean by more than ϵ, after n independent observations.

Information gain is one splitting criteria commonly used to build tree models, which is also used in Hoeffding Trees. It is defined as the difference between the *weighted average entropy* of the splitted subsets and the *entropy* of the class distribution before splitting, being the entropy defined as in (20). The entropy measures the purity of subsets for a distribution of class labels, consisting of fractions $p_1, ..., p_n$, summing to 1.

$$entropy(p_1, p_2, ..., p_n) = \sum_{i=1}^{n} -p_i \cdot \log_2 p_i \tag{20}$$

Classification in Hoeffding Tree models can be performed using two strategies: Majority Class and Naïve Bayes. Majority Class is the classification strategy by default. It filters down the tree from the root to a leaf and retrieves the most likely class label, which is the one that appears more times associated to the classified attribute values. This strategy is improved by Naïve Bayes, which also accounts conditional probabilities of attribute values in the tree, given each class. Counts are stored at leafs to measure how many times the Naïve Bayes has outperformed the Majority Class predicting the corresponding class. Therefore, the leaf only returns the Naïve Bayes prediction when its accuracy is higher than the Majority Class. Otherwise it returns the Majority Class prediction. Naïve Bayes has been shown superior to Majority Class for certain types of problems [Holmes et al. 2005].

6.6.2.2 *Probabilistic Classifiers*

Naïve Bayes represents the class of probabilistic classifiers in SHStream. Naïve Bayes is naturally incremental as it deals with heterogeneous data and missing values and is a very competitive algorithm for small datasets [Agrawal et al. 1993]. A more detailed explanation of this classifier is provided in Section 6.5.2.

6.6.2.3 *Ensemble Algorithms*

Ensemble models perform classification by means of voting, using outcomes given by several models. That configuration has been proven to attain higher levels of accuracy

than those obtained by single classifiers alone [Tumer and Ghosh 1996]. The diversity of the representation of model concepts by ensemble models reduces overfitting, providing robustness to noise in the log data.

SHStream implements four ensemble algorithms that use Hoeffding Trees as base models:

WEIGHTED MAJORITY ALGORITHM [LITTLESTONE AND WARMUTH 1989] Learn models by giving a positive weight α to each model in the pool that correctly classifies one learning instance and discounting a given ratio β of the weight of models that incorrectly classify the learning instance. The number of mistakes m was proven to be bounded in a sequence of predictions from a pool of algorithms A by $O(\log|A| + m)$, if one of the algorithms of A makes at most m mistakes.

OZABOOST [OZA AND RUSSELL 2001] Learn several models in a sequence, increasing weights of learning instances misclassified by former models disposed in the sequence. This approach reinforces learning, in the subsequent models of the sequence, of learning instances misclassified in the previous models, similarly to Adaboost [Freund and Schapire 1995] for batch learning scenarios. This algorithm divides the total weights into two halves, giving one half to correctly classified learning instances and the other half to misclassified learning instances. Misclassified learning instances are reinforced intrinsically during the learning process at the next model of the sequence by the classifier's accuracy — as the classifier's accuracy increases, the number of misclassified learning instances decreases. Consequently, by dividing the total weights by less learning instances, they will receive more weight individually.

OZABAG [OZA AND RUSSELL 2001] Learn models from a *bootstrap replicate* of learning instances drawn randomly from the training dataset according to a probability $\mathrm{Poisson}(1)$, similarly to Bagging [Breiman 1996b] for batch learning, presented in Section 6.5.2. *Bootstrapping* reduces variance errors caused by learning instances with a small number of repetitions in the dataset, since they have low probability of being used to train models.

HOEFFDING OPTION TREES [PFAHRINGER ET AL. 2007] Build Hoeffding Trees with additional *option nodes* — equivalent to several Hoeffding Trees build upon the same tree structure. By representing several decision trees in a single compact structure, it is possible to reduce the space required to save independent tree instances, as required for traditional ensembles. Additionally, contrasting with other ensemble models, the model interpretability can be preserved if a small number of option nodes are used [Kohavi and Kunz 1997].

6.6.2.4 *Concept Drift Algorithms*

The models' classification performance can drop significantly after software updates, system configuration changes or hardware changes. This problem is known in machine learning as *concept drift*. The high frequency of software updates and the popularization of virtualization in system infrastructures are important contributors of model staleness.

Virtualization technologies allow dynamic changes of the system resources available for the server application. Reallocation of server resources could change the server behavior in several ways. As an example, by doubling the virtual machine's memory available, the server behavior will likely change for the same workloads.

In offline learning, an inaccurate model can be replaced by a new model trained with more recent data. In online learning, there are approaches for dealing with the concept drift problem without retraining models from scratch. Several learning algorithms implement *forgetting mechanisms* using *sliding windows* and *fading factors* [Kuncheva 2004a][Gama et al. 2009]. These mechanisms allow abandonment of stale model components during the learning process.

Fading factors and the sliding windows algorithms have shown similar performance in online learning scenarios [Gama et al. 2009]. Fading factors are memory-less mechanisms that attribute less weights to older observations used to form models, forcing the change detection algorithm to focus on the most recent data. On the other hand, sliding window algorithms keep a variable-length window of recently seen observations, assuming that the window has the maximal length statistically consistent with the hypothesis there has been no change in the average value inside the window. That means that the window size varies to enclose only the later elements that together maintain the error below a given threshold. Adaptive Sliding Window Algorithm (ADWIN) [Bifet et al. 2009] is the most popular algorithm of this class.

Adaptive Hoeffding Trees is a variant of Hoeffding Trees exploited in our failure prediction approach that uses ADWIN to evolve trees with new server behaviors. ADWIN monitors the performance of branches in the tree and replace them with new branches with higher accuracy. Similarly, ensembles can be extended with ADWIN to handle the concept drift problem. We use ADWIN with OzaBoost and OzaBag in our failure prediction approach.

6.6.3 *Implementation of Failure Prediction*

Learning algorithms train prediction models using log instances labeled in terms of the patterns they represent: normalcy patterns or pre-failure patterns. These log instances respect failure-free periods. Log instances respecting failure periods (failure patterns) are recognized by the failure detector through analysis of service quality metrics and are used for evaluation purposes.

6.6.3.1 *Delimitation of the Pre-failure Period*

Pre-failure patterns are delimited by a time window that starts at one undetermined moment before the failure and finishes immediately before the failure. Thus, the variability of the size of that time window could lead to pre-failure patterns being confounded with normalcy patterns and vice-versa, introducing errors in the learning process.

Instead of performing a sharp delimitation of the *pre-failure window* (including all pre-failure patterns), we consider a short pre-failure window preceded by a *window of uncertainty*, as illustrated in Figure 70. Any data within the window of uncertainty are ignored in the learning process and only log instances within the pre-failure window are trained as *pre-failure patterns*. This solution minimizes data segmentation errors caused by incorrect separation of log instances pertaining to each pattern type.

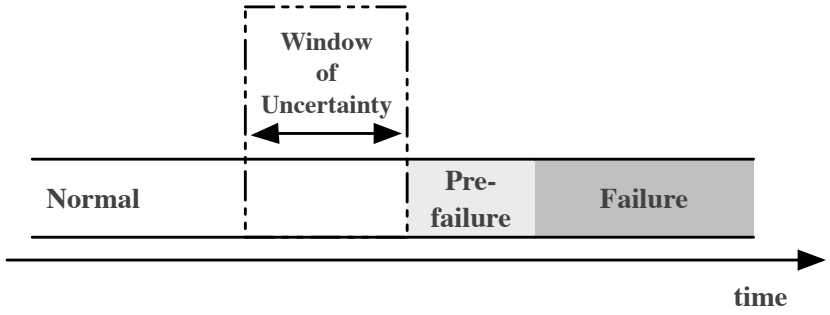


Figure 70: The window of uncertainty precedes the pre-failure window to avoid confounding pre-failure patterns with normalcy patterns.

6.6.3.2 *Methodology for Failure Prediction, Model Learning and Model Evaluation*

Algorithm 2 presents the process followed for failure prediction and for learning and evaluation of models. This process has the following steps:

1. One log instance is picked periodically, containing features values pertaining to one log time interval. It aggregates application, system and network metrics into one single row of data;
2. Derived metrics (Section 6.2) are calculated and added to the log instance;
3. In case the log instance belongs to a failure-free state[4], it is classified by all prediction models and the classification given by the model with higher performance up to that moment — determined from the accuracy of previous predictions — is chosen to decide if recovery is required;

4 The service state is provided by the failure detector.

Algorithm 2 Algorithm for: (1) Online Learning and Evaluation of Models; and (2) Online Failure Prediction.

```
Require: size(buffer) is WindowOfUncertainty
  loop
    I ⇐ readNewInstance()
    f ⇐ isFailState(I)                                          ▷ Classification
    if f is false then
      for i = 1 to nModels do
        p_i ⇐ classifyFailurePrediction(Model_i, I)
      end for
    end if
    if productionMode and (p_mostAccurate is Failure or f is true) then
      launchRecovery(DiagnosedProblem)
    end if                                                      ▷ Learning
    L ⇐ buildLearningInstance(I, f, p)
    addToEnd(buffer, L)
    if not isBufferFull(buffer) then
      jump to next loop iteration
    end if
    F ⇐ removeFirst(buffer)
    if distanceFail(buffer) ∈ [1, preFailWindow] then
      for i = 1 to nModels do
        learn(Model_i, F, Prefailure)
        updateModelStatistics(Model_i, F, Prefailure)
      end for
    else if distanceFail(buffer, F) is ∞ then
      if F.p_mostAccurate = Normal or learningMode then
        for i = 1 to nModels do
          learn(Model_i, F, Normal)
          updateModelStatistics(Model_i, F, Normal)
        end for
      end if
    end if                                                      ▷ Evaluation
    mostAccurate ⇐ evaluateBestModel(Model)
  end loop
```

4. One learning instance is built by combining the log instance, classifications performed by all algorithms and the current observable service state (Figure 71).

Each learning instance is stored at the end of a *circular buffer* (Figure 72) to defer the assessment of predictions until the end of the period covered by prediction. At that time, it is possible to determine whether the prediction result was accurately predicted. In the meantime, after being stored in the circular buffer, each learning instance slides over the buffer at every iteration — triggered by the arrival of a new learning instance — until reaching the first position of the buffer. Then, in the next iteration, the learning instance is removed from the queue to be assessed, according to the following rules:

- It is used to train models as a pre-failure pattern, if its distance to the next failure instance in the buffer is lower or equal than the size of the pre-failure window;
- It is used to train models as a normalcy pattern, if the buffer only contains failure-free instances;

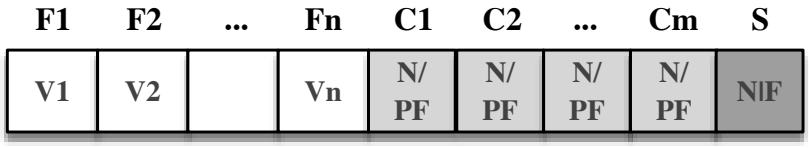


Figure 71: Structure of the learning instance with features values F, outcomes of classification models C (Normal or Pre-failure) and the actual service state S (Normal or Failure).

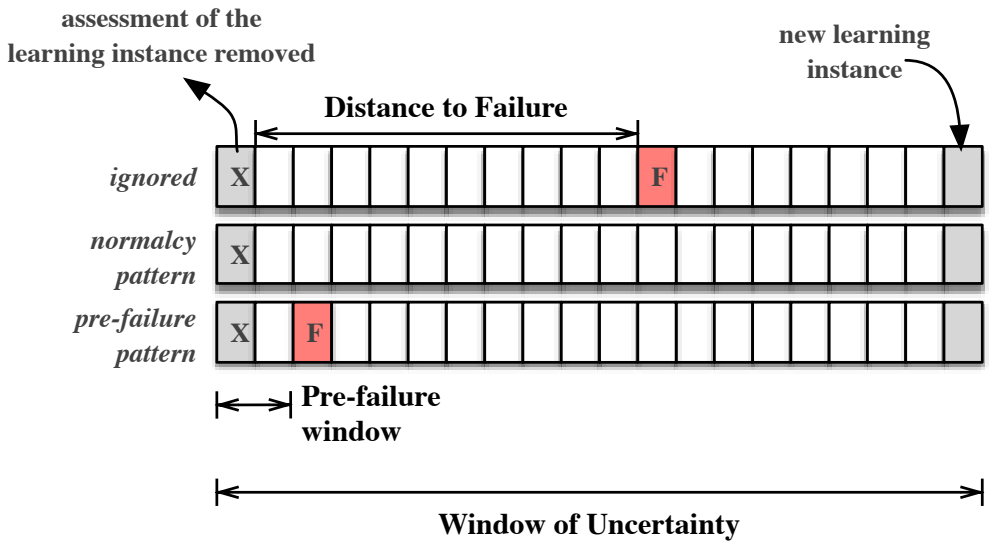


Figure 72: The three different learning scenarios.

- It is ignored otherwise, because it is outside the pre-failure window but within the window of uncertainty.

The circular buffer size is dimensioned by the size of the window of uncertainty. Thus, during the assessment phase — when the learning instance is removed from the buffer — if the buffer has at least one learning instance labelled as failure, thus any learning instance within the buffer but outside the pre-failure window is ignored for learning, since it belongs to the window of uncertainty. Otherwise, one learning instance is recognized as a pre-failure pattern in the learning process if its distance to the next learning instance labelled as failure is less or equal than the pre-failure window's size. Finally, when all learning instances in the buffer are labelled as normal by the failure detector, the learning instance under assessment is learned by models as a normalcy pattern, since the window of uncertainty only contains normal instances.

6.6.3.3 *SHStream Running Modes*

SHStream has two running modes: *production mode* and *learning mode*. In production mode, SHStream handles real workloads generated by end-users and organic faults activated during normal execution of the service. In learning mode, SHStream handles synthetic workloads and fault loads, devised to train and evaluate prediction models. The learning mode allows fast instantiation of prediction models and is also applied to scenarios where the learning data are unobtainable automatically.

In most services, performance anomalies occur with low frequency. That means that, prediction models can show low performance during long periods before sufficient data become available for learning. Using synthetic workloads and fault loads to initialize prediction models will likely increase prediction performance when SHStream has not gathered enough production data for learning. Also, in the production mode, there is one scenario where data are unavailable for learning. This scenario is depicted in Figure 73 and is described as follows. When failures are predicted and repaired, and are not followed by any failure occurrence, two conditions may have occurred:

1. The failure was correctly predicted and the execution of the repair action overcame the failure;
2. The failure was incorrectly predicted (false positive).

By reason of two different interpretations of the scenario where failures are predicted and not occurred, all log instances associated to that scenario cannot be recognized automatically as neither normalcy patterns nor pre-failure patterns. Consequently, that scenario is only available for learning when SHStream is running in learning mode, using synthetic workloads and fault loads with predetermined configurations.

6.6.4 *Experimental Design*

This section presents the configuration of the experimental work undertaken to evaluate the failure prediction activity in the SHStream framework. It presents the testbed and the metrics used to evaluate the performance of failure prediction models.

6.6.4.1 *Testbed*

The experimental testbed is specified in Section 4.5.1. SHStream is installed outside the virtual container to access global system metrics and is configured to gather performance metrics with a sampling interval of 2 seconds. That means that one log instance is generated every 2 seconds for classification and learning. SHStream requires that two important parameters be set: the size of the pre-failure window and the minimum look-ahead time of prediction.

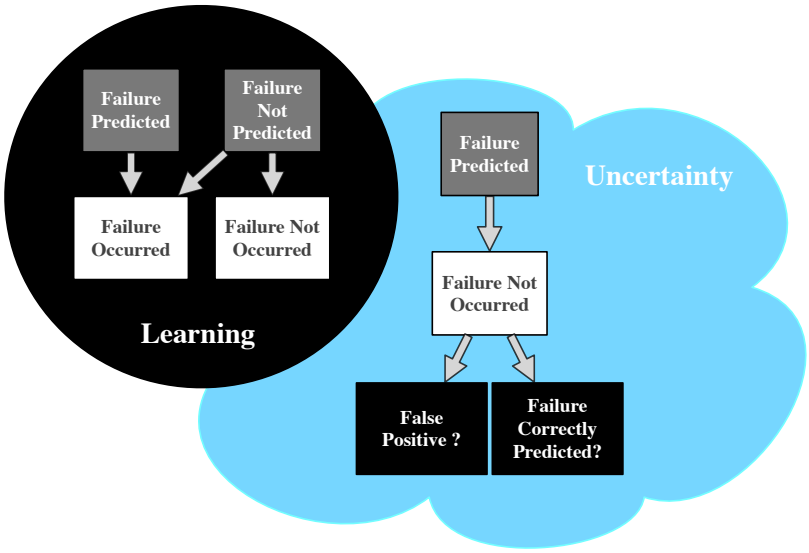


Figure 73: Scenarios where learning can and cannot be performed automatically in production mode.

The pre-failure window is set to the size of 2 learning instances. Larger pre-failure window sizes would theoretically gather a wider set of pre-failure patterns, but can introduce larger errors. This parameter is set empirically set, but can be adjusted later to exploit possible improvements of prediction performance.

The minimum look-ahead time of prediction is the second parameter to be set. Its value should be sufficient to absorb the execution time of repair techniques, to anticipate the occurrence of failures. Since server migration is the most lengthy technique evaluated in this thesis, the minimum look-ahead time of prediction should be set to allow the execution of this technique. As described in Chapter 5, server migration is the most lengthy technique, with execution times less than 3 seconds, in average. This value represents the time necessary to rescue the server checkpoint to a fallback host and resume it there. As long as each log instance classified corresponds to a time period of 2 seconds, we consider that one failure is predicted when the corresponding pre-failure pattern is correctly classified with a look-ahead time of at least 2 log instances (4 seconds) before the failure (Figure 74).

6.6.4.2 *Evaluation of Prediction Performance*

In batch learning, the performance of prediction models is evaluated once using the entire dataset (e.g., through cross-validation). In online learning, the learning activity rolls out continuously and simultaneously with the classification of log instances. Consequently, the prediction performance requires a continuous evaluation, instead of a one-time evaluation of the whole dataset.

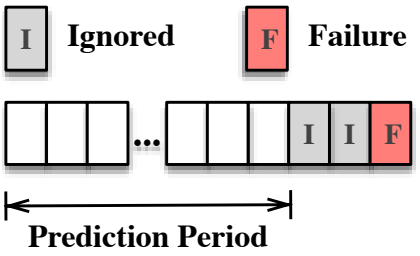


Figure 74: A failure is considered predicted when the respective pre-failure pattern is correctly classified at least 4 seconds (2 learning instances) before the failure.

The models' prediction performance is evaluated in terms of the:

- False positives and false negatives over the number of learning instances;
- Misclassifications of each fault type (CPU, memory, I/O);
- Recall, precision and f-measure of each learning algorithm over time;
- SHStream execution overhead during non-faulty periods.

Since SHStream performs load control (described in Section 4.4.4), we do not expect workload-related failures. In consequence, only failures caused by performance anomalies are expected. However, we evaluate both the ability of: (1) SHStream in controlling the server load; and (2) classifiers in predicting failures caused by performance anomalies.

The metrics used for the evaluation of the classification performance of each model are calculated by the SHStream application. These metrics are required to select the classification outcome provided by the model with the best performance at the end of each classification iteration — used to decide whether to repair the server or not.

6.6.5 *Experimental Results*

Experimental results are presented with the following order:

1. Effectiveness of the load control mechanism;
2. SHStream overheads during fault-free periods;
3. Failure prediction performance.

A discussion of the experimental results follows their presentation.

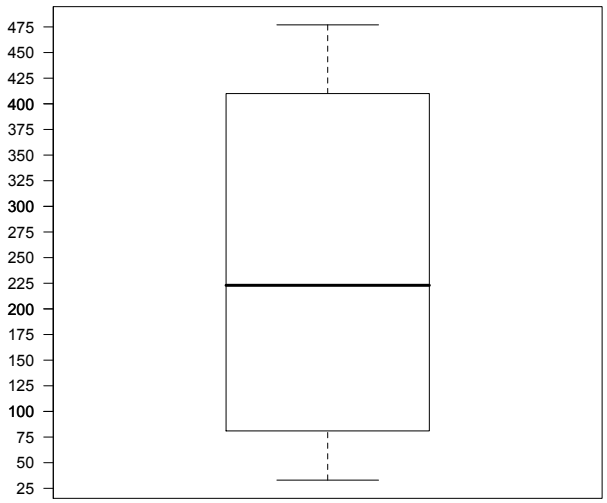


Figure 75: Minimum value, lower quartile, median, upper quartile and maximum value of the transition gap.

6.6.5.1 *Load Control*

The ability of the load control mechanism to avoid server overloading by controlling the acceptance of new connections was evaluated using a safe margin $\alpha = 0.05$ (described in Section 4.4.4). During the entire run, we observed that:

- No workload-related failures were observed during the execution of the benchmarks;
- The transition gap[5] varies between 81ms in the first quartile and 410ms in the third quartile (Figure 75);
- Only 2 unnecessary connection redirections were observed during the execution of the benchmarks.

As expected, unnecessary redirections can occur if the server receives several simultaneous requests, when it is approaching the edge of its capacity.

5 Period during which the server rejects new requests due to the impact of recent requests, with no data transmitted by the server, on the *excess margin* (metric that controls the acceptance of requests).

6.6.5.2 *SHStream Overheads*

The SHStream overheads were measured by running the benchmarks twice. In the first run, the benchmarks were run simultaneously with the execution of SHStream. In the second run, the SHStream application was deactivated during the execution of benchmarks and the number of requests was restrained to avoid server overloading.

We observed that the server capacity, measured by the maximum number of simultaneous connections handled by the server without service degradation, is the same whether the SHStream application is running or not. The explanation for that observation is that video-streaming services are resource intensive and the resources consumed by SHStream are negligible when compared with those consumed by video-streaming requests individually.

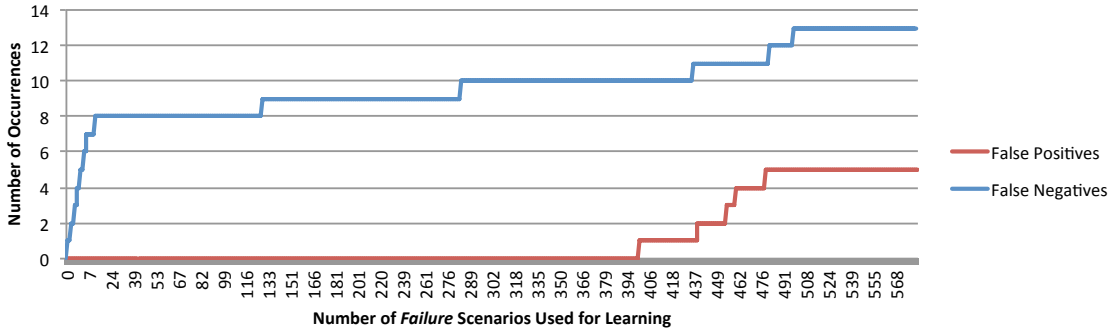
6.6.5.3 *Failure Prediction Results*

Figure 76 presents the models' classification performance over the number of failure scenarios of the benchmarks used for learning. Figure 76a relates the number of false positives[6] and false negatives[7] with the number of failure scenarios. Each classification outcome (normalcy pattern or pre-failure pattern) is obtained by selecting, at each iteration, the classification outcome given by the model with best performance. It is noticeable 5 false positives, which occur after a large number of failure scenarios. On the other hand, the number of false negatives stabilizes at 8 occurrences after 15 failure scenarios, occurring infrequently afterwards. Figure 76b shows that the best classifier predicts consistently failures between fault types.

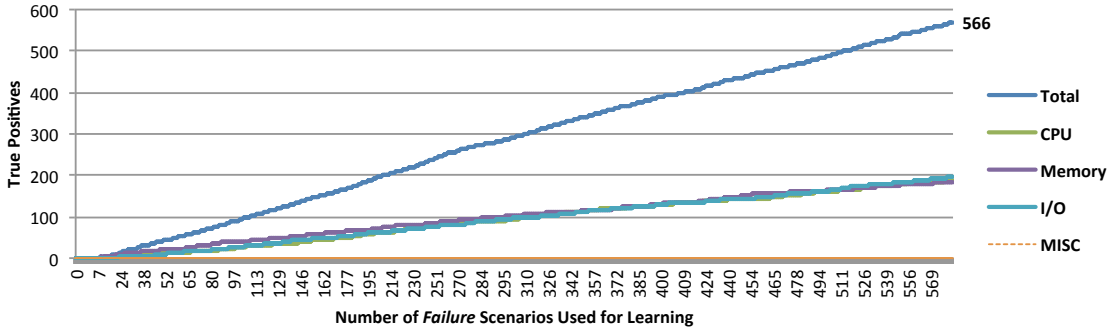
Figure 77a and Figure 77b present the recall and precision of each learning algorithm. It is observed that all algorithms, except the ADWIN algorithms and the Naïve Bayes, achieved high levels of recall and precision in the benchmark. The recall of these algorithms approximates to 98%. On the other hand, the precision approximates to 99% in the same algorithms except the Hoeffding Trees. Standard Hoeffding Trees achieved 98% of precision, a little below its ensemble variant. Ensemble algorithms achieved the highest prediction performance, but with small differences when compared to other algorithms.

Some algorithms exhibited low levels of recall and/or precision. The recall of Naïve Bayes and the precision of the ADWIN variant of Ozaboost fall sharply after a significant number of learning instances. Also, the recall of the ADWIN variant of Ozabag remains very low during the entire run.

6 Normalcy patterns classified as pre-failure patterns.
7 Pre-failure patterns classified as normalcy patterns.



(a) Relation of the number of false positives and false negatives with the number of failure scenarios used for learning.



(b) Relation of the number of true positives of each fault type with the number of failure scenarios used for learning. Misc faults are organic faults not injected intentionally.

Figure 76: Relation of the number for failure scenarios with prediction performance. Slashed lines represent zero values or small values not visible due to the scale of the plot.

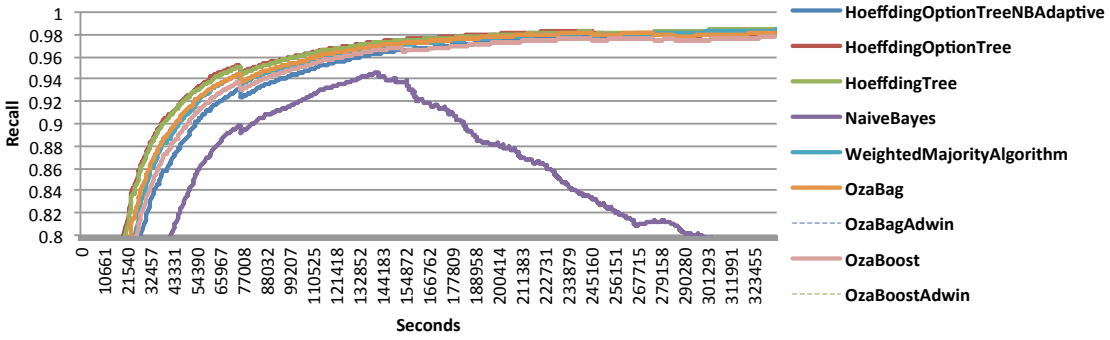
6.6.6 *Discussion of Results*

This section discusses the experimental results obtained in the same order they are presented in Section 6.6.5.

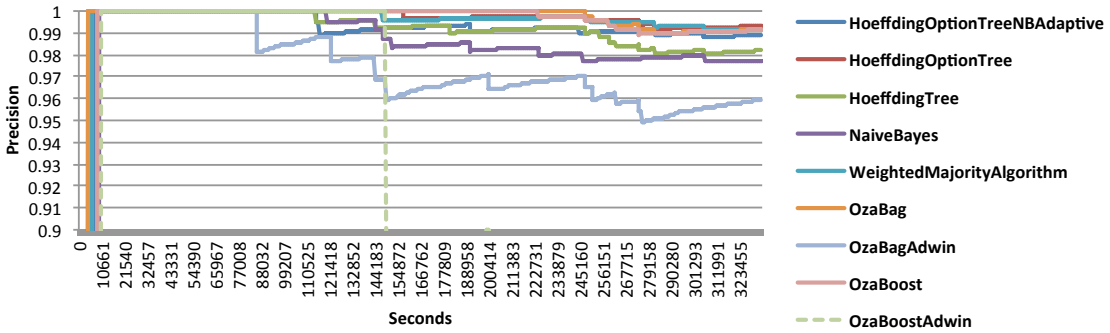
6.6.6.1 *Load Control*

SHStream implements a load control mechanism to ensure self-protection against overloading. Despite not incorporating the core of our research, ensuring self-protection against workload-related failures will guarantee that all performance failures are caused by performance anomalies.

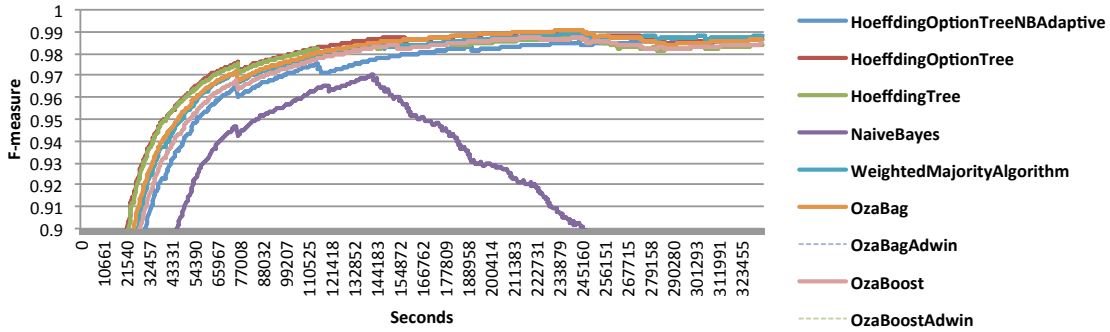
Experimental results validated the efficacy of SHStream in redirecting connection requests which, if accepted, would lead the server to an overloading state. The load control approach has the limitation of occasional unnecessary redirections when the server is close to its capacity limit, caused by the nature of the approach. Yet, we believe



(a) Recall of each classifier



(b) Precision of each classifier



(c) F-measure of each classifier

Figure 77: Failure prediction performance of each algorithm. Slashed lines represent zero values or small values not visible due to the scale of the plot.

that in a well-dimensioned distributed system with several video server instances, the sporadic redirection of connection requests to other servers would have a negligible impact on the overall service performance.

6.6.6.2 *Performance Overheads*

The performance overhead induced by running the SHStream application is an important issue of concern. Experimental analysis showed that SHStream builds accurate models iteratively with the arrival of new learning data and simultaneously performs classification of log instances without impacting the server nominal capacity.

6.6.6.3 *Impact of the Number of Failure Scenarios on Classification Performance*

Since models are trained iteratively, it is expected an increase of the prediction performance aligned with the number of failure scenarios available for learning. Experimental results showed that false negatives are consistent during the earlier stages of learning (up to 15 failure scenarios), but rare afterwards. That means that models are able to capture most of the pre-failure patterns, even when they are trained with a small number of learning instances.

As in any failure prediction approach, false positives typically lead to unnecessary execution of repair actions that waste system resources and can perturb the service quality perceived by users. SHStream showed absence of false positives during the earlier stages of learning. After stabilization of the prediction performance of models, the number of false positives remains low relatively to the number of log instances classified. On top of that, the low-cost repair actions proposed in this thesis can absorb the effect of these false positives on performance without user-visible failures.

6.6.6.4 *Breakdown of Prediction Performance into Fault Types*

The prediction performance of models is homogeneous across fault types — the CPU, memory and I/O faults showed equivalent prediction performance. This observation is valid for any number of failure scenarios used to train models.

6.6.6.5 *Breakdown of Prediction Performance into Learning Algorithms*

Ensemble algorithms exhibited the best performance. That means that the use of several Hoeffding Trees models provides a slight improvement over the prediction performance of a single model — the latter has approximately 1% less precision. The lower precision of single Hoeffding Trees models relatively to ensembles can be compensated by the high levels of interpretability provided by the former. Thus, single Hoeffding Trees models can be preferable when the analysis of models by human operators is a desirable requirement.

Probabilistic classifiers represented in SHStream by Naïve Bayes showed poor performance. This observation is coherent with the results obtained for TANs in the Pure Streaming infrastructure. The coherence of results is not surprising, since TANs appear as a natural extension to the Naïve Bayes that drops the conditional independence assumption. However, the poor results of Naïve Bayes can also be attributed to the viola-

tion of the conditional independence assumption demanded by this algorithm, which states that the features should be independent. As an example, the CPU usage parameters can be dependent on I/O activity parameters. Yet, notwithstanding this theoretical assumption of feature independence is often violated in practice, practical experience has shown that Naïve Bayes performs surprisingly well regardless [Rish 2001][Zhang 2004][Domingos and Pazzani 1997]. We believe that the resilience of decision trees and ensemble models to outliers is the main reason behind their good performance relative to probabilistic models, in both batch learning and online learning scenarios. As described before, outliers are noise patterns obtained in the delimitation of pre-failure patterns for normalcy patterns that can compromise the classification performance of models.

6.6.6.6 *Performance of Algorithms Extended with ADWIN*

Ensembles algorithms combined with ADWIN for change detection revealed poor classification performance. Surprisingly, the use of ADWIN in Adaptive Hoeffding Trees has no impact on the performance of the classifiers. Through the analysis of this algorithm, we noticed that the branches in the tree are only replaced when their accuracy decreases and the new branches demonstrated to be more accurate. This contrasts with the implementation of ADWIN in Ozaboost and Ozabag, which removes the worst classifier of the ensemble and adds a new classifier to the ensemble when a change is detected, even though the new classifier exhibits worse performance.

The most likely explanation for the poor performance of ADWIN is the heterogeneity of pre-failure patterns. Performance anomalies lead the server to unstable conditions, generating nonuniform combination of features values with low repeatability. The non-stationarity of data can force important data to be discarded.

Since performance anomalies are rare events that may have to be learned during long periods of time, care must be taken to avoid losing important knowledge useful for failure prediction. Adaptive Hoeffding Trees seem to be adequate to the characteristics of our problem, since the branches in the tree are only replaced by branches that improve the model performance. Another intuitive approach to handle the concept drift problem without losing important data is to segment the learning activity into periods delimited by external events manually set by human operators or captured automatically from the system — e.g., hardware changes and major software updates. Therefore, in case the classification performance drops below a specified level after the occurrence of a new event, a new model is trained from scratch.

In actual computer systems, the frequency of software updates and changes in the infrastructure is high. This constitutes a challenge to the maintenance of classification models. The reason is that they can take long periods of time to be trained from scratch. Benchmarks are solutions with high potential for rapid creation of models. They can be an effective solution to train models and rapidly exploit them for classification of

log instances. We believe that benchmarks similar to those used in our experimental work (presented in Chapter 4) constitute a relevant contribution to that purpose.

6.7 GLOBAL ANALYSIS OF FAILURE PREDICTION

Experimental results testify the viability of our approach for prediction of performance failures, in both HTTP Streaming and Pure Streaming infrastructures. Decision trees provide a superior performance over the other algorithms evaluated in both video-streaming infrastructures. Their performance is only surpassed by ensembles of models. We believe that the resilience to noise of these algorithms is the main reason behind their good results.

Notwithstanding online learning schemes allow continuous learning, both batch learning and online learning models have to been trained with a sufficient number of instances to provide a minimum level of accuracy. Benchmarks can give a fundamental contribute to train models from scratch the first time they are used and when the server application or infrastructure is changed or updated.

The failure prediction and failure repair activities have to be integrated. The look-ahead time provided by failure prediction allows the rescue of the server checkpoint in the server migration technique and the execution of any of the repair techniques before the occurrence of user visible failures. Also, there are negligible server down-times associated to the execution of repair techniques (server migration and reboot), as shown in Chapter 5. Yet, the absence of QoE degradation depends on the assumption that the server can be warmed-up appropriately.

QoE degradation during the server warm-up period is avoided when the reduction of the load in the primary server (faulty server), by reason of their transference to the secondary server (rebooted server) being warmed-up — alleviating the load pressure in the primary server — is sufficient to avoid failures until the end of the server warm-up period. Otherwise, the secondary server may has to replace the primary server and assume all load. In such circumstances, for high server load levels, some end-users may experience QoE degradation until the end of the server warm-up period.

False positives are an important issue in any failure prediction approach. Although the frequency of false positives is low in our infrastructure, it is important to consider their impact on the service quality. In our approach, false positives offer less risk of QoE degradation than true positives. The reason is that, the server warm-up process can be completed when the repair action is triggered by a false positive, since failures are unexpected in the meantime.

6.8 CHAPTER SUMMARY

This chapter presented and evaluated experimentally our failure prediction approach in the Pure Streaming and HTTP Streaming infrastructures. Pre-failure patterns are

in the core of this approach. They are formed not only by generic system metrics values, but also by process, network and application-level metrics values obtained before failure occurrence. Also, derived metrics are included to expose the tendencies of all metrics at each time interval.

We presented a failure prediction solution for Pure Streaming using batch learning algorithms and another for HTTP Streaming using online learning algorithms. The latter includes a segmentation method for separating normalcy patterns from pre-failure patterns during the learning process with small errors. Also, it includes a methodology to employ all learning algorithms available for training several models and for classification of log instances. Then, it selects, at each iteration, the model with the best performance until the current time, for deciding whether repair is needed or not.

The experimental results showed that, in both infrastructures, the performance failures can be predicted with small errors and with enough anticipation to execute the repair actions. Plus, it was observed that false positives are rare and their performance cost can be tolerated using cheap repair techniques, such as server migration and reboot techniques (Chapter 5).

The success of proactive recovery depends not only on the look-ahead time provided by failure prediction but also on the selection of appropriate repair techniques. The repair techniques should be selected according to the fault type diagnosed. Chapter 7 addresses the problem of diagnosing failures in video-streaming systems, through classification of failure patterns according to the respective type.

7

FAILURE DIAGNOSIS

Diagnosis strategies are classified in terms of time constraints as: *online diagnosis* and *offline diagnosis*. The diagnosis strategy adopted dictates the recovery strategy, as schematized in Figure 78.

In online diagnosis strategies, the failure diagnosis activity is performed within limited time bounds to minimize the recovery time. The diagnosis activity is triggered by the indication of failure given by the failure prediction or failure detection activities. *Proactive diagnosis* is performed online, after prediction of failures. Thus, the failure diagnosis process relies on data collected at the prediction time to classify forthcoming failures. *Reactive diagnosis* is also performed online, but after detection of failures. This diagnosis strategy is similar to proactive diagnosis, but relies on classification of failure patterns using data gathered after failure occurrence. Thus, differences in the accuracy of both diagnosis strategies are expected, since the log data collected at the time the failures are predicted are associated to system behaviors different from those expected after failure occurrence.

Figure 79 illustrates the workflow of proactive and reactive strategies executed during failure recovery, involving the failure prediction, failure diagnosis and repair activities. Failure prediction is followed by proactive diagnosis, for classification of predicted failures. Yet, reactive diagnosis is still required in case of failure misdiagnosis — leading to a repair action that is ineffective to mitigate the error — or unpredicted failures. In case the reactive diagnosis also presents errors, it is required manual intervention to recover the system.

The autonomic abilities of self-healing systems permit overcoming failures without or with minimum human intervention. However, self-healing techniques circumvent the failures but do not remove the faults in the origin of the failures. In other words, the errors are cleaned by repair actions but the faults remain dormant until their next activation. *Offline diagnosis* can support the maintenance of software to remove these faults.

Notwithstanding the fault removal process requires human intervention, it will benefit from automatic failure classification. Examples of classification outcomes are the separation of workload-related failures from performance anomalies, the identification of the fault location or the indication of the resource directly affected by the fault. Recognizing these failure patterns through human inspection of logged data is awkward and time consuming, in particular because they are specific to the system.

This chapter addresses the diagnosis of failures in video-streaming services. This self-healing activity complements the monitoring, repair and failure prediction activ-

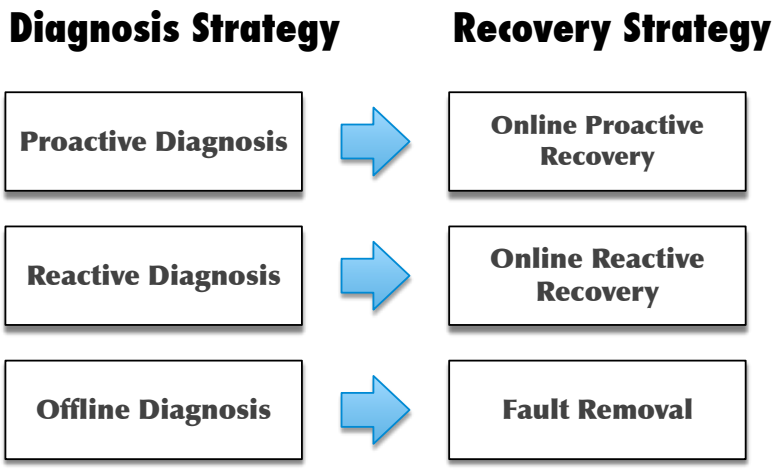


Figure 78: Diagnosis and recovery strategies.

ities with classification of failures detected or predicted into their respective types. Diagnosis outcomes enable the selection of the most appropriate corrective actions for the failures identified. The failure diagnosis methodology, algorithms and experimental results are presented in this chapter for the Pure Streaming and HTTP Streaming infrastructures.

7.1 FAILURE DIAGNOSIS APPROACH

Failure diagnosis in Pure Streaming is performed under the assumption of a traditional server infrastructure composed by a video-streaming server and one probing agent running in an independent machine. The Pure Streaming server has complex behaviors, due to the strict control of the RTP protocol over the data streamed within each session. The dynamic control of client-server data flows leads to variability of server behaviors and resources required during the lifespan of each session. Such complexity is propitious to error-prone load control mechanisms external to the server system, since the server resources consumed by client workloads are difficult to estimate. Hence, classification of failures into workload-related failures and performance anomalies is valuable for failure diagnosis, since both types of failures can occur.

The classification approach adopted for Pure Streaming services is less valuable in the HTTP Streaming infrastructure, since the simplicity of HTTP server behaviors enables the implementation of effective load control mechanisms, such as that proposed in Chapter 4 and evaluated in Chapter 6. Therefore, assuming that the mechanism for self-protection against workload-related failures implemented by the HTTP Streaming infrastructure is effective, all failures diagnosed are performance anomalies. Thus, only localization patterns are addressed in the diagnosis process.

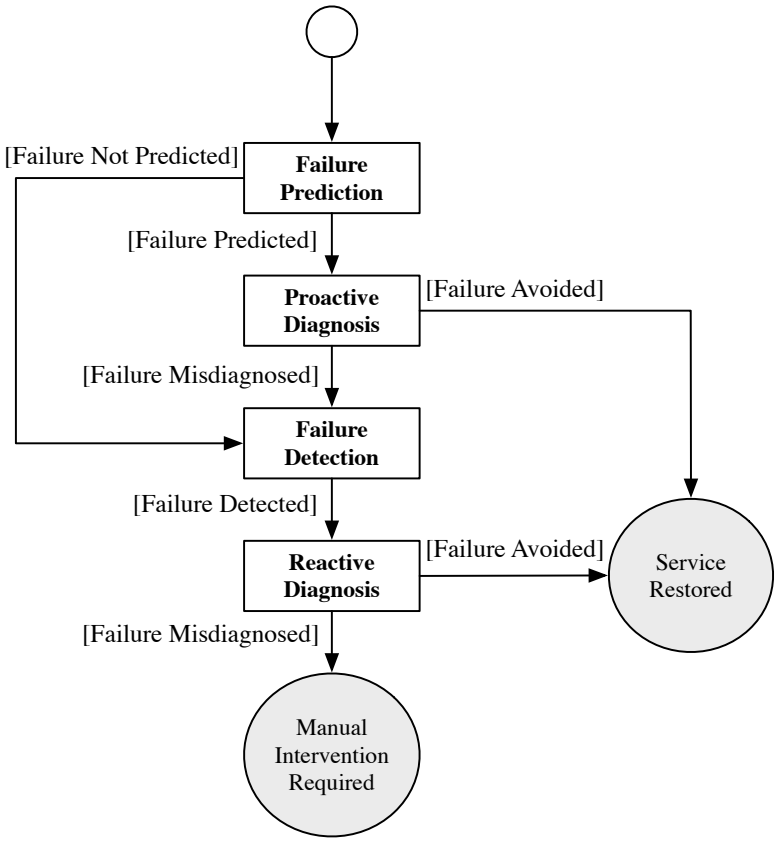


Figure 79: Failure recovery workflow using proactive and reactive strategies.

Contrasting with the Pure Streaming infrastructure, the HTTP Streaming infrastructure (SHStream) uses virtualization to isolate the video-streaming application from its runtime environment, for improved recoverability. Thus, in the HTTP Streaming infrastructure, the information about the origin of failures is valuable for selecting appropriate repair actions. It allows determining whether the errors are located in the virtual container where the server is running — requiring only intervention at the VC level — or otherwise, the problem is located in the system or network environment where the virtual container runs.

Failure prediction is presented in Chapter 6 as a pre-failure pattern detection problem. Pre-failure patterns represent system behaviors preceding failures, gathered in the form of system, application and network metrics/parameters values. Just as in the failure prediction approach, our failure diagnosis approach identifies error patterns interpreted as symptoms of failure being experienced (reactive diagnosis) or that might occur in the near future (proactive diagnosis). Error patterns are pre-failure patterns in reactive diagnosis and failure patterns in proactive diagnosis, which are interpreted for each possible classification outcome.

Classification of error patterns in the diagnosis activity is based on the breakdown of performance problems presented in Figure 17 (Chapter 4). However, we adapt that failure classification structure to be aligned with the repair actions designed for our self-healing infrastructures, as presented in Figure 80. Both pre-failure patterns and failure patterns are divided primarily into two main groups: *failure-type patterns* and *localization patterns*.

7.1.1 *Failure-Type Patterns*

The self-healing infrastructure for Pure Streaming services is not self-aware of the actual server load, avoiding any type of load control. For this reason, workload-related failures are likely to occur. Classification of failure-type patterns into workload-related failures — referenced as client-workload overloading — and performance anomalies is thus required.

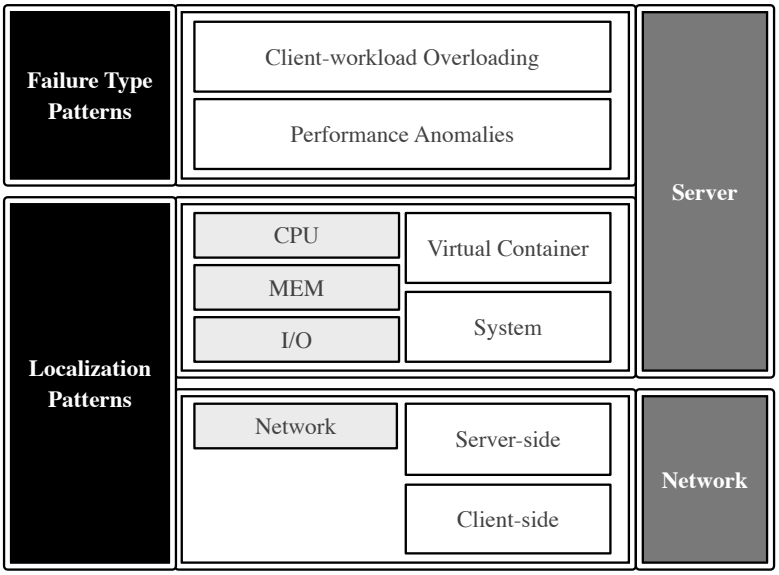


Figure 80: Classification structure of localization patterns and failure-type patterns.

In online recovery scenarios, the failure type determines the appropriate repair action. Workload-related failures are overcome by reducing the server load or readjusting allocated resources. On the other hand, performance anomalies may require restarting the server application or rebooting the operating system. In offline fault repair activities, the diagnosis outcome for failure-type patterns allows filtering log data regarding the: (1) software faults associated to performance anomalies; and (2) workload-related failures associated to an underprovisioned infrastructure or to a weak load control. These data will support corrective software maintenance.

One intuitive approach for discriminating performance anomalies from workload-related occurrences resides in determining whether the server performance is expected for the workload generated by client requests, using an approach similar to that presented in [Cherkasova and Staley 2003]. In that way, we would deduce that failures are caused by performance anomalies when the client workloads are insufficient to push the server load above the server capacity. Otherwise, the failures are caused by workload-related failures. However, despite the simplicity of this approach, it presents two problems:

1. **Specification and measurement of client workloads** — workload changes would perturb the evaluation of the impact of client requests in the server behavior, due to the impact of the temporal locality. For example, the server could handle n connections when 90% of them request the same video, but it could overload if the same percentage of connections request one video exclusively;

2. **Complex server behaviors** — Pure Streaming servers adapt dynamically the flow of data sent to end-users, according to the control information and statistics reported by clients regularly. That means that the consumption of resources varies according to several conditions. Network bandwidth variability, client buffer states and the users' interactions with the player (e.g., Play, Pause and Time Seeking) interfere with the server resources consumed during the lifecycle of the video-streaming request.

We adopt a failure classification approach based on identification of patterns in the log data. Thus, failure diagnosis is implemented by identifying recurrent pre-failure patterns — in proactive diagnosis — and failure patterns — in reactive diagnosis — specific to each failure type. This approach is independent of the workload type and can be used to model complex server behaviors.

7.1.2 *Localization Patterns*

The self-healing infrastructure for HTTP Streaming controls the system workload to avoid overloading scenarios. Thus, performance anomalies are considered the unique cause of failures. Anyway, the repair techniques executed should be specific to the failure location.

The problem of classification of localization patterns is akin to the problem of classification of failure-type patterns, despite the former has more possible classification outcomes. Figure 81 presents the workflow for diagnosis of performance anomalies through classification of localization patterns. Firstly, each log instance is classified in terms of the resource directly affected by the fault: CPU, memory, I/O or network. Then, when the classification outcome is not the network, it is classified by another classifier in terms of its location within the server's host.

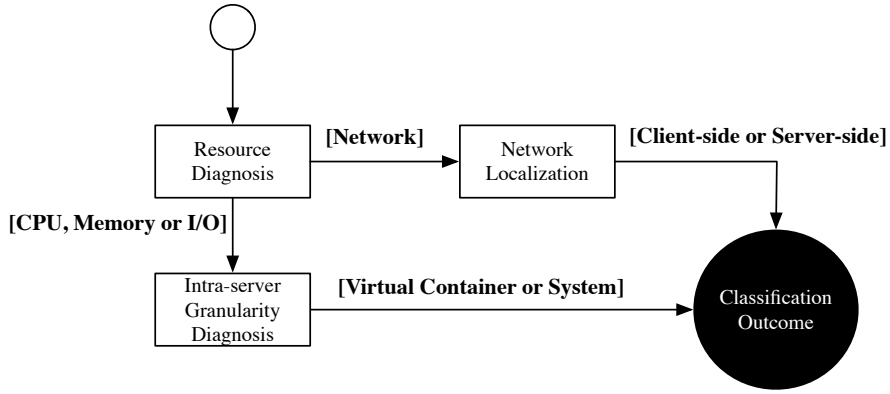


Figure 81: Workflow followed for failure classification.

7.1.2.1 *Server Failures*

The location of failures classified as CPU, memory or I/O is classified again as follows:

- **Virtual container failure** — when the server application process running in the virtual container is responsible for the failure;
- **System failure** — occurring at the operating system level, outside the virtual container.

Process-level metrics/parameters (e.g., CPU usage of the server application's process) are important features for creation of classification models. This is because localization patterns can consider not only application performance and global system features, but also process-level features that scrutinize the impact of the server application on the system performance. Therefore, it is possible to discriminate patterns inherent to errors in the video application process from patterns inherent to operating system errors.

7.1.2.2 *Failure Classification for Fault Removal Activities*

Although the main focus of this chapter is on online diagnosis, our failure classification scheme provides valuable information for software maintenance activities. All failure classes except those associated to system resources (CPU, memory, I/O) have a corresponding repair action in the self-healing lifecycle. However, the classification of system resources can provide the identification of faults through their reflection on resources, such as:

- **Infinite loop faults** (e.g., unterminated CPU intensive threads) — manifests as CPU exhaustion, when computation is performed inside the loop; manifests as I/O exhaustion, when there are disk accesses inside de loop;

- **Memory leaks** — memory allocated dynamically is not deallocated appropriately, leading to memory exhaustion;
- Any other faults causing disequilibrium between client workloads and system behavior or leading to other anomalous system behaviors.

Since the classification of system resources provides information about the resource directly prejudiced by the error, it could help pinpointing the root cause of the failure during fault removal activities. Yet, the development of this research topic is out of the scope of this thesis.

7.1.2.3 *Network Failures*

When the failure is classified as a network problem, it is necessary to determine whether it is located server-side or client-side. We use a simple approach to determine the localization of the network problem based on the analysis of coherence of failures between requests. When the service degradation is uniformly observed among all requests handled by the server, the failure will likely be localized server-side. On the other hand, when a small subset of requests is experiencing service quality degradation, the network failures will likely be localized client-side.

The threshold over the number of faulty requests used for separation of client-side network failures from server-side network failures has to be specified — empirically by an expert or through analysis of historical data for each service.

7.1.3 *Mapping Failure Classes to Repair Actions*

The classification of failure-type patterns and localization patterns will decide the repair action to apply. The correspondence between the classification outcomes and their repair actions is illustrated in Figure 17 (Chapter 4).

Systems that implement load control mechanisms can anticipate the acceptance of new requests by the server and refuse or redirect them to other server instances. Otherwise, in case of server overloading, workload-related failures can be overcome by:

- Migrating the server to a host better provided with resources;
- Renegotiating more server resources;
- Reducing the server load (e.g., terminating part of the requests being handled).

Performance anomalies lead the server application or the underlying operating system to a nondeterministic condition. The recovery from these anomalous conditions is attempted using the techniques presented in Chapter 5, by restoring the faulty context (server application or the operating system) to normality, using one of the following techniques:

- **Reboot the server's machine** (the host) — when the fault originates outside the virtual container where the video server application is running. This repair action is preceded by the migration of the server application to another host;
- **Reboot the virtual container** — when the fault originates inside the virtual container where the video server application is running;
- **Migrate the virtual container to another host** — when the fault originates outside the virtual container or in the server-side network.

The selection of appropriate repair actions involves the diagnosis of localization patterns to determine whether errors are located inside the virtual container, outside the virtual container or otherwise, has originated in the network. When the errors are located outside the virtual container or in the server-side network, the virtual container is free of errors and thus, can be rescued to another host. Therefore, the server application state and client-server connections are maintained intact after recovery. Otherwise, when errors originate within the virtual container, it has to be restarted — in the same or in another host.

7.2 CLASSIFICATION OF FAILURE PATTERNS IN PURE STREAMING

We use the following methodology for training models and classifying log instances in proactive failure diagnosis:

1. Create regression models to forecast feature values from prediction time to failure time;
2. Create classification models using batch learning algorithms, for classification of failure patterns (feature values observed after failure occurrence);
3. For each predicted failure, it is initiated a two-step classification process of the respective log instance. Firstly, the feature values captured at the prediction time are used to forecast the feature values to the failure time, using the regression models. Secondly, the feature values forecasted are used by the classification models to diagnose the failure.

We explore a classification approach for failure diagnosis based on forecasted failure patterns. Instead of classifying pre-failure patterns to diagnose predicted failures, the feature values are forecasted from the failure prediction time to the failure occurrence time before being classified.

7.2.1 *Research Questions*

This chapter addresses the following research questions relative to the diagnosis activity in the Pure Streaming infrastructure:

1. What is the diagnosis performance obtained for classification of failure patterns into workload-related failures and performance anomalies?
2. What is the error introduced by the forecasting of failure patterns from the time each failure is predicted to the time it is detected?
3. What is the performance difference between proactive diagnosis and reactive diagnosis?
4. Are the classification performance resilient to workload changes?

The failure diagnosis approach presented in this chapter is evaluated experimentally to answer these research questions.

7.2.2 *Formalization of the Failure Diagnosis Problem*

The core of our diagnosis approach is based on the hypothesis that each failure cause has specific signatures (failure patterns), represented by specific combinations of feature values. So, the identification of failure types is based on the discrimination of failure patterns associated to workload-related failures from failure patterns associated to performance anomalies.

In reactive diagnosis, the forecasting of feature values is unnecessary because failures are detected after their occurrence. Thus, the failure diagnosis problem is formalized as follows. Being F_t one of the failure states $F = \{\text{overloading, anomaly}\}$ at time t and M_t the vector of values for n relevant features at time t, the problem resumes in learning a classifier that maps the space of possible values of M to failure states F. Models are trained using logs in the form $< M_t, F_t >$ collected from the system during failure periods.

In proactive diagnosis, the feature values are unknown at prediction time. Hence, each feature value m_{t+i}, with $m \in M$, is forecasted individually through linear regression [Han et al. 2006] from a set of values $S_{m,t}$. S_m is a subset of all features gathered by logs considered relevant for forecasting the value of each feature m, t is the time when the feature value is forecasted and i the look-ahead time of prediction. $S_m \neq M$ is expected, since S_m contains the features used by the models to forecast the value of each feature value m used for classification, while M contains the features relevant for classification.

Linear regression can be *single-variate* or *multi-variate*. Figure 82 illustrates the forecasting of the CPU parameter using single-variate linear regression. This technique

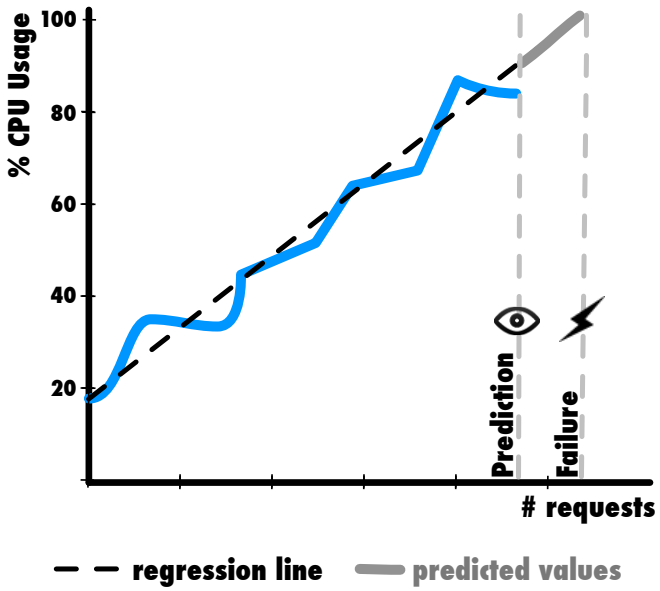


Figure 82: Prediction of the CPU usage parameter to the failure time.

uses a single independent variable ($|S| = 1$) to determine the value of the dependent variable. In this example, the number of requests is the independent variable used to forecast the CPU parameter. Contrasting with single-variate regression, multi-variate regression uses several independent variables to predict the value of the dependent variable. That means that each regression model R_m forecasts the value of each feature m_{t+i} from $S_{m,t}$, being $|S| \geqslant 1$.

We use multi-variate regression for prediction of individual feature values. This approach requires a feature selection approach for selection of the relevant features that will be used to forecast each feature value.

7.2.3 *Algorithms*

Failure diagnosis relies on two types of algorithms for creation of models: *classification algorithms* and *regression algorithms*.

7.2.3.1 *Classification Algorithms*

Both proactive diagnosis and reactive diagnosis approaches use batch learning algorithms — similar to those used for failure prediction — to create diagnosis models. We use two classes of learning algorithms to build classification models: probabilistic algorithms and decision trees algorithms.

The Naïve Bayes and the C4.5 algorithms are applied to creation of probabilistic models and decision trees, respectively. These two traditional algorithms have been

proven effective in modeling patterns for a broad type of problems. The C4.5 algorithm, in particular, also exhibited the best performance of all algorithms explored in failure prediction (Chapter 6). The Naïve Bayes and C4.5 algorithms are presented in Section 6.5.2.

7.2.3.2 *Regression Algorithms*

The multivariate regression models used to forecast feature values have the form of (21), being Y the data matrix containing the response variables (forecasted features), X the matrix of feature values, β the matrix of factor coefficients (model parameters) and ϵ the matrix of noise terms.

$$Y = X\beta + \epsilon \tag{21}$$

Statistical estimation and inference in linear regression focuses on β, the matrix of regression coefficients. This parameter is obtained by the *least-squares fit* method [Björck 1996]. This method finds the model parameters best fitting the data. It finds its optimum when the sum of the squared residuals U, calculated as in (22), is a minimum.

$$U = \sum_{i=1}^{n} r_i^2 \tag{22}$$

Each residual r_i represents the difference between the value observed for the dependent variable y_i and the value forecasted by the regression model, as formulated in (23).

$$r_i = y_i - X_i\beta \tag{23}$$

The forecasting of feature values using regression models introduces errors that will sum up to the classification errors generated by diagnosis models. Thus, quantification of regression errors is necessary for error analysis, with the purpose of determining the impact that the forecasting of features has on the global diagnosis performance.

7.2.4 *Evaluation of Diagnosis Models*

Errors observed in the diagnosis process can be broken down into the errors resulting from: (1) failure prediction; (2) the forecast of feature values; and (3) the classification of failure patterns.

False positives in failure prediction result in arbitrary and useless diagnosis, since the failure classification is performed for log data associated to non-faulty system conditions. For that reason, we remove all scenarios containing failure prediction errors from the evaluation of the diagnosis performance.

7.2.4.1 *Errors Resulting from the Forecasting of Feature Values*

Forecasting errors are quantified by measuring the difference between the feature values forecasted through regression and the feature values observed after failure occurrence.

The Root-Mean Square Error (RMSE) presented in (24) determines the average magnitude of the regression error. It is calculated by the difference between the values forecasted by the model p_i and the values observed from measurements a_i.

$$\text{Root Mean-Squared Error} = \sqrt{\frac{\sum_{i=1}^{n}(p_i - a_i)^2}{n}} \tag{24}$$

The RMSE metric provides a diagnosis error with limited interpretation. That means that there is no basis for separating acceptable from unacceptable RMSE values. Yet, as usually the RMSE has the same unit as the dependent variable, it could be empirically evaluated with respect to that unit. As an example, for a datum which ranges from 0 to 100 (e.g., CPU usage in percentage), a RMSE of 5 can be considered small but a RMSE of 50 is undoubtedly large.

RMSE is tied to single variate analysis, because it is scaled to the variable being analyzed. When the error analysis involves several variables, the RMSE value of each variable should be normalized to allow comparison of the RMSE values of variables with different scales. The Normalized Root-Mean Square Error (NRMSE) (25) is used for that purpose. It divides the RMSE by the range of the values observed for the forecast variable.

$$\text{NRMSE} = \frac{\text{RMSE}}{x_{max} - x_{min}} \tag{25}$$

Since the values of the NRMSE are normalized, the global error resulting from the average of the NRMSE of all variables can be calculated.

7.2.4.2 *Classification Errors*

Performance evaluation of classification models relies on metrics presented in Section 6.3.1: *precision*, *recall* and *f-measure*. These metrics quantify the ability of classification models to discriminate failure patterns.

In failure diagnosis, the performance metrics are calculated for each classification outcome: workload-related failure or performance anomaly. This requirement contrasts with the classification performance evaluation in failure prediction. In failure prediction problems, the normality condition is a neutral state and consequently, the failure condition is the unique class that deserves attention. To exemplify, the number

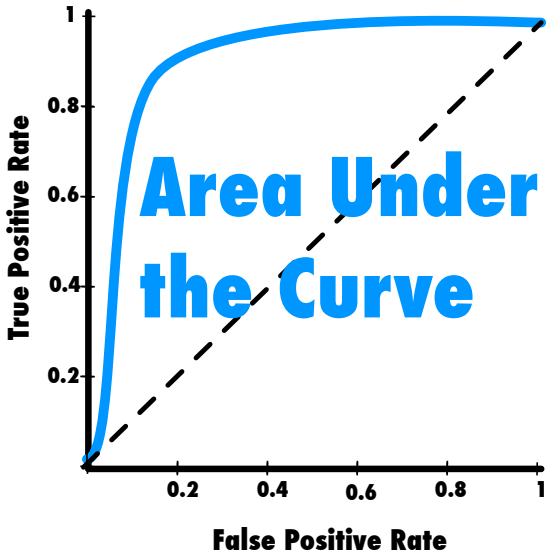


Figure 83: Area under a ROC curve. The classification performance improves as the AUC approaches to 1 and worsen towards 0.

of true positives represent the number of failures classified as such. In failure diagnosis, the same metric is interpreted according to the class being considered — the true positives of workload-related failures or the true positives of performance anomalies.

Failure classification pursues the largest number of true positives with the smaller number of false positives, for each class. This is usually subject to a tradeoff between true positives and false positives, as usually the increase of one of these metrics leads to the decrease of the other. The Receiver Operating Characteristic (ROC) curve, illustrated in Figure 83, allows an analysis of the tradeoff between the true positive rate and the false positive rate. The Area Under Curve (AUC) of the ROC resumes into a single value the relationship between both rates, which is representative of the classification performance of models. The classification performance improves as AUC approaches the 1 and worsen towards 0. Perfect performance is achieved when the AUC equals 1 because it represents the maximum area, attained when the number of false positives and false negatives, for each class, is zero.

We use the AUC as a complementary metric for the classification performance. This metric is more adequate to balanced datasets [Davis and Goadrich 2006], such as those used for failure diagnosis — the number of log instances is similar in both failure classes.

7.2.5 *Generalization of Models Between Workload Types*

Popularity of videos is one workload factor difficult to specify and measure, but impacts the server behavior significantly, since it interferes with the temporal locality of data, as explained in Section 4.5.2.1. Thus, the server would exhibit multimodal behavior for a specific combination of the number and type of videos requested, when the workload type changes in the vector of popularity. Thus, it is important to evaluate the resilience of the classification performance when the popularity of videos changes.

In addition to the ten-fold cross validation, we evaluate the models' classification performance when the models are trained with a workload of reference and are evaluated with workloads that drift the popularity of videos into two opposite popularity directions: pure popular and pure unpopular. This evaluation scheme is denoted in this Chapter as *inter-benchmark validation*. The results of the inter-benchmark validation reflect the generalization ability of models in terms of popularity of videos.

7.2.6 *Experimental Results*

This section presents the results of the experimental evaluation of our approach in reactive diagnosis and proactive diagnosis scenarios.

7.2.6.1 *Feature Selection*

Feature selection is an important activity for improving the classification performance of models created using algorithms that do not select relevant features intrinsically — Naïve Bayes benefits from feature selection but C4.5 trees select relevant features intrinsically. This topic is developed in Section 6.4.2. Each relevant feature identified will be further subjected to forecasting by regression models, to obtain the input values of the failure classifiers.

We present the results of feature selection in two phases:

1. Analysis of the discriminative power of each feature;
2. Selection of relevant combination of metrics using the Linear Forward Selection algorithm.

Figure 84 presents the f-measure calculated for a subset of metrics, used individually to create decision trees models with the C4.5 algorithm. It shows that the *memory* parameter (memory usage) presents the highest discriminatory power for performance anomalies, reaching 98%, followed by the *Mem PID* (the memory consumed by the server application's process) with 50%, the *Resident Set Size* (the non-swapped physical memory used by application process) with 50%, and the *VM PID* (virtual memory used by the application process) with 49%.

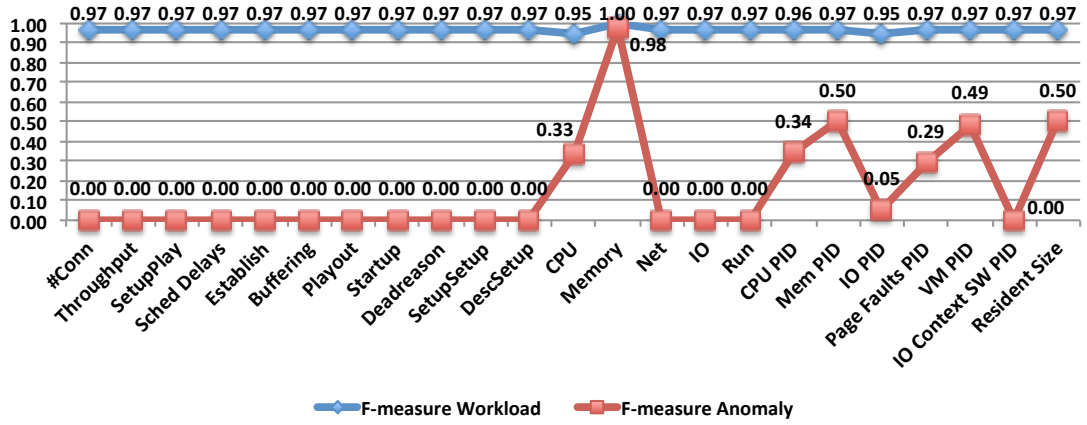


Figure 84: Classification performance of diagnosis using each feature. Features not shown have a f-measure of zero for performance anomalies.

By running the Linear Forward Selection algorithm, the combination of the *Session Establishment Time* and the *Time Between Two Setups* were also chosen to complement the aforementioned list of metrics. That means that these two features have low discriminative power alone, but can improve the model discriminative power when combined with other features.

Even though the analysis of the discriminatory power of features is valuable to understand the experimental results, the feature selection activity is performed for each system configuration dynamically. Thus, the list of features identified is not generalizable to other system configurations.

7.2.6.2 *Experimental Results for Reactive Diagnosis*

This section presents the results of the experimental work undertaken to evaluate the models' classification performance in reactive diagnosis scenarios. The experimental process has the following steps:

1. **Run the mix+spike and mix+anomaly benchmarks** (described in Section 4.5.3) to collect log data;
2. **Perform feature selection** to select metrics with discriminative power to create classification models (required only for the Naïve Bayes classifier);
3. **Build models and evaluate** their classification performance using ten-fold validation;
4. **Run the popular+spike, unpopular+spike, popular+anomaly and unpopular+anomaly benchmarks** to collect log data;

5. **Evaluate the models' performance** with the log data previously collected, using inter-benchmark validation.

Figure 85a and Table 5a present the evaluation results of the Naïve Bayes classifier, obtained through ten-fold validation and inter-benchmark validation. Similarly, Figure 85b and Table 5b present equivalent results for the J48 implementation of the C4.5 algorithm. The results of ten-fold cross validation are represented in the bar graph labeled as *10-Fold Workload* and *10-Fold Anomaly*, for workload-related failures (client-workload overloading) and performance anomalies, respectively. The remaining labels show equivalent results for inter-benchmark validation, obtained from the classification of failure patterns when the client workload drifts to pure popular and pure unpopular.

From the analysis of results we observe that, notwithstanding both models have high classification performance, C4.5 models outperform the Naïve Bayes models counterparts in all configurations. Using ten-fold validation, the C4.5 reaches 100% of recall and precision for both workload-related failures and performance anomalies. For the same validation scheme, Naïve Bayes has lower values, with 99.7% of recall for workload-related failures and 95.3% of precision for performance anomalies.

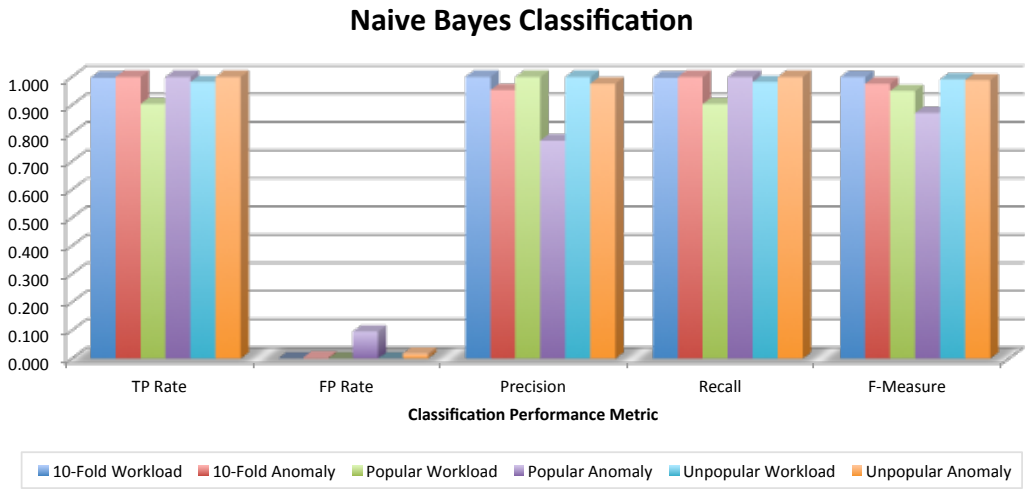
The recall and precision of C4.5 are 100% when the workload drifts to pure unpopular, for all failure causes. These values decrease to 93.6% of recall for workload-related failures and 83.7% of precision for failures caused by performance anomalies, when the workload drifts to pure popular. Accordingly, the ROC area (AUC) and the NRMSE are 96.8% and 0.22, respectively, for pure popular workloads.

The Naïve Bayes has lower classification performance than C4.5 trees in both pure popular and pure unpopular configurations. In the pure popular configuration, the recall of workload-related failures is 90.4% and the precision of performance anomalies observed is 77.4%. In the pure unpopular configuration, the recall of workload-related failures is 98.2% and the precision of performance anomalies is 97.6%. All other metrics values of Naïve Bayes in pure popular and pure unpopular configurations are 100%. Additionally, the ROC area (AUC) is 99.4% and 96.8% in the pure popular configuration for workload-related failures and performance anomalies, respectively. The same metrics achieved 100% in the pure popular configuration. Additionally, the NRMSE is 0.269 and 0.092 in the pure popular and pure unpopular configurations, respectively.

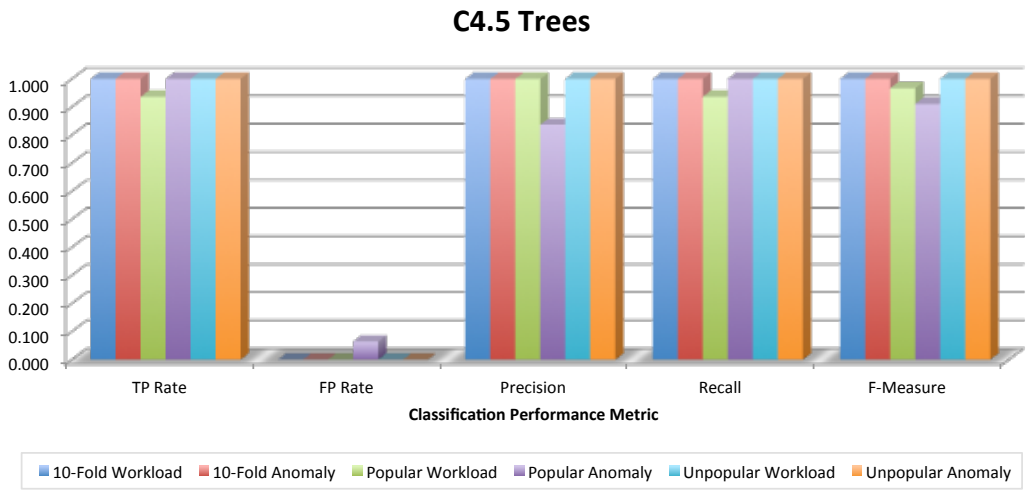
7.2.6.3 *Experimental Results for Proactive Diagnosis*

Proactive diagnosis faces an additional challenge when compared with reactive diagnosis of failures. It should deal with the problem of forecasting feature values from the prediction time to the failure time, without increasing the global diagnosis error significantly.

This section presents the results of the experimental work undertaken to evaluate the models' classification performance in proactive diagnosis scenarios. The experi-



(a) Per-class results of the Naïve Bayes



(b) Per-class results of C4.5 Trees

Figure 85: Classification performance of Naïve Bayes and C4.5 Trees using ten-fold validation and inter-benchmark validation. Inter-benchmark validation evaluates the classification performance when the workload changes to pure popular and pure unpopular.

mental process followed to obtain the evaluation results extends the equivalent process presented for reactive diagnosis, by adding the step of forecasting the feature values used for classification of predicted failures. The results are presented only for C4.5 Trees, due to their superior performance in reactive diagnosis. Also, the experimental results discard failure prediction errors (addressed in Chapter 6). To that end, all log

Naive Bayes			TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Root Mean Squared Error
10-Fold Cross-validation	Class	Workload	0.997	0.000	1.000	0.997	0.999	1.000	0.044
		Anomalies	1.000	0.003	0.953	1.000	0.976	1.000	
	Weighted Avg.		0.997	0.000	0.997	0.997	0.997	1.000	
Popular Workload (Validation)	Class	Workload	0.904	0.000	1.000	0.904	0.950	0.994	0.269
		Anomalies	1.000	0.096	0.774	1.000	0.872	0.968	
	Weighted Avg.		0.928	0.024	0.944	0.928	0.931	0.987	
Unpopular Workload (Validation)	Class	Workload	0.982	0.000	1.000	0.982	0.991	1.000	0.092
		Anomalies	1.000	0.018	0.976	1.000	0.988	1.000	
	Weighted Avg.		0.990	0.007	0.990	0.990	0.990	1.000	

(a) Naïve Bayes

J48			TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Root Mean Squared Error
10-Fold Cross-validation	Class	Workload	1.000	0.000	1.000	1.000	1.000	1.000	0.002
		Anomalies	1.000	0.000	1.000	1.000	1.000	1.000	
	Weighted Avg.		1.000	0.000	1.000	1.000	1.000	1.000	
Popular Workload (Validation)	Class	Workload	0.936	0.000	1.000	0.936	0.967	0.968	0.220
		Anomalies	1.000	0.064	0.837	1.000	0.911	0.968	
	Weighted Avg.		0.952	0.016	0.960	0.952	0.953	0.968	
Unpopular Workload (Validation)	Class	Workload	1.000	0.000	1.000	1.000	1.000	1.000	0.000
		Anomalies	1.000	0.000	1.000	1.000	1.000	1.000	
	Weighted Avg.		1.000	0.000	1.000	1.000	1.000	1.000	

(b) C4.5 (J48)

Table 5: Tables with the classification performance of Naïve Bayes and C4.5 Trees, using ten-fold validation and inter-benchmark validation. Inter-benchmark validation evaluates the classification performance when the workload changes to pure popular and pure unpopular.

instances mispredicted by the failure prediction activity are removed from the datasets previously to the failure classification.

The forecasting errors introduced by multivariate linear regression are measured by NRMSE. This metric is calculated for each forecasted feature, and the resultant values are further averaged. Figure 86 shows that the average NRMSE is small. NRMSE is below 6% (the values presented are absolute, but normalized to 100), in average, even for anticipation times of 20 seconds. Results for larger look-ahead times are ignored, since the failure prediction performance for larger anticipation times is small, as shown in Chapter 6.

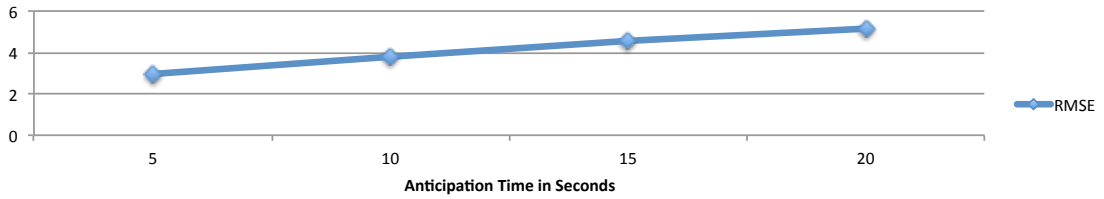


Figure 86: Average NRMSE of regression, calculated for all metrics. RMSE values are normalized to 100, according to the range of observed values, for each variable considered.

Even though the NRMSE is an important metric to evaluate the forecasting error, it is unable to measure the error's impact on the classification performance. This latter can be calculated through comparison of the classification errors obtained for reactive diagnosis with the classification errors obtained for proactive diagnosis.

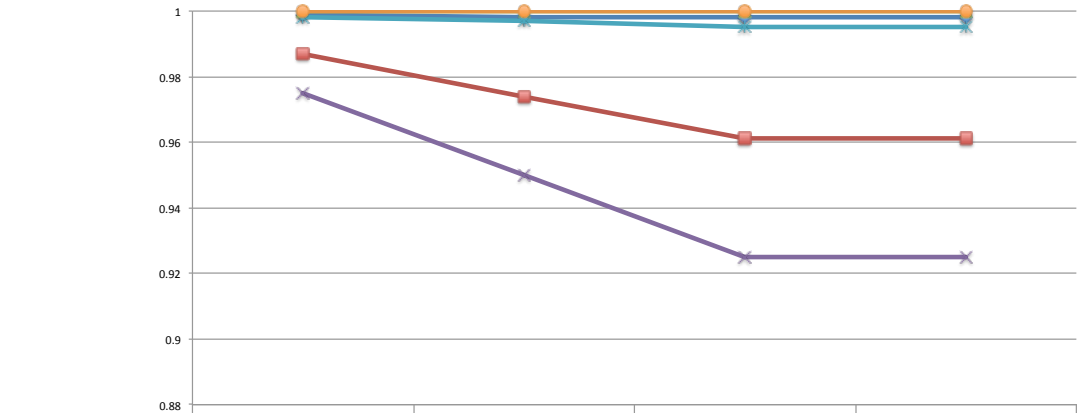
Figure 87 presents the precision, recall and f-measure metrics values obtained for proactive diagnosis. These results respect the ten-fold validation of C4.5 models, considering look-ahead times ranging from 5 to 20 seconds. It is noticeable a small decrease of the recall and precision values within the entire look-ahead time interval. The recall maintains at 100% for workload-related failures, similarly to precision values for performance anomalies. The recall values range from 97.5% to 92.5% for performance anomalies within the look-ahead time interval. The precision is stable for workload-related failures within the look-ahead time interval, ranging from 99.8% to 99.5%.

Figure 88 presents the classification performance loss of proactive diagnosis when compared with reactive diagnosis. For performance anomalies, the recall drops between 2.5% — for look-ahead times of 5 seconds — and 7.5% — for look-ahead times above 10 seconds. For workload-related failures, the precision suffers a loss between 0.2% — for look-ahead times of 5 seconds — and 0.5% — for look-ahead times above 10 seconds.

7.2.7 *Discussion of Results*

The experimental results previously presented provide answers to the research questions stated early in Section 7.2.1. They cover the classification performance of algorithms, the resilience of the models' performance to workload changes and the performance difference between proactive diagnosis and reactive diagnosis.

The experimental results show that C4.5 trees and Naïve Bayes models provide high classification performance in our failure diagnosis approach. In reactive diagnosis, the C4.5 tree models achieved perfect recall and precision using the workload of reference, outperforming the Naïve Bayes classifier. The higher performance of decision trees relatively to other algorithms is coherent with the failure prediction results.



	5	10	15	20
F-measure Workload	0.999	0.998	0.998	0.998
F-measure Anomalies	0.987	0.974	0.961	0.961
Recall Workload	1	1	1	1
Recall Anomalies	0.975	0.95	0.925	0.925
Precision Workload	0.998	0.997	0.995	0.995
Precision Anomalies	1	1	1	1

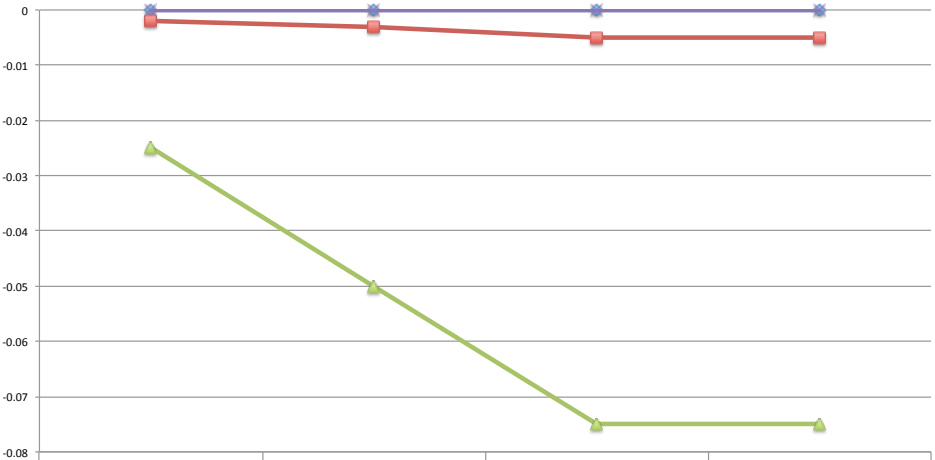
Anticipation Time in Seconds

Figure 87: Failure classification performance for workload-related failures and performance anomalies in proactive diagnosis.

The classification models were firstly evaluated with the training dataset. Afterwards, the same models were evaluated with log data obtained using benchmarks that bring the workload popularity to its extremes, to evaluate the resilience of classification models to workload changes.

Experimental results show that the classification performance loss due to workload changes is noticeable only when the workload of reference drifts to the pure popular workload (all requests target the same video). In these scenarios, the precision of performance anomalies fall significantly, leading to the inadequate execution of the repair actions required to performance anomalies, in replacement of those required for workload-related failures. Fortunately, this is an extreme workload corresponding to a worst-case scenario unexpected in production systems. Even in cache servers (e.g., proxy caches), where the distribution of requests between video files is more compact than in backend streaming servers [Wang et al. 2002][Zink et al. 2008], it is expected the distribution of client requests over several video files. Thus, the classification performance losses caused by workload changes are expected lower than those presented.

Our proactive diagnosis approach performs failure classification of feature values forecasted from the failure prediction time to the failure occurrence time. The resulting forecasting errors are responsible for the performance difference between proactive diagnosis and reactive diagnosis. Experimental results show that the forecasting errors measured both over feature values and classification performance are small. The precision, in particular, is similar in both diagnosis strategies.



	5 Seconds	10 Seconds	15 Seconds	20 Seconds
Recall Workload	0	0	0	0
Precision Workload	-0.002	-0.003	-0.005	-0.005
Recall Anomalies	-0.025	-0.05	-0.075	-0.075
Precision Anomalies	0	0	0	0

Figure 88: Performance loss in proactive diagnosis relatively to reactive diagnosis.

An overall analysis of the experimental results obtained for diagnosis of performance failures in the Pure Streaming infrastructure reveals an extremely high classification performance of failure types. These results are maintained even when the client workloads change significantly. The classification performance provided by machine learning algorithms is higher in the failure diagnosis than in failure prediction. This is expected, due to the lack of segmentation errors in the log data used for failure diagnosis problems. That means that all log instances classified correspond to effective failures. By contrast, in failure prediction problems the pre-failure patterns can be confounded with normalcy patterns, introducing additional errors.

7.3 CLASSIFICATION OF FAILURE PATTERNS IN HTTP STREAMING

We use the following methodology for learning and classifying log instances in failure diagnosis of HTTP Streaming systems:

1. Each log instance predicted or detected is used to train and evaluate resource diagnosis models, simultaneously with its classification into: *network failures*, *CPU*, *memory* or *I/O*;

2. Each log instance previously classified either as CPU, memory or I/O, is used to train and evaluate intra-server diagnosis models, simultaneously with its classification into: *virtual container* or *system*;

3. Each log instance previously classified as network failure is classified into: *client-side* or *server-side*.

The former two classification problems require complex models of the system, application and network states, demanding experimental evaluation. Conversely, classification of network failures relies on a threshold value for the percentage of failing requests, with the purpose of delimiting client-side failures from server-side failures. Since this classification approach depends on a simple threshold set specifically for each service, its experimental evaluation is unnecessary.

7.3.1 *Research Questions*

This section addresses the following research questions relative to the diagnosis activity in the HTTP Streaming infrastructure:

1. What is the performance of classification models, trained with online learning algorithms, in discriminating failure patterns into each of the localization classes?
2. What is the performance difference between classification of failure patterns and classification of pre-failure patterns exposed by log instances indicated by the failure prediction activity?
3. Which online learning algorithms exhibit the best diagnosis performance?
4. How many learning instances are required until the stabilization of the classification performance, using online learning algorithms?

The failure diagnosis approach presented in this chapter is evaluated experimentally for HTTP Streaming, to answer these research questions.

7.3.2 *Formalization of Diagnosis of Localization Patterns*

The Pure Streaming infrastructure approaches proactive diagnosis as a classification problem over feature values forecasted from prediction time to failure time. By contrast, proactive diagnosis in the HTTP Streaming infrastructure is implemented through classification of pre-failure patterns. This approach is rooted in the hypothesis of the existence of singularities inherent to each resource and intra-server location in the log data gathered at the prediction time.

The insight for using feature values respecting the failure prediction time for failure diagnosis is described as follows. One failure is predicted due to singularities in the log data arising after fault activation. Singularities are captured in the form of patterns more likely to observe in the course of a pre-failure period than during error-free periods. Hence, it is also reasonable to consider the hypothesis that the singularities

used for failure prediction are different for each resource and intra-server location and thus, can be exploited to classify failures in the diagnosis process. This hypothesis is formalized as:

> Being M the vector of feature values relative to application, system (global and process) and network parameters/metrics, the resource directly affected by the failure (CPU, memory, I/O or network) and the location of the failure within the server machine (inside or outside the virtual container) can be determined from patterns represented by the combination of values attributed to relevant features $\overrightarrow{M} \subseteq M$.

The formalization of the diagnosis problem for localization patterns includes both the resource patterns and the intra-server location patterns.

The problem of diagnosing resources can be formalized as follows. Being $R = \{\text{CPU, memory, I/O, network}\}$ the list of resources admissible for classification and M_t the vector of values for n relevant metrics of a given log instance at time t, the resource classification problem is defined as the mapping of M_t to one resource $r \in R$. Since t represents the prediction time, M_t represents feature values at prediction time, which indicates a failure at time $t + i$ in the future.

When the resource r is not the network, the classification outcome l gives the failure location within the server, being $l \in L$ and $L = \{\text{virtual container, system}\}$. The outcome l is determined from M_t, similarly to r.

7.3.3 *Implementation of Failure Diagnosis*

The online learning algorithms presented in Section 6.6.2 for failure prediction are also used for building classification models for failure diagnosis.

Similarly to the failure prediction activity, the diagnosis activity performs classification, learning and evaluation of models, but always with log data gathered by exercising the server with synthetic workloads and fault loads. The reason is that in failure diagnosis, the log data cannot be labeled automatically with the corresponding class before being used to train classification models. Figure 89 illustrates that problem.

When one recovery action is executed as a consequence of a predicted failure, the occurrence or the absence of a posterior failure is unable to evidence the diagnosis correctness. There are several motives associated to the success or failure of a recovery intervention. False positives observed in failure prediction, ineffective repair actions and other conditions leading to errors on any stage of the recovery process, avoids the analysis of the diagnosis correctness based on the effectiveness of the whole recovery cycle. Hence, log instances have to be labelled with the respective diagnosis types manually, which is time consuming in offline learning and unfeasible in online learning.

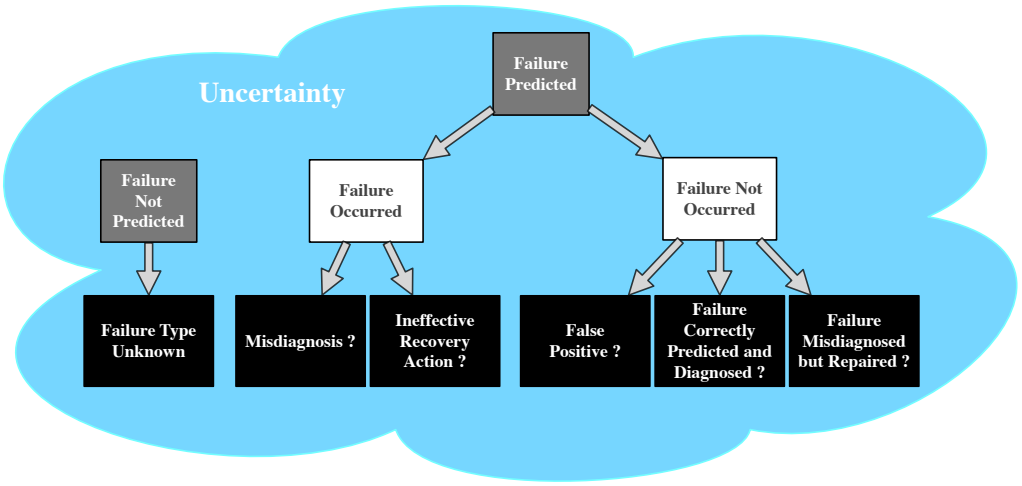


Figure 89: Learning data for training diagnosis models are incomplete in production systems, because the failure type occurred cannot be determined automatically.

The SHStream running modes presented for failure prediction (production mode and learning mode) are also applied to failure diagnosis. In production mode, SHStream performs failure classification but deactivates online learning. On the other hand, SHStream switches to the learning mode when running the benchmarks designed for online learning of models. During that period, the SHStream application is notified about the failure being injected through a flag file created by the script that runs the benchmarks. When the file is present, the SHStream application reads from the file the fault type injected. Thereafter, all log instances are labelled with the fault type read.

The Algorithm 3 redesigns the Algorithm 2 developed in Chapter 6, by combining the diagnosis activity with the failure prediction activity. It includes the classification of the faulty resource and the fault location, using two distinct groups of models: *ModelDiagRes* and *ModelDiagLoc*. For each group of models, the classification outcome given by the model with best classification performance up to the current time, is chosen as the diagnosis outcome for that group. This diagnosis process is similar to that followed for failure prediction.

7.3.4 *Experimental Work*

This section presents the experimental results of the:

1. Impact of the number of learning instances on the performance of classification of resources;
2. Breakdown of the performance of classification of resources by learning algorithm;

Algorithm 3 Algorithm for: (1) Online Learning and Evaluation of Models for Failure Prediction and Failure Diagnosis; and (2) Failure Prediction and Diagnosis of Log Data.

```
Require: size(buffer) is WindowOfUncertainty
  loop
    I ⇐ readNewInstance()
    f ⇐ isFailState(I)                                              ▷ Classification
    if f is false then
      for i = 1 to nModels do
        p_i ⇐ classifyFailurePrediction(Model_i, I)
      end for
    end if
    if p_mostAccurate is Failure or f is true then
      [res, loc] ⇐ diagnosis([ModelDiagRes_mostAccurateRes,
                              ModelDiagLoc_mostAccurateLoc], I)
      launchRecovery(res, loc)
    end if                                                          ▷ Learning
    L ⇐ buildLearningInstance(I, f, p, res, loc)
    if not isBufferFull(buffer) then
      jump to next loop iteration
    end if
    addToEnd(buffer, L)
    F ⇐ removeFirst(buffer)
    if distanceFail(buffer) ∈ [1, preFailWindow] then
      for i = 1 to nModels do
        learn(Model_i, F, Prefailure)
        updateModelStatistics(Model_i, F, Prefailure)
        if learningMode then
          learn([ModelDiagRes_i, ModelDiagLoc_i],
                              F, FaultTypeInjected)
          updateModelStatistics([ModelDiagRes_i, ModelDiagLoc_i],
                              F, FaultTypeInjected)
        end if
      end for
    else if distanceFail(buffer, F) is ∞ then
      if F.p_mostAccurate = Normal or learningMode then
        for i = 1 to nModels do
          learn(Model_i, F, Normal)
          updateModelStatistics(Model_i, F, Normal)
        end for
      end if
    end if                                                          ▷ Evaluation
    mostAccurate ⇐ evaluateBestModel(Model)
    mostAccurateRes ⇐ evaluateBestModel(ModelDiagRes)
    mostAccurateLoc ⇐ evaluateBestModel(ModelDiagLoc)
    logForOfflineAnalysis(F)
  end loop
```

3. Classification performance difference between reactive diagnosis and proactive diagnosis;

4. Breakdown of the performance of classification of error locations by learning algorithm.

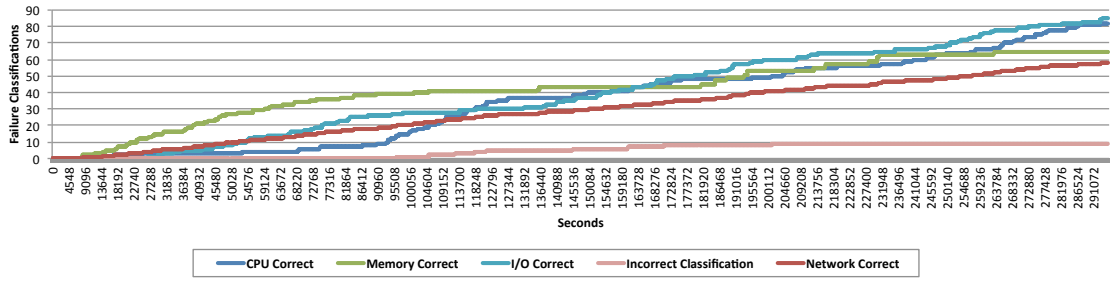


Figure 90: Number of log instances of each fault type correctly classified and number of log instances incorrectly classified.

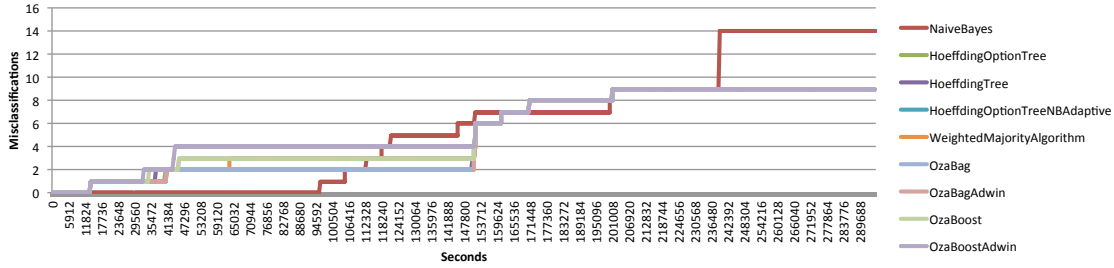


Figure 91: Breakdown of the number of misclassified log instances by algorithm.

In our experimental work, we use the mix+anomalyNet benchmark presented in Section 4.5.3.3. This benchmark extends the mix+anomaly benchmark by including network faults that manifest as packet losses and large packet delay variance.

7.3.4.1 *Impact of the Number of Learning Instances on the Performance of Classification of Resources*

Figure 90 presents the number of failure scenarios correctly diagnosed for each resource directly impacted by the fault injected and also, the sum of all incorrect classifications. It is noticeable that the diagnosis error stabilizes at 9 incorrect classifications, after training models with a number of instances varying between 40 and 60 for each resource type, using the model with best performance at each classification iteration. After stabilization, the number of misclassifications remains unchanged.

7.3.4.2 *Breakdown of the Performance of Classification of Resources by Learning Algorithm*

Figure 91 breaks down the number of misclassifications by each algorithm, during the execution of the benchmark. The Naïve Bayes performance takes longer to stabilize (after 14 misclassifications) than the other algorithms. After completing approximately two thirds of the learning process, all algorithms except Naïve Bayes stabilizes at 9 misclassifications.

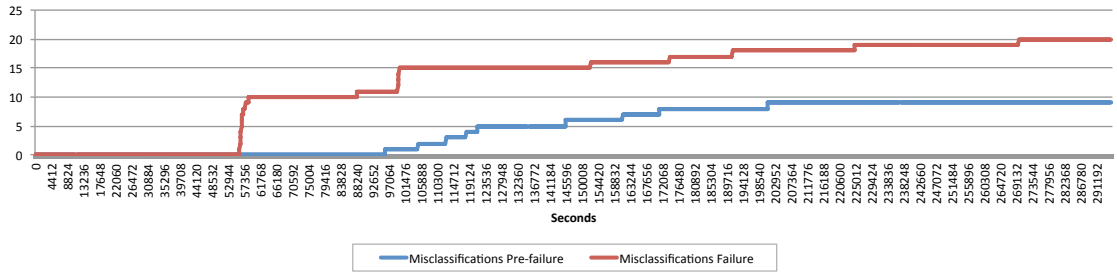


Figure 92: Misclassifications of resources using data gathered at prediction time (proactive diagnosis) and at failure time (reactive diagnosis).

7.3.4.3 *Classification Performance Difference Between Reactive Diagnosis and Proactive Diagnosis*

The evaluation of the reactive diagnosis approach requires changing the implementation of Algorithm 3 to train models with data gathered at the failure time (failure patterns), in replacement of data gathered at the prediction time (pre-failure patterns).

Figure 92 shows the number of misclassifications using models trained with pre-failure patterns (proactive diagnosis) and failure patterns (reactive diagnosis). It is noticeable the better performance of models trained with pre-failure patterns (9 misclassifications) compared with those created with failure patterns (20 misclassifications). Another important observation is that the number of misclassifications stabilizes after approximately 50 learning instances of each failure type, in average, for models trained with pre-failure patterns (Figure 90). By contrast, models created with failure patterns continue to misclassify log instances until the end of the experimental period.

7.3.4.4 *Breakdown of the Performance of Classification of Error Locations by Learning Algorithm*

In failure diagnosis, the classification of the faulty resource precedes the classification of the failure location. That means that when the resource diagnosed is any but the network, the log instance is classified again to determine whether the fault is located within the virtual container or outside it.

Figure 93 presents the number of correct classifications of each location and the total number of misclassifications. It is noticeable that the classification error stabilizes after 7 misclassifications, corresponding to approximately 60 learning instances of each location being used to train models.

7.3.5 *Discussion of Results*

The experimental results obtained testify the ability of online learning models to perform proactive diagnosis — using pre-failure patterns — and reactive diagnosis — us-

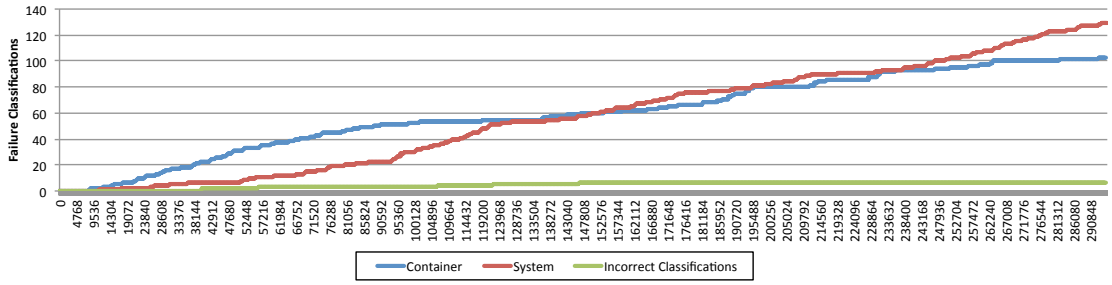


Figure 93: Number of correct classifications of failures localized inside and outside the virtual container, and the total number of misclassifications performed.

ing failure patterns — through classification of resources and locations of performance failures. The diagnosis outcomes are valuable information for selection of appropriate repair actions in the self-healing lifecycle (as described in Section 7.1.3) and also for supporting further fault removal activities.

7.3.5.1 *Diagnosis Performance of Online Learning Algorithms*

Online learning algorithms perform better in failure diagnosis than in failure prediction (Chapter 6), not only in terms of the best classifier, but also in terms of the performance homogeneity between classifiers. This observation is consistent with the results obtained for diagnosis in the Pure Streaming infrastructure. This is justified by the less noisier training datasets used for diagnosis, compared with those used for failure prediction. In failure diagnosis, the training data respects only faulty periods. This contrasts with the failure prediction approach, which permits mislabelled learning instances, due to the blurred boundary between the normal and the pre-failure periods, already discussed in Chapter 6.

Dynamic selection of models introduces diversity of learning algorithms in the failure prediction and diagnosis activities. Notwithstanding the multi-model diagnosis approach exhibited no performance improvement over a single-model diagnosis approach — in case the learning algorithm with the best performance is always used — the diversity of algorithms can provide robustness to the self-healing framework when the classification performance decays for any reason. Also, a multi-model infrastructure enables automatic model training and evaluation with new algorithms adopted by the infrastructure in the future. Therefore, we consider advantageous the implementation of a multi-model diagnosis approach, even because the lack of any performance penalty introduced by the online learning algorithms considered.

7.3.5.2 *Diagnosis Performance with the Number of Learning Instances*

The number of misclassifications of the resources associated to the failures is low until the stabilization of the classification error, even when a small number of learning in-

stances is available. Hoeffding Trees and ensemble algorithms revealed to be the fastest learning algorithms — measured by the number of learning instances required to avoid further misclassifications. On the other hand, Naïve Bayes required more learning instances until reaching the stabilization point. Yet, all models exhibited equivalent performance after some point during the execution of the benchmark.

7.3.5.3 *Performance Comparison Between Proactive Diagnosis and Reactive Diagnosis*

Pre-failure patterns proved to be powerful discriminators in failure diagnosis scenarios. When the diagnosis models are trained with data associated to pre-failure patterns, the number of misclassifications stabilizes before the end of the benchmark. Conversely, the classification models trained with failure patterns (data gathered at the failure time) misclassify log instances during the entire execution of the benchmark.

The better performance of models trained with pre-failure patterns relatively to models trained with failure patterns is explained by the use of learning data associated to server conditions where errors are in a early stage of propagation. Pre-failure patterns are captured in a stage where errors start to impact other components and resources, before they finally manifest into failures. Immediately after fault activation, the error is encountered with none or small ramifications. This is the best moment to perform diagnosis. Thus, the sooner the feature values are gathered, the closer they are to the fault activation during the propagation process. Consequently, the server states captured by features are simpler and will likely capture the first resource or resources affected by the fault, which are that or these directly impacted by it. With simpler patterns, the resultant models have less complexity and consequently, produce less classification errors.

7.4 CHAPTER SUMMARY

This chapter addressed the problem of diagnosing performance failures predicted or detected in video-streaming systems. Failure diagnosis was undertaken differently for the Pure Streaming and HTTP Streaming infrastructures presented in Chapter 4.

The Pure Streaming infrastructure is incorporated in a traditional non-virtualized environment. It includes a Pure Streaming server, which typically has complex client-server flows that difficult the control of the server load — based on the specification of client workloads being handled — making it prone to workload-related failures. Hence, failure diagnosis is defined for this infrastructure as the process of separating workload-related failures from failures caused by software faults (performance anomalies). The classification of these failures would allow applying countermeasures to reduce the server load, for workload-related failures, and triggering repair actions, for performance anomalies. The experimental results showed that decision tree models achieve perfect classification performance in the diagnosis activity implemented by the

Pure Streaming infrastructure. Plus, they demonstrated the resilience of classification models to workload changes in terms of popularity.

The HTTP Streaming infrastructure demands a diagnosis classification structure different from that considered for the Pure Streaming infrastructure. This is justified by two particularities of the former infrastructure: (1) it implements the concept of Autonomic Element using container-based virtualization technologies; and (2) it includes a self-protection mechanism against server overloading. Due to (1), the classification of performance anomalies into the resource directly affected by the fault (network or other server resource), combined with the classification of the localization of the associated errors (inside or outside the virtual container), would support the selection of repair techniques implemented by the infrastructure: server migration, system reboot or virtual container reboot. Collaterally, the classification of server resources can also support further fault removal activities. Due to (2), the implementation of self-protection mechanisms against server overloading in HTTP Streaming is facilitated by the simplicity of the HTTP flows. That means that only performance anomalies are expected to occur. The results obtained using the HTTP Streaming infrastructure showed that, using a sufficient number of learning instances — achieved using our benchmark — all models were able to discriminate without errors the resource responsible for each failure predicted and the location of the error.

Experimental results taken for HTTP Streaming expose the higher performance of proactive diagnosis over reactive diagnosis. The explanation for that phenomenon is the temporal proximity of the data gathered at the prediction time from the fault activation. The farthest the data gathered for analysis are from the fault activation, the higher the probability of error propagation throughout the application or the system, making system behavior patterns more complex. Intuitively, the simpler the failure patterns identified from log data, the easier to capture them by models and the smaller the number of learning instances required to stabilize the classification performance. Pre-failure patterns are thus identified as important contributors for failure diagnosis activities.

8

CONCLUSIONS

The main idea of Autonomic Computing resides in the development of self-aware systems, able to manage themselves without (or with minimum) human intervention. This thesis explores the self-healing aspect of Autonomic Computing, whose concern is to take measures for overcoming performance failures using self-knowledge about the system behavior. As any other Autonomic Computing aspect, the self-healing aspect implements the MAPE-K control loop, constituted mainly by five parts: monitoring, analyzing, planning, executing and knowledge base. These parts form a circulatory system.

Self-healing is applied to autonomous proactive and reactive recovery of performance failures in video-streaming systems. It covers the monitoring, failure prediction, failure diagnosis and repair activities that instantiate the MAPE-K concept in our self-healing infrastructures. Self-healing systems perform monitoring using sensors to collect measurements from the environment. The resulting data are analyzed according to a knowledge base that supports the detection and analysis of anomalous system states, triggering the planning and execution of repair actions. One self-healing infrastructure is presented for each of the video-streaming technologies addressed in this thesis: Pure Streaming and HTTP Streaming.

Pure Streaming is a traditional video-streaming technology less popular nowadays. This technology uses the RTP protocol for transmitting data at the same pace they are processed by video players. This protocol is accompanied by the RTCP protocol, which provides control functions, such as synchronization, reporting and data reception statistics.

Pure Streaming has been replaced by HTTP Streaming in the recent years. The latter benefits from its protocol-level simplicity, permeability of HTTP traffic through firewalls and the widespread availability of the infrastructure that supports the HTTP ecosystem in the Internet. HTTP Streaming techniques include Progressive Download and Adaptive Bitrate. In Progressive Download services, the request-responses are transmitted by the server similarly to any other web static resource. On the other hand, Adaptive Bitrate streaming allows video files to be decomposed into small downloadable segments, each encoded with different qualities. Each segment is requested by the video player with a specific encoding determined from the analysis of several parameters. This guarantees dynamic adaption to client-side, network and server-side conditions during the video playback.

The self-healing lifecycle for the Pure Streaming infrastructure is defined as the sequence of monitoring[1], failure prediction and failure diagnosis. That sequence is extended with the repair activity in the HTTP Streaming infrastructure.

The monitoring activity is implemented similarly in both self-healing infrastructures, except in terms of the application-level metrics adopted. Failure prediction uses batch learning algorithms in the Pure Streaming infrastructure and online learning algorithms in the HTTP Streaming infrastructure. Also, the classification structure adopted for the diagnosis activity is different in both infrastructures. In the Pure Streaming infrastructure, failures are classified into system overloading failures — caused by client workloads — and performance anomalies — caused by software faults. This is an appropriate classification scheme, since the complex behaviors of this video-streaming approach — mainly due to complex client-server flows — makes the control of the server load based on the specification of client workloads being handled, a difficult and error prone task. Diagnosis in HTTP Streaming is implemented as the classification of the failure resource (network or one of the server internal resources) and the failure location (inside or outside the virtual container where the server is running).

The HTTP Streaming infrastructure implements container-based virtualization to ensure performance isolation between the self-healing functionality and the main server functionality within the Autonomic Element, and also to support server repair interventions. For repair interventions, container-based virtualization provides a structure for efficient server failover between machines, reboot and server (re)instantiation in any machine, independently of the operating system or any other processes running outside the server's virtual container. Since container-based virtualization runs on top of the operating system, the basic unit of rebooting, checkpointing and migration is delimited by the management structures of the virtual container and the in-memory structures attached to the video-streaming server's process. The self-healing infrastructure combines the facilities provided by virtualization with a methodology to warm-up the server after reboot, without degradation of the quality of experience of end-users under certain assumptions.

8.1 MAIN CONTRIBUTIONS OF THIS THESIS

This thesis presents the design, technologies, methodologies and experimental evaluation results of two self-healing infrastructures for video-streaming systems. The novelty of our work is present at all these levels.

Our HTTP Streaming infrastructure is adequate for recent video-streaming services, since HTTP Streaming is nowadays the de facto streaming technology for delivering video-streaming content in the Internet. This infrastructure relies on container-based virtualization for building an Autonomic Element that instantiate the MAPE-K concept through self-monitoring, self-prediction, self-diagnosis and self-repair capabilities. So,

1 Including data gathering and failure detection.

any virtual machine or physical machine running a video server can be converted into an Autonomic Element using our approach. To that purpose, container-based virtualization is adopted to isolate the performance of the self-healing functionality (Autonomic Manager) from the performance of the server application (Managed Element), collocated in the same host. To our knowledge there has been no previous attempt to build Autonomic Elements from server applications in a generic way.

The major contributions of this thesis are provided by the failure prediction and failure diagnosis activities. These activities create the opportunity for recovering failures without QoE penalties, an important requirement of time sensitive services as video-streaming. Both failure prediction and failure diagnosis activities use either batch learning algorithms or online learning algorithms for building models iteratively from log data. The self-healing infrastructure implements the functionality required to build and evaluate models for failure prediction and diagnosis. It includes feature selection — implicitly by some learning algorithms or using independent feature selection algorithms — multi-algorithm model building, dynamic evaluation of models and dynamic selection of models.

Failure prediction models reflect pre-failure patterns that are able of characterizing anomalous system conditions before they translate into user-visible failures in the short-term. In addition to the benefits of failure anticipation, prediction models also recognize unhealthy system conditions that otherwise would be confounded with transient service failures (e.g., network variations). Pre-failure patterns were shown also to be powerful discriminators for diagnosis of predicted failures, since they represent errors in an early stage of propagation. As shown through experimental evaluation, classification of pre-failure patterns is more accurate than classification of failure patterns identified on log data after failure occurrence.

Despite relying on generic virtualization techniques to repair the system, our work provides important insights for selection of the most effective and efficient virtualization techniques and their expected impact on the service quality. The use of container-based virtualization for supporting efficient server migration and reboot techniques is also innovative and a relevant contribution for repairing systems with the video-streaming characteristics. These techniques create small downtimes that could be absorbed by client-side buffering. This assumption enables service continuity when repair is triggered either by true positives or false positives in the failure prediction activity. Despite the small percentage of false positives, its impact on the service quality is nullified by the low cost repair actions proposed.

The server warm-up approach is another important contribution. It allows the repair of the system while avoiding failures caused by the reduced server capacity during the warm-up period. This approach can be applied to both operating system and virtual container reboots.

8.2 FUTURE WORK

This thesis presents several contributions to the area of self-healing of video-streaming systems. However, there are much work to be done in this area, mainly in the failure prediction, failure diagnosis and repair activities.

We believe that an important part of the work to be done in self-healing systems is related with the development of benchmarks to train models capable of representing comprehensively the system normal and faulty behaviors. The high frequency of updates and the dynamic allocation of system resources provided by modern virtualization infrastructures make system models obsolete in the short-term. Consequently, these models have to be retrained frequently.

Online learning algorithms could train models when the systems are being used in production environments. However, the automatic segmentation of log data relative to normal, pre-failure and failure periods cannot be always performed, in particular for diagnosis problems, as explained in Chapter 6 and Chapter 7. Therefore, benchmarks are a promising approach for training system models automatically in short periods of time. The efficiency of these benchmarks is an important concern, due to the frequency of system changes demanding new models.

Our research work in repair techniques for video-streaming systems can be evolved in two different ways:

1. Develop an approach to provide completion guarantees for the warm-up process after a server reboot, without QoE degradation;

2. Develop a mechanism for selection of hosts to receive virtual containers, when employing the: (1) server migration technique; or (2) restoration of a rebooted virtual container in another host.

This thesis presents a server warm-up approach for reboot actions, but without completion guarantees. Recovery guarantees are provided only for the restoration of the server. Devising a server warm-up approach using the workload types being handled by the server before its recovery is a research issue that deserves further investigation. This issue is not confined to the Autonomic Element context, since it is required the intervention of other servers and/or external services. One potential strategy for the server warm-up problem is to replace the faulty virtual container by a replica of a healthy virtual container that is currently handling a similar workload type (e.g., a server node behind the same load balancer). Yet, this solution presents several challenges, because not only the virtual container should be warmed-up, but also the host where it is running. Hence, the restored virtual container would also be hosted in one of the hosts that have been handling a similar workload.

The repair techniques proposed in this thesis depends on an orthogonal solution to select the host that will receive the virtual container when the server is migrated

or when the rebooted virtual container is restored in another host. A host selection methodology based on the resources being used by each virtual container (e.g., using the *beancounters* described in Chapter 5) and the hosts that have been warmed-up with a similar workload, can be investigated in future work.

BIBLIOGRAPHY

[ope] Openvz. URL http://wiki.openvz.org/. (Accessed: 2014-09-30).

[rsy] rsync. URL http://rsync.samba.org/. (Accessed: 2014-09-30).

[str a] Streamingloadtool, April a. URL http://www.opensource.apple.com/source/QuickTimeStreamingServer/QuickTimeStreamingServer-452/StreamingLoadTool. (Accessed: 2014-09-30).

[str b] Stress tool, b. URL http://weather.ou.edu/~apw/projects/stress/. (Accessed: 2014-09-30).

[PIP 1998] Real time streaming protocol, 1998. URL http://tools.ietf.org/html/rfc2326. (Accessed: 2014-09-30).

[key 2010] Keynote streaming perspective streamq, 2010. URL http://keynote.com. (Accessed: 2014-09-30).

[ama 2012] Amazon elb service event in the us-east region, April 2012. URL http://aws.amazon.com/message/680587/. (Accessed: 2014-09-30).

[dar 2012] Darwin streaming server, April 2012. URL http://dss.macosforge.org/. (Accessed: 2014-09-30).

[goo 2012] Google apps incident report, April 2012. URL http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/us/appsstatus/ir/plibxfjh8whr44h.pdf. (Accessed: 2014-09-30).

[h26 2012] H264 streaming module, April 2012. URL http://h264.code-shop.com/trac. (Accessed: 2014-09-30).

[sar 2012] System activity report, April 2012. URL http://linux.die.net/man/1/sar. (Accessed: 2014-09-30).

[sig 2012] Hyperic's system information gatherer (sigar) api, 2012. URL http://sourceforge.net/projects/sigar/files/. (Accessed: 2014-09-30).

[vim 2012] Vimeo, April 2012. URL http://vimeo.com. (Accessed: 2014-09-30).

[you 2012] Youtube, April 2012. URL http://www.youtube.com. (Accessed: 2014-09-30).

[Abhari and Soraya 2010] Abdolreza Abhari and Mojgan Soraya. Workload generation for youtube. *Multimedia Tools Appl.*, 46(1):91–118, January 2010. ISSN 1380-7501. doi: 10.1007/s11042-009-0309-5. URL http://dx.doi.org/10.1007/s11042-009-0309-5.

[Adamic and Huberman 2002] Lada A Adamic and Bernardo A Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[Adhikari et al. 2012] V.K. Adhikari, S. Jain, Yingying Chen, and Zhi-Li Zhang. Vivisecting youtube: An active measurement study. In *INFOCOM, 2012 Proceedings IEEE*, pages 2521–2525, March 2012. doi: 10.1109/INFCOM.2012.6195644.

[Agrawal et al. 1993] R. Agrawal, T. Imielinski, and A. Swami. Database mining: a performance perspective. *Knowledge and Data Engineering, IEEE Transactions on*, 5(6):914 –925, dec 1993.

[Akoush et al. 2010] S. Akoush, R. Sohan, A. Rice, A.W. Moore, and A. Hopper. Predicting the performance of virtual machine migration. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 37 –46, aug. 2010. doi: 10.1109/MASCOTS.2010.13.

[Ameigeiras et al. 2012] Pablo Ameigeiras, Juan J. Ramos-Munoz, Jorge Navarro-Ortiz, and J.M. Lopez-Soler. Analysis and modeling of youtube traffic. *Transactions on Emerging Telecommunications Technologies*, 23(4):360–377, 2012. ISSN 2161-3915. doi: 10.1002/ett.2546. URL http://dx.doi.org/10.1002/ett.2546.

[Arlat et al. 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *Software Engineering, IEEE Transactions on*, 16(2):166–182, Feb 1990. ISSN 0098-5589.

[Arlitt and Williamson 1997] Martin F. Arlitt and Carey L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Trans. Netw.*, 5(5):631–645, October 1997. ISSN 1063-6692. doi: 10.1109/90.649565. URL http://dx.doi.org/10.1109/90.649565.

[Arpaci-Dusseau and Arpaci-Dusseau 2001] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, pages 33–, Washington, DC, USA, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=874075.876394.

[Attariyan et al. 2012] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[Avizienis et al. 2004] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11 – 33, jan.-march 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

[Aweya et al. 2002] James Aweya, Michel Ouellette, Delfin Y. Montuno, Bernard Doray, and Kent Felske. An adaptive load balancing scheme for web servers. *International Journal of Network Management*, 12(1):3–39, 2002. ISSN 1099-1190. doi: 10.1002/nem.421. URL http://dx.doi.org/10.1002/nem.421.

[Azim et al. 2014] Md. Tanzirul Azim, Iulian Neamtiu, and Lisa M. Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 623–628, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642955. URL http://doi.acm.org/10.1145/2642937.2642955.

[Babcock et al. 2002] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-507-6.

[Barth 2008] Wolfgang Barth. *Nagios: System and Network Monitoring*. No Starch Press, San Francisco, CA, USA, 2nd edition, 2008. ISBN 1593271794.

[Ben Halima et al. 2008] R. Ben Halima, K. Drira, and M. Jmaiel. A qos-oriented reconfigurable middleware for self-healing web services. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 104–111, Sept 2008. doi: 10.1109/ICWS.2008.113.

[Bifet and Kirkby 2009] Albert Bifet and Richard Kirkby. Data stream mining: a practical approach. Technical report, The University of Waikato, August 2009.

[Bifet et al. 2009] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New ensemble methods for evolving data streams. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 139–148. ACM, 2009.

[Bifet et al. 2010] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, August 2010. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1756006.1859903.

[Björck 1996] Åke Björck. *Numerical methods for least squares problems*. Siam, 1996.

[Bodik et al. 2005] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A Fox, M.I Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 89–100, June 2005. doi: 10.1109/ICAC.2005.18.

[Bouckaert 2003] Remco R. Bouckaert. Choosing between two learning algorithms based on calibrated tests. In *In ICML'03*, pages 51–58. Morgan Kaufmann, 2003.

[Brecht et al. 2004] Tim Brecht, David Pariag, and Louay Gammo. accept () able strategies for improving web server performance. In *USENIX Annual Technical Conference, General Track*, pages 227–240, 2004.

[Breiman 1996a] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996a. ISSN 0885-6125. doi: 10.1007/BF00058655. URL http://dx.doi.org/10.1007/BF00058655.

[Breiman 1996b] Leo Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996b. ISSN 0885-6125.

[Breslau et al. 1999] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134 vol.1, 1999. doi: 10.1109/INFCOM.1999.749260.

[Bressoud and Schneider 1996] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, February 1996. ISSN 0734-2071. doi: 10.1145/225535.225538. URL http://doi.acm.org/10.1145/225535.225538.

[Brown et al. 2001] A Brown, G. Kar, and A Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 377–390, 2001. doi: 10.1109/INM.2001.918054.

[Bucciol et al. 2005] P. Bucciol, E. Masala, N. Kawaguchi, K. Takeda, and J.C. De Martin. Performance evaluation of h. 264 video streaming over inter-vehicular 802.11 ad hoc networks. In *Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005. IEEE 16th International Symposium on*, volume 3, pages 1936 –1940, sept. 2005. doi: 10.1109/PIMRC.2005.1651778.

[Candea and Fox 2001a] G. Candea and A. Fox. Recursive restartability: turning the reboot sledgehammer into a scalpel. *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 125–130, May 2001a.

[Candea and Fox 2001b] George Candea and Armando Fox. Designing for high availability and measurability. In *Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability*, 2001b.

[Candea and Fox 2003] George Candea and Armando Fox. Crash-only software. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251054.1251066.

[Carreira et al. 1998] J. Carreira, H. Madeira, and J.G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *Software Engineering, IEEE Transactions on*, 24(2):125–136, Feb 1998. ISSN 0098-5589. doi: 10.1109/32.666826.

[Carzaniga et al. 2008] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. Healing web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer*, 10(6):493–502, 2008. ISSN 1433-2779. doi: 10.1007/s10009-008-0088-8. URL http://dx.doi.org/10.1007/s10009-008-0088-8.

[Castelli et al. 2001] V. Castelli, R.E. Harper, P. Heidelberger, S.W. Hunter, K.S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, March 2001. ISSN 0018-8646. doi: 10.1147/rd.452.0311.

[Cha et al. 2007] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 1–14, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-908-1. doi: 10.1145/1298306.1298309. URL http://doi.acm.org/10.1145/1298306.1298309.

[Chakareski et al. 2005] J. Chakareski, S. Han, and B. Girod. Layered coding vs. multiple descriptions for video streaming over multiple paths. *Multimedia Systems*, 10(4):275–285, 2005. ISSN 0942-4962. doi: 10.1007/s00530-004-0162-3. URL http://dx.doi.org/10.1007/s00530-004-0162-3.

[Chawla 2005] NiteshV. Chawla. Data mining for imbalanced datasets: An overview. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 853–867. Springer US, 2005. ISBN 978-0-387-24435-8. doi: 10.1007/0-387-25465-X_40. URL http://dx.doi.org/10.1007/0-387-25465-X_40.

[Che et al. 2010] Jianhua Che, Yong Yu, Congcong Shi, and Weimin Lin. A synthetical performance evaluation of openvz, xen and kvm. In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pages 587 –594, dec. 2010. doi: 10.1109/APSCC.2010.83.

[Chen et al. 2002] MY Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, pages 595–604, 2002.

[Chen et al. 1994] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994. ISSN 0360-0300. doi: 10.1145/176979.176981. URL http://doi.acm.org/10.1145/176979.176981.

[Cheng et al. 2002] Shang-Wen Cheng, D. Garlan, B. Schmerl, P. Steenkiste, and Ningning Hu. Software architecture-based adaptation for grid computing. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 389–398, 2002. doi: 10.1109/HPDC.2002.1029939.

[Cheng 2007] Xu Cheng. Understanding the characteristics of internet short video sharing: Youtube as a case study. In *Procs of the 7th ACM SIGCOMM Conference on Internet Measurement, San Diego (CA, USA), 15*, page 28, 2007.

[Cheng et al. 2008] Xu Cheng, C. Dale, and Jiangchuan Liu. Statistics and social network of youtube videos. In *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pages 229–238, 2008. doi: 10.1109/IWQOS.2008.32.

[Cherkasova and Gupta 2004] L. Cherkasova and Minaxi Gupta. Analysis of enterprise media server workloads: access patterns, locality, content evolution, and rates of change. *Networking, IEEE/ACM Transactions on*, 12(5):781–794, Oct 2004. ISSN 1063-6692. doi: 10.1109/TNET.2004.836125.

[Cherkasova and Phaal 2002] L. Cherkasova and P. Phaal. Session-based admission control: a mechanism for peak load management of commercial web sites. *Computers, IEEE Transactions on*, 51(6):669–685, Jun 2002. ISSN 0018-9340. doi: 10.1109/TC.2002.1009151.

[Cherkasova et al. 2008] L. Cherkasova, K. Ozonat, Ningfang Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 452 –461, june 2008.

[Cherkasova and Staley 2003] Ludmila Cherkasova and Loren Staley. Building a performance model of streaming media applications in utility data center environment. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 52, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1919-9.

[Chun et al. 2008] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 287–292, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1404014.1404038.

[Clark et al. 2005] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems*

Design & Implementation - Volume 2, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251203.1251223.

[Claypool and Tanner 1999] Mark Claypool and Jonathan Tanner. The effects of jitter on the perceptual quality of video. In *Proceedings of the seventh ACM international conference on Multimedia (Part 2)*, MULTIMEDIA '99, pages 115–118, New York, NY, USA, 1999. ACM. ISBN 1-58113-239-5. doi: 10.1145/319878.319909. URL http://doi.acm.org/10.1145/319878.319909.

[Cohen et al. 2004] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design Implementation*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[Collange et al. 2012] D. Collange, M. Hajji, J. Shaikh, M. Fiedler, and P. Arlos. User impatience and network performance. In *Next Generation Internet (NGI), 2012 8th EURO-NGI Conference on*, pages 141–148, June 2012. doi: 10.1109/NGI.2012.6252146.

[Corporation 2009] Standard Performance Evaluation Corporation. web2009 benchmark, 2009. URL http://www.spec.org/web2009. (Accessed: 2014-09-30).

[Cotroneo et al. 2003] D. Cotroneo, C. di Flora, G. Paolillo, and S. Russo. Modeling and detecting failures in next-generation distributed multimedia applications. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 379 – 388, oct. 2003. doi: 10.1109/RELDIS.2003.1238091.

[Covell et al. 2004] M. Covell, B. Seo, S. Roy, M. Spasojevic, L. Kontothanassis, N. Bhatti, and R. Zimmermann. Calibration and prediction of streaming-server performance. *HP Labs Tecnical Report HPL-2004-206*, 2004.

[Coverdale 2001] P. Coverdale. Itu-t study group 12: Multimedia qos requirements from a user perspective. In *Workshop on QoS and user perceived transmission quality in evolving networks*, 2001.

[Cully et al. 2008] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association. ISBN 111-999-5555-22-1. URL http://dl.acm.org/citation.cfm?id=1387589.1387601.

[Cunha and Silva 2015a] C. A. Cunha and L. M. Silva. Reboot-based recovery of performance anomalies in adaptive bitrate video-streaming services. *Special Issue on High Performance Computing in Parallel and Distributed Systems*, 2015a.

[Cunha and Silva 2016] C. A. Cunha and L. M. Silva. Building autonomic elements from video-streaming servers. *Dependable and Secure Computing, IEEE Transactions on*, 2016.

[Cunha and Silva 2012] C.A. Cunha and L.M. Silva. Separating performance anomalies from workload-explained failures in streaming servers. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 292–299, May 2012. doi: 10.1109/CCGrid.2012.58.

[Cunha and Silva 2013a] C.A. Cunha and L.M. Silva. Shstream: Self-healing framework for http video-streaming. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 514–521, May 2013a. doi: 10.1109/CCGrid.2013.87.

[Cunha and Silva 2013b] C.A. Cunha and L.M. Silva. Prediction of performance failures in video-streaming servers. In *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, pages 283–292, Dec 2013b. doi: 10.1109/PRDC.2013.50.

[Cunha and Silva 2015b] C.A. Cunha and L.M. Silva. Reboot-based recovery of performance anomalies in adaptive bitrate video-streaming services. In *Dependable Computing (PRDC), 2015 IEEE 21th Pacific Rim International Symposium on*, Nov 2015b.

[Cunha and Silva 2011] C.A.S. Cunha and L.M. Silva. Failure prediction in video-streaming servers through performance analysis of server and client-server interactions. In *Parallel and Distributed Computing and Networks / 720: Software Engineering*. ACTA Press, 2011. doi: 10.2316/p.2011.719-065. URL http://dx.doi.org/10.2316/P.2011.719-065.

[Dai et al. 2006] Yuan-Shun Dai, M. Hinchey, M. Madhusoodan, J.L. Rash, and Xukai Zou. A prototype model for self-healing and self-reproduction in swarm robotics system. In *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, pages 3–10, Sept 2006. doi: 10.1109/DASC.2006.10.

[Dai et al. 2011] Yuanshun Dai, Yanping Xiang, Yanfu Li, Liudong Xing, and Gewei Zhang. Consequence oriented self-healing and autonomous diagnosis for highly reliable systems and software. *Reliability, IEEE Transactions on*, 60(2):369–380, June 2011. ISSN 0018-9529. doi: 10.1109/TR.2011.2136490.

[Dashofy et al. 2002] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pages 21–26, New York, NY, USA, 2002. ACM. ISBN 1-58113-609-9. doi: 10.1145/582128.582133. URL http://doi.acm.org/10.1145/582128.582133.

[Davis and Goadrich 2006] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on*

Machine Learning, ICML '06, pages 233–240, New York, NY, USA, 2006. ACM. ISBN 1-59593-383-2. doi: 10.1145/1143844.1143874. URL http://doi.acm.org/10.1145/1143844.1143874.

[Dean and Barroso 2013] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408794. URL http://doi.acm.org/10.1145/2408776.2408794.

[Dietterich 2000] Thomas G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157, 2000. ISSN 0885-6125. doi: 10.1023/A:1007607513941. URL http://dx.doi.org/10.1023/A%3A1007607513941.

[Do Cuong 2007] Cuong Do Cuong. Seattle conference on scalability: Youtube scalability. *Video, June*, 2007.

[Dobre et al. 2011] C. Dobre, F. Pop, V. Cristea, and O.M. Achim. A virtualization-based approach to dependable service computing. *Scalable Computing: Practice and Experience*, 12(3), 2011.

[Domingos and Hulten 2000] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM. ISBN 1-58113-233-6.

[Domingos and Pazzani 1997] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997. ISSN 0885-6125. doi: 10.1023/A:1007413511361. URL http://dx.doi.org/10.1023/A%3A1007413511361.

[Elnozahy et al. 2002] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002. ISSN 0360-0300.

[Fayyad et al. 1996] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996. ISBN 0-262-56097-6.

[Feamster and Balakrishnan 2002] Nick Feamster and Hari Balakrishnan. Packet loss recovery for streaming video. In *12th International Packet Video Workshop*, pages 9–16. PA: Pittsburgh, 2002.

[Fiedler et al. 2010] M. Fiedler, T. Hossfeld, and P. Tran-Gia. A generic quantitative relationship between quality of experience and quality of service. *Network, IEEE*, 24(2): 36–41, 2010. ISSN 0890-8044. doi: 10.1109/MNET.2010.5430142.

[Finamore et al. 2011] Alessandro Finamore, Marco Mellia, Maurizio M. Munafò, Ruben Torres, and Sanjay G. Rao. Youtube everywhere: Impact of device and infrastructure synergies on user experience. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 345–360, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1013-0. doi: 10.1145/2068816.2068849. URL http://doi.acm.org/10.1145/2068816.2068849.

[Forbes et al. 2011] Catherine Forbes, Merran Evans, Nicholas Hastings, and Brian Peacock. *Statistical distributions*. John Wiley & Sons, 2011.

[Forrest et al. 1996] S. Forrest, S.A Hofmeyr, A Somayaji, and T.A Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128, May 1996. doi: 10.1109/SECPRI.1996.502675.

[Freund and Schapire 1995] Yoav Freund and Robert Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In Paul Vitányi, editor, *Computational Learning Theory*, volume 904 of *Lecture Notes in Computer Science*, pages 23–37. Springer Berlin / Heidelberg, 1995. ISBN 978-3-540-59119-1.

[Friedman et al. 1997] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997. ISSN 0885-6125. doi: 10.1023/A:1007465528199. URL http://dx.doi.org/10.1023/A%3A1007465528199.

[Frossard 2001] P. Frossard. Fec performance in multimedia streaming. *Communications Letters, IEEE*, 5(3):122–124, March 2001. ISSN 1089-7798. doi: 10.1109/4234.913160.

[Fuad et al. 2006] M.M. Fuad, D. Deb, and M.J. Oudshoorn. Adding self-healing capabilities into legacy object oriented application. In *Autonomic and Autonomous Systems, 2006. ICAS '06. 2006 International Conference on*, pages 51–51, July 2006. doi: 10.1109/ICAS.2006.10.

[Gama et al. 2009] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. Issues in evaluation of stream learning algorithms. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 329–338, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9.

[Ganek and Corbi 2003] A.G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003. ISSN 0018-8670. doi: 10.1147/sj.421.0005.

[Garlan et al. 2004] D. Garlan, Shang-Wen Cheng, An-Cheng Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, Oct 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.175.

[Garlan and Schmerl 2002] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pages 27–32, New York, NY, USA, 2002. ACM. ISBN 1-58113-609-9. doi: 10.1145/582128.582134. URL http://doi.acm.org/10.1145/582128.582134.

[Gaudin et al. 2011] Benoit Gaudin, Emil Iordanov Vassev, Patrick Nixon, and Michael Hinchey. A control theory based approach for self-healing of un-handled runtime exceptions. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 217–220, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0607-2. doi: 10.1145/1998582.1998633. URL http://doi.acm.org/10.1145/1998582.1998633.

[George et al. 2003] Selvin George, David Evans, and Steven Marchette. A biological programming model for self-healing. In *Proceedings of the 2003 ACM Workshop on Survivable and Self-regenerative Systems: In Association with 10th ACM Conference on Computer and Communications Security*, SSRS '03, pages 72–81, New York, NY, USA, 2003. ACM. ISBN 1-58113-784-2. doi: 10.1145/1036921.1036929. URL http://doi.acm.org/10.1145/1036921.1036929.

[Ghinea and Thomas 1998] G. Ghinea and J. P. Thomas. Qos impact on user perception and understanding of multimedia video clips. In *Proceedings of the sixth ACM international conference on Multimedia*, MULTIMEDIA '98, pages 49–54, New York, NY, USA, 1998. ACM. ISBN 0-201-30990-4. doi: 10.1145/290747.290754. URL http://doi.acm.org/10.1145/290747.290754.

[Gill et al. 2007] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: A view from the edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 15–28, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-908-1. doi: 10.1145/1298306.1298310. URL http://doi.acm.org/10.1145/1298306.1298310.

[Gorla et al. 2012] Alessandra Gorla, Mauro Pezze, Jochen Wuttke, Leonardo Mariani, and Fabrizio Pastore. Achieving cost-effective software reliability through self-healing. *Computing and Informatics*, 29(1):93–115, 2012.

[Gray 1996] J.N. Gray. Why do computers stop and what can be done about it? In Computer Society Press, editor, *Fifth IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1996.

[Grottke et al. 2006] M. Grottke, Lei Li, K. Vaidyanathan, and K.S. Trivedi. Analysis of software aging in a web server. *Reliability, IEEE Transactions on*, 55(3):411 –420, sept. 2006.

[Gu and Wang 2009] Xiaohui Gu and Haixun Wang. Online anomaly prediction for robust cluster systems. *Data Engineering, International Conference on*, 0:1000–1011, 2009. ISSN 1084-4627.

[Gupta et al. 2009] H. Gupta, A. Mahanti, and V.J. Ribeiro. Revisiting coexistence of poissonity and self-similarity in internet traffic. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1–10, Sept 2009. doi: 10.1109/MASCOT.2009.5366239.

[Gurguis and Zeid 2005] Sherif A. Gurguis and Amir Zeid. Towards autonomic web services: Achieving self-healing using web services. *SIGSOFT Softw. Eng. Notes*, 30 (4):1–5, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083069. URL http://doi.acm.org/10.1145/1082983.1083069.

[Gutlein et al. 2009] M. Gutlein, E. Frank, M. Hall, and A. Karwath. Large-scale attribute selection using wrappers. In *Computational Intelligence and Data Mining, 2009. CIDM '09. IEEE Symposium on*, pages 332–339, 2009. doi: 10.1109/CIDM.2009.4938668.

[Guyon and Elisseeff 2003] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, March 2003. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=944919.944968.

[Hall et al. 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009. ISSN 1931-0145. doi: 10.1145/1656274.1656278. URL http://doi.acm.org/10.1145/1656274.1656278.

[Han et al. 2006] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006.

[Han et al. 1995] S. Han, K.G. Shin, and H.A Rosenberg. Doctor: an integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213, Apr 1995. doi: 10.1109/IPDS.1995.395831.

[Hastie et al. 2001] Trevor Hastie, Robert Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction: with 200 full-color illustrations*. New York: Springer-Verlag, 2001.

[Hastie et al. 2009] Trevor Hastie, Robert Tibshirani, Jerome Friedman, T Hastie, J Friedman, and R Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[Hawkins 2004] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[Hearst et al. 1998] Marti A. Hearst, ST Dumais, E Osman, John Platt, and Bernhard Scholkopf. Support vector machines. *Intelligent Systems and their Applications, IEEE*, 13(4):18–28, 1998.

[Hemminger 2005] S. Hemminger. Network emulation with netem. In *Linux Conf Au*, April 2005. URL http://developer.osdl.org/shemminger/netem/LCA2005_paper.pdf.

[Hofer and Fahringer 2007] Jürgen Hofer and Thomas Fahringer. Grid application fault diagnosis using wrapper services and machine learning. In BerndJ. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 233–244. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74973-8. doi: 10.1007/978-3-540-74974-5_19.

[Hoffmann et al. 2007] G.A Hoffmann, K.S. Trivedi, and M. Malek. A best practice guide to resource forecasting for computing systems. *Reliability, IEEE Transactions on*, 56(4): 615–628, Dec 2007. ISSN 0018-9529. doi: 10.1109/TR.2007.909764.

[Holmes et al. 2005] Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. Stress-testing hoeffding trees. In AlípioMário Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama, editors, *Knowledge Discovery in Databases: PKDD 2005*, volume 3721 of *Lecture Notes in Computer Science*, pages 495–502. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-29244-9. doi: 10.1007/11564126_50. URL http://dx.doi.org/10.1007/11564126_50.

[Horn et al. 2001] Gavin B. Horn, Per Knudsgaard, Soren B. Lassen, Michael Luby, and Jens Eilstrup Rasmussen. A scalable and reliable paradigm for media on demand. *Computer*, 34(9):40–45, 2001. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/2.947088.

[Horn 2001] Paul Horn. Autonomic computing: Ibm\'s perspective on the state of information technology. 2001.

[Huhns et al. 2003] Michael N. Huhns, Vance T. Holderfield, and Rosa Laura Zavala Gutierrez. Robust software via agent-based redundancy. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, pages 1018–1019, New York, NY, USA, 2003. ACM. ISBN 1-58113-683-8. doi: 10.1145/860575.860774. URL http://doi.acm.org/10.1145/860575.860774.

[Ickin et al. 2013] S. Ickin, M. Fiedler, and K. Wac. Energy-based anomaly detection in quality of experience. In *Wireless Personal Multimedia Communications (WPMC), 2013 16th International Symposium on*, pages 1–6, June 2013.

[Jiang and Schulzrinne 2002] W. Jiang and H. Schulzrinne. Perceived quality of packet audio under bursty losses. In *IEEE INFOCOM*, page 1, 2002.

[Jiang and Schulzrinne 2000] Wenyu Jiang and Henning Schulzrinne. Modeling of packet loss and delay and their effect on real-time multimedia service quality. In *PROCEEDINGS OF NOSSDAV '2000*, 2000.

[Jovanovic et al. 2006] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6 pp.–263, May 2006. doi: 10.1109/SP.2006.29.

[Kaiser et al. 2002] Gail E Kaiser, Philip N Gross, Gaurav Kc, Janak Parekh, and Giuseppe Valetto. An approach to autonomizing legacy systems. *Computer Science Technical Report Series*, 2002.

[Kang et al. 2008] Xiaozhu Kang, Hui Zhang, Guofei Jiang, Haifeng Chen, Xiaoqiao Meng, and Kenji Yoshihira. Measurement, modeling, and analysis of internet video sharing site workload: A case study. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 278–285. IEEE, 2008.

[Kelly 2005] Terence Kelly. Detecting performance anomalies in global applications. In *Proceedings of the 2nd conference on Real, Large Distributed Systems - Volume 2*, WORLDS'05, pages 42–47, Berkeley, CA, USA, 2005. USENIX Association.

[Kephart and Chess 2003] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2003.1160055.

[Kim et al. 2011] Hyunjoo Kim, Yaakoub el Khamra, Ivan Rodero, Shantenu Jha, and Manish Parashar. Autonomic management of application workflows on hybrid computing infrastructure. *Scientific Programming*, 19(2):75–89, 01 2011. doi: 10.3233/SPR-2011-0319. URL http://dx.doi.org/10.3233/SPR-2011-0319.

[Kim 2003] Kyoung-Jae Kim. Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1–2):307 – 319, 2003. ISSN 0925-2312. doi: http://dx.doi.org/10.1016/S0925-2312(03)00372-2. URL http://www.sciencedirect.com/science/article/pii/S0925231203003722. <ce:title>Support Vector Machines</ce:title>.

[Kim et al. 2013] Myoungjin Kim, Yun Cui, Seungho Han, and Hanku Lee. Towards efficient design and implementation of a hadoop-based distributed video transcoding system in cloud computing environment. *International Journal of Multimedia and Ubiquitous Engineering*, 8(2):213–224, 2013.

[King et al. 2011] T.M. King, A.A. Allen, Yali Wu, P.J. Clarke, and A.E. Ramirez. A comparative case study on the engineering of self-testable autonomic software. In *Engineering of Autonomic and Autonomous Systems (EASe), 2011 8th IEEE International Conference and Workshops on*, pages 59–68, April 2011. doi: 10.1109/EASe.2011.16.

[Klaue et al. 2003] Jirka Klaue, Berthold Rathke, and Adam Wolisz. Evalvid – a framework for video transmission and quality evaluation. In Peter Kemper and WilliamH. Sanders, editors, *Computer Performance Evaluation. Modeling Techniques and Tools*, volume 2794 of *Lecture Notes in Computer Science*, pages 255–272. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40814-7. doi: 10.1007/978-3-540-45232-4_16. URL http://dx.doi.org/10.1007/978-3-540-45232-4_16.

[Knuth 1997] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-89684-2.

[Kohavi 1995] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8.

[Kohavi and Kunz 1997] Ron Kohavi and Clayton Kunz. Option decision trees with majority votes. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, pages 161–169, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. ISBN 1-55860-486-3.

[Koopman 2003] P. Koopman. Elements of the self-healing system problem space. *WADS Workshop on Software Architectures for Dependable Systems*, 2003.

[Koutras and Platis 2007] V. P. Koutras and AN. Platis. Voip availability and service reliability through software rejuvenation policies. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on*, pages 262–269, June 2007. doi: 10.1109/DEPCOS-RELCOMEX.2007.54.

[Krishna 2001] C. M. Krishna. *Real-Time Systems*. John Wiley Sons, Inc., 2001. ISBN 9780471346081. doi: 10.1002/047134608X.W1683. URL http://dx.doi.org/10.1002/047134608X.W1683.

[Kuncheva 2004a] Ludmila I Kuncheva. Classifier ensembles for changing environments. In *Multiple classifier systems*, pages 1–15. Springer, 2004a.

[Kuncheva 2004b] LudmilaI. Kuncheva. Classifier ensembles for changing environments. In Fabio Roli, Josef Kittler, and Terry Windeatt, editors, *Multiple Classifier Systems*, volume 3077 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2004b. ISBN 978-3-540-22144-9. doi: 10.1007/978-3-540-25966-4_1. URL http://dx.doi.org/10.1007/978-3-540-25966-4_1.

[Laddaga 1997] R Laddaga. Darpa broad agency announcement on self-adaptive software, 1997. URL http://www.darpa.mil/ito/Solicitations/PIP_9812.html.

[Laddaga 1999] Robert Laddaga. Guest editor's introduction: Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14(3):26–29, 1999. ISSN 1094-7167. doi: http://doi.ieeecomputersociety.org/10.1109/MIS.1999.769879.

[Lamport et al. 1982] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. ISSN 0164-0925. doi: 10.1145/357172.357176. URL http://doi.acm.org/10.1145/357172.357176.

[Lan et al. 2010] Zhiling Lan, Ziming Zheng, and Yawei Li. Toward automated anomaly identification in large-scale systems. *Parallel and Distributed Systems, IEEE Transactions on*, 21(2):174–187, Feb 2010. ISSN 1045-9219. doi: 10.1109/TPDS.2009.52.

[Lederer et al. 2012] Stefan Lederer, Christopher Müller, and Christian Timmerer. Dynamic adaptive streaming over http dataset. In *Proceedings of the 3rd Multimedia Systems Conference*, MMSys '12, pages 89–94, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1131-1. doi: 10.1145/2155555.2155570. URL http://doi.acm.org/10.1145/2155555.2155570.

[Lee 2005] J. Lee. *Scalable Continuous Media Streaming Systems: Architecture, Design, Analysis and Implementation*. Wiley, 2005. ISBN 9780470857649. URL https://books.google.pt/books?id=7fuvu52cyNEC.

[Lee 1998] J.Y.B. Lee. Parallel video servers: a tutorial. *MultiMedia, IEEE*, 5(2):20–28, Apr 1998. ISSN 1070-986X. doi: 10.1109/93.682522.

[Lee and Leung 2002] J.Y.B. Lee and R.W.T. Leung. Design and analysis of a fault-tolerant mechanism for a server-less video-on-demand system. In *Parallel and Distributed Systems, 2002. Proceedings. Ninth International Conference on*, pages 489–494, Dec 2002. doi: 10.1109/ICPADS.2002.1183446.

[Lee and Wong 2000] J.Y.B. Lee and P.C. Wong. Performance analysis of a pull-based parallel video server. *Parallel and Distributed Systems, IEEE Transactions on*, 11(12):1217–1231, Dec 2000. ISSN 1045-9219. doi: 10.1109/71.895790.

[Lemos 2003] R. Lemos. Icse 2003 wads panel: Fault tolerance and self-healing. In *Proceedings of the ICSE*, 2003.

[Li et al. 2002] Lei Li, K. Vaidyanathan, and K.S. Trivedi. An approach for estimation of software aging in a web server. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, pages 91 – 100, 2002.

[Liang et al. 2006] Y. Liang, Y. Zhang, M. Jette, Anand Sivasubramaniam, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 425 –434, june 2006.

[Littlestone and Warmuth 1989] N. Littlestone and M.K. Warmuth. The weighted majority algorithm. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 256 –261, oct-1 nov 1989.

[Lu and cker Chiueh 2009] Maohua Lu and Tzi cker Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 534–543, June 2009. doi: 10.1109/DSN.2009.5270295.

[Manning et al. 2008] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2008.

[Marcus and Stern 2000] Evan Marcus and Hal Stern. *Blueprints for High Availability: Designing Resilient Distributed Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2000. ISBN 0-471-35601-8.

[Menasce et al. 2011] D. Menasce, H. Gomaa, S. Malek, and J.P. Sousa. Sassy: A framework for self-architecting service-oriented systems. *Software, IEEE*, 28(6):78–85, Nov 2011. ISSN 0740-7459. doi: 10.1109/MS.2011.22.

[Miorandi et al. 2010] Daniele Miorandi, David Lowe, and Lidia Yamamoto. Embryonic models for self–healing distributed services. In Eitan Altman, Iacopo Carrera, Rachid El-Azouzi, Emma Hart, and Yezekael Hayel, editors, *Bioinspired Models of Network, Information, and Computing Systems*, volume 39 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 152–166. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12807-3. doi: 10.1007/978-3-642-12808-0_15. URL http://dx.doi.org/10.1007/978-3-642-12808-0_15.

[Mirgorodskiy et al. 2006] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller. Problem diagnosis in large-scale computing environments. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 11–11, Nov 2006. doi: 10.1109/SC.2006.50.

[Mitra et al. 2011] Siddharth Mitra, Mayank Agrawal, Amit Yadav, Niklas Carlsson, Derek Eager, and Anirban Mahanti. Characterizing web-based video sharing workloads. *ACM Trans. Web*, 5(2):8:1–8:27, May 2011. ISSN 1559-1131. doi: 10.1145/1961659.1961662. URL http://doi.acm.org/10.1145/1961659.1961662.

[Mori et al. 2010] Tatsuya Mori, Ryoichi Kawahara, Haruhisa Hasegawa, and Shinsuke Shimogawa. Characterizing traffic flows originating from large-scale video sharing services. In Fabio Ricciato, Marco Mellia, and Ernst Biersack, editors, *Traffic Monitoring and Analysis*, volume 6003 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12364-1. doi: 10.1007/978-3-642-12365-8_2.

[Mosberger and Jin 1998] David Mosberger and Tai Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, December 1998. ISSN 0163-5999.

[Nafaa et al. 2008] A. Nafaa, T. Taleb, and L. Murphy. Forward error correction strategies for media streaming over wireless networks. *Communications Magazine, IEEE*, 46(1): 72–79, January 2008. ISSN 0163-6804. doi: 10.1109/MCOM.2008.4427233.

[Nagpal et al. 2003] Radhika Nagpal, Attila Kondacs, and Catherine Chang. Programming methodology for biologically-inspired self-assembling systems. In *AAAI Spring Symposium on Computational Synthesis*, 2003.

[Nakatogawa et al. 2006] Y. Nakatogawa, Y. Jiang, M. Kanda, K. Mori, R. Takanuki, and Y. Kuba. Autonomous fault recovery technology for achieving fault-tolerance in video on demand system. In *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, pages 113–120, Dec 2006. doi: 10.1109/ISM.2006.39.

[Nguyen and Zakhor 2002] Thinh Nguyen and Avideh Zakhor. Distributed video streaming with forward error correction. Packet Video Workshop, 2002.

[Ogle et al. 2004] David Ogle, Heather Kreger, Abdi Salahshour, Jason Cornpropst, Eric Labadie, Mandy Chessell, Bill Horn, and John Gerken. Canonical situation data format: The common base event v1.0.1, 2004. URL http://www.eclipse.org/tptp/platform/documents/resources/cbe101spec/CommonBaseEvent_SituationData_V1.0.1.pdf.

[Olups 2010] Rihards Olups. *Zabbix 1.8 Network Monitoring*. Packt Publishing, 2010. ISBN 184719768X, 9781847197689.

[Oppenheimer et al. 2003] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, volume 67. Seattle, WA, 2003.

[Oza and Russell 2001] Nikunj C. Oza and Stuart Russell. Online bagging and boosting. In *In Artificial Intelligence and Statistics*, pages 105–112. Morgan Kaufmann, 2001.

[Padala et al. 2007] P. Padala, X. Zhu, Z. Wang, S. Singhal, K.G. Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Laboratories Technical Report*, 2007.

[Parashar and Hariri 2005] M. Parashar and S. Hariri. Autonomic computing: An overview. *Unconventional Programming Paradigms*, 3566:257–269, 2005.

[Parhami 1994] B. Parhami. Voting algorithms. *Reliability, IEEE Transactions on*, 43(4):617–629, Dec 1994. ISSN 0018-9529. doi: 10.1109/24.370218.

[Pariag et al. 2007] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton. Comparing the performance of web server architectures. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 231–243, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273021. URL http://doi.acm.org/10.1145/1272996.1273021.

[Patterson et al. 2002] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.

[Pearl 1988] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[Peng et al. 2005] H. Peng, Fulmi Long, and C. Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(8):1226–1238, 2005. ISSN 0162-8828. doi: 10.1109/TPAMI.2005.159.

[Pertet and Narasimhan December 2005] Soila M. Pertet and Priya Narasimhan. Causes of failure in web applications. Pdl technical report pdl-cmu-05-109, Carnegie Mellon University, December 2005.

[Pfahringer et al. 2007] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. New options for hoeffding trees. In Mehmet Orgun and John Thornton, editors, *AI 2007: Advances in Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 90–99. Springer Berlin / Heidelberg, 2007.

[Pilgrim 2010] Mark Pilgrim. *HTML5: Up and Running*. O'Reilly Media, Inc., 1st edition, 2010. ISBN 0596806027, 9780596806026.

[Portokalidis and Keromytis 2011] Georgios Portokalidis and AngelosD. Keromytis. Reassure: A self-contained mechanism for healing software using rescue points. In Tetsu Iwata and Masakatsu Nishigaki, editors, *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 16–32. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25140-5. doi: 10.1007/978-3-642-25141-2_2. URL http://dx.doi.org/10.1007/978-3-642-25141-2_2.

[Powers et al. 2005] Rob Powers, Moises Goldszmidt, and Ira Cohen. Short term performance forecasting in enterprise systems. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, pages 801–807, New York, NY, USA, 2005. ACM. ISBN 1-59593-135-X.

[Psaier et al. 2010] Harald Psaier, Florian Skopik, Daniel Schall, and Schahram Dustdar. Behavior monitoring in self-healing service-oriented systems. *2013 IEEE 37th Annual Computer Software and Applications Conference*, 0:357–366, 2010. ISSN 0730-3157. doi: http://doi.ieeecomputersociety.org/10.1109/COMPSAC.2010.43.

[Puri and Ramchandran 1999] Rohit Puri and K. Ramchandran. Multiple description source coding using forward error correction codes. In *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, volume 1, pages 342–346 vol.1, Oct 1999. doi: 10.1109/ACSSC.1999.832349.

[Quinlan 1993] John Ross Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[Reibman et al. 1999] A.R. Reibman, Hamid Jafarkhani, M.T. Orchard, and Yao Wang. Performance of multiple description coders on a real channel. In *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, volume 5, pages 2415–2418 vol.5, 1999. doi: 10.1109/ICASSP.1999.760609.

[Rish 2001] Irina Rish. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46. IBM New York, 2001.

[Robertson et al. 2003] A. Robertson, B. Wittenmark, and M. Kihl. Analysis and design of admission control in web-server systems. In *American Control Conference, 2003. Proceedings of the 2003*, volume 1, pages 254 – 259 vol.1, june 2003. doi: 10.1109/ACC.2003.1238947.

[Robertson and Laddaga 2005] Paul Robertson and Robert Laddaga. Model based diagnosis and contexts in self adaptive software. In Ozalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen, editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 112–127. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26009-7. doi: 10.1007/11428589_8. URL http://dx.doi.org/10.1007/11428589_8.

[Russell et al. 2003] Daniel M Russell, Paul P Maglio, Rowan Dordick, and Chalapathy Neti. Dealing with ghosts: Managing the user experience of autonomic computing. *IBM Systems Journal*, 42(1):177–188, 2003.

[Ruta and Gabrys 2005] Dymitr Ruta and Bogdan Gabrys. Classifier selection for majority voting. *Information Fusion*, 6(1):63 – 81, 2005. ISSN 1566-2535. doi: http://dx.doi.org/10.1016/j.inffus.2004.04.008. URL http://www.sciencedirect.com/science/article/pii/S1566253504000417. <ce:title>Diversity in Multiple Classifier Systems</ce:title>.

[Saha 2007] Goutam Kumar Saha. Self-healing software. *Ubiquity*, 2007(March):4:1–4:1, March 2007. ISSN 1530-2180. doi: 10.1145/1241852.1241853. URL http://doi.acm.org/10.1145/1241852.1241853.

[Sahoo et al. 2010] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226, April 2010. doi: 10.1109/ICCNT.2010.49.

[Sahoo et al. 2003] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proc. of the ninth ACM SIGKDD international conference on knowledge discovery and data mining*, KDD '03, pages 426–435, NY, USA, 2003. ACM. ISBN 1-58113-737-0.

[SANDVINE 2013] Inc ULC SANDVINE. Global internet phenomena report 2013, 2013. URL https://www.sandvine.com/downloads/general/global-internet-phenomena/2013/sandvine-global-internet-phenomena-report-1h-2013.pdf.

[Sapankevych and Sankar 2009] N.I. Sapankevych and Ravi Sankar. Time series prediction using support vector machines: A survey. *Computational Intelligence Magazine, IEEE*, 4(2):24–38, 2009. ISSN 1556-603X. doi: 10.1109/MCI.2009.932254.

[Sapuntzakis et al. 2002] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, December 2002. ISSN 0163-5980. doi: 10.1145/844128.844163. URL http://doi.acm.org/10.1145/844128.844163.

[Schneider et al. 2014] Chris Schneider, Adam Barker, and Simon Dobson. Autonomous fault detection in self-healing systems: Comparing hidden markov models and artificial neural networks. In *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, ADAPT '14, pages 24:24–24:31, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2514-1. doi: 10.1145/2553062.2553065. URL http://doi.acm.org/10.1145/2553062.2553065.

[Schonfeld 2011] E Schonfeld. Netflix now the largest single source of internet traffic in north america, 2011. URL http://techcrunch.com/2011/05/17/netflix-largest-internet-traffic/. (Accessed: 2015-03-05).

[Schulzarine et al. 1996] C.H. Schulzarine, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications, 1996.

[Sheng et al.] Victor S Sheng, Rahul Tada, and Abhinav Atla. An empirical study of class noise impacts on supervised learning algorithms and measures.

[Singh et al. 2000] Raghavendra Singh, Antonio Ortega, Lionel Perret, and Wenqing Jiang. Comparison of multiple-description coding and layered coding based on network simulations, 2000. URL http://dx.doi.org/10.1117/12.382931.

[Sodagar 2011] I. Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *MultiMedia, IEEE*, 18(4):62–67, April 2011. ISSN 1070-986X. doi: 10.1109/MMUL.2011.71.

[Soltesz et al. 2007] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3.

[Steinmetz 1996] R. Steinmetz. Human perception of jitter and media synchronization. *Selected Areas in Communications, IEEE Journal on*, 14(1):61 –72, jan 1996. ISSN 0733-8716. doi: 10.1109/49.481694.

[Stockhammer 2011] Thomas Stockhammer. Dynamic adaptive streaming over http –: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, pages 133–144, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0518-1.

[Sultan et al. 2002] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: connection migration for service continuity in the internet. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 469 – 470, 2002.

[Summers et al. 2012a] Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. To chunk or not to chunk: Implications for http streaming video server performance. In *Proceedings of the 22Nd International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV '12, pages 15–20, New York, NY, USA, 2012a. ACM. ISBN 978-1-4503-1430-5. doi: 10.1145/2229087.2229093. URL http://doi.acm.org/10.1145/2229087.2229093.

[Summers et al. 2012b] Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. Methodologies for generating http streaming video workloads to evaluate web server performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 2:1–2:12, New York, NY, USA, 2012b. ACM. ISBN 978-1-4503-1448-0. doi: 10.1145/2367589.2367602. URL http://doi.acm.org/10.1145/2367589.2367602.

[Systems 2013] Cisco Systems. Cisco visual networking index: Forecast and methodology, 2013–2018, 2013. URL http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf.

[Tamura 2008] Yoshi Tamura. Kemari: Virtual machine synchronization for fault tolerance using domt, June 2008.

[Tan and Gu 2010] Yongmin Tan and Xiaohui Gu. On predictability of system anomalies in real world. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 133–140, 2010. doi: 10.1109/MASCOTS.2010.22.

[Tan et al. 2010] Yongmin Tan, Xiaohui Gu, and Haixun Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 173–182, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9.

[Theimer et al. 1985] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, SOSP '85, pages 2–12, New York, NY, USA, 1985. ACM. ISBN 0-89791-174-1. doi: 10.1145/323647.323629. URL http://doi.acm.org/10.1145/323647.323629.

[Thissen et al. 2003] U Thissen, R van Brakel, A.P de Weijer, W.J Melssen, and L.M.C Buydens. Using support vector machines for time series prediction. *Chemometrics and Intelligent Laboratory Systems*, 69(1–2):35 – 49, 2003. ISSN 0169-7439. doi: http://dx.doi.org/10.1016/S0169-7439(03)00111-4. URL http://www.sciencedirect.com/science/article/pii/S0169743903001114.

[Tumer and Ghosh 1996] K. Tumer and J. Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection science*, 8(3-4):385–404, 1996.

[Villegas et al. 2013] NorhaM. Villegas, Gabriel Tamura, HausiA. Müller, Laurence Duchien, and Rubby Casallas. Dynamico: A reference model for governing control objectives and context relevance in self-adaptive software systems. In Rogério de Lemos, Holger Giese, HausiA. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 265–293. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35812-8. doi: 10.1007/978-3-642-35813-5_11. URL http://dx.doi.org/10.1007/978-3-642-35813-5_11.

[Vin et al. 1994] H. Vin, P. Goyal, and A. Goyal. A statistical admission control algorithm for multimedia servers. In *Proceedings of the Second ACM International Conference on Multimedia*, MULTIMEDIA '94, pages 33–40, New York, NY, USA, 1994. ACM. ISBN 0-89791-686-7. doi: 10.1145/192593.192616. URL http://doi.acm.org/10.1145/192593.192616.

[Wang et al. 2002] Bing Wang, S. Sen, M. Adler, and D. Towsley. Optimal proxy cache allocation for efficient streaming media distribution. In *INFOCOM 2002. Twenty-First*

Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 3, pages 1726–1735 vol.3, 2002. doi: 10.1109/INFCOM.2002.1019426.

[Wijesekera et al. 1999] Duminda Wijesekera, Jaideep Srivastava, Anil Nerode, and Mark Forrsti. Experimental evaluation of loss perception in continuous media. *Multimedia Systems*, 7(6):486–499, November 1999. ISSN 0942-4962. doi: 10.1007/s005300050149. URL http://dx.doi.org/10.1007/s005300050149.

[Witten et al. 2011] Ian H Witten, Eibe Frank, and Mark A Hall. *Data Mining: Practical Machine Learning Tools and Techniques: Practical Machine Learning Tools and Techniques*. Elsevier, 2011.

[Xie et al. 2007] M. Xie, Q. P. Hu, Y. P. Wu, and S. H. Ng. A study of the modeling and analysis of software fault-detection and fault-correction processes. *Quality and Reliability Engineering International*, 23(4):459–470, 2007. ISSN 1099-1638. doi: 10.1002/qre.827. URL http://dx.doi.org/10.1002/qre.827.

[Yang et al. 2007] Lingyun Yang, Chuang Liu, Jennifer M. Schopf, and I Foster. Anomaly detection and diagnosis in grid environments. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–9, Nov 2007. doi: 10.1145/1362622.1362667.

[Yoshimura et al. 2011] T. Yoshimura, H. Yamada, and K. Kono. Can linux be rejuvenated without reboots? In *Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on*, pages 50–55, Nov 2011. doi: 10.1109/WoSAR.2011.12.

[Yu et al. 2006] Hongliang Yu, Dongdong Zheng, Ben Y. Zhao, and Weimin Zheng. Understanding user behavior in large-scale video-on-demand systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 333–344, New York, NY, USA, 2006. ACM. ISBN 1-59593-322-0. doi: 10.1145/1217935.1217968. URL http://doi.acm.org/10.1145/1217935.1217968.

[Zayas 1987] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, SOSP '87, pages 13–24, New York, NY, USA, 1987. ACM. ISBN 0-89791-242-X. doi: 10.1145/41457.37503. URL http://doi.acm.org/10.1145/41457.37503.

[Zhang 2004] Harry Zhang. The optimality of naive bayes. In Valerie Barr and Zdravko Markov, editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*. AAAI Press, 2004.

[Zhao et al. 2012] Zhijie Zhao, Gunwoo Kim, Doug Young Suh, and J. Ostermann. Comparison between multiple description coding and forward error correction for scalable video coding with different burst lengths. In *Multimedia Signal Processing (MMSP), 2012 IEEE 14th International Workshop on*, pages 37–42, Sept 2012. doi: 10.1109/MMSP.2012.6343412.

[Zheng and Webb 2010] Fei Zheng and Geoffrey I Webb. Tree augmented naive bayes. In *Encyclopedia of Machine Learning*, pages 990–991. Springer, 2010.

[Zheng et al. 2006] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, and M.A Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4):240–253, April 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.38.

[Zink et al. 2008] Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. Watch global, cache local: Youtube network traffic at a campus network: measurements and implications, 2008. URL http://dx.doi.org/10.1117/12.774903.

[Zink et al. 2009] Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. Characteristics of youtube network traffic at a campus network – measurements, models, and implications. *Computer Networks*, 53(4):501 – 514, 2009. ISSN 1389-1286. Content Distribution Infrastructures for Community Networks.

Coimbra, September 2015
Author's Edition.